Attack the Glass (ATG): Technical

Technical Whitepaper on the Attack the Glass (ATG) Vulnerability

v2.4.7, September 14, 2026

TLP:CLEAR

Contact: Robert Beckett

Signal: ndiligence.99

Email: atg@nDiligence.com

Table of Contents

Executive Summary 7
Part I - The Vulnerability and the Adversary 15
Chapter 1: Client-Side Rendering (CSR) 16
1.1: The Client Runtime 18
1.2: Client-Side Rendering 19
1.3: The Move from Server to Client 20
1.4: Why CSR Won 21
1.5: The Application Blueprint 21
1.6: The Perpetual State Machine 23
1.7: Code and Data on Different Paths 24
1.8: Verification Ends at Delivery 25
1.9: The Allowlist Fallacy 27
1.10: The Manifest Illusion 28
1.11: Data as Code 29
1.12: Code That Writes Code 30
1.13: The Glass Beyond the Browser 30
1.14: Malice Without Malware 31
1.15: CSR and Client Runtime AI 33
Chapter 2: The ATG Vulnerability Architecture 34
2.1: The Internet Is a Supply Chain 34
2.2: The Most Important Supply Chain Is the Least Protected 35
2.3: The Digital Supply Chain 36
2.4: The Glass: The Final Mile of Every Digital System 38
2.5: Integration on the Glass (IOTG) 39
2.6: URL Overtrust 40
2.7: The Static Resource Assumption 42
2.8: The Digital Supernode 44
2.9: The ATG Vulnerability Defined 46

2.10: Supply Chain Injection 48
2.11: The Two-Stage Attack 49
2.12: Content Swapping 52
2.13: Post-Load and Post-Trust Modification 54
2.14: Targeted Delivery 56
2.15: The Integrity Blindspot 58
2.16: ATG Applies to All Networks, Not Just the Internet 60
2.17: D5 Effects 61
2.18: Scope and Exclusions 66
2.19: ATG Beyond the Browser 66
2.20: ATX: Beyond the Glass 68
Chapter 3: The Adversary: Capability and Operations 71
3.1: ATG Adversary Capability 72
3.2: The Commercialization Gray Zone 75
3.3: ATG Operational Patterns 78
3.4: The AI Amplification Factor 80
3.5: Placement, Access, Influence, and Control 82
Part II - The Surface Landscape 84
Chapter 4: Languages and Runtimes 85
4.1: Why the Language Matters 85
4.2: What Languages Allow 86
4.3: User Experience Attack Surface by Language 91
4.4: Notable Language Profiles 93
4.5: Defense Across Languages 95
Chapter 5: The Platform Landscape 98
5.1: Why the Platform Matters 99
5.2: What Makes a Platform an ATG Surface 100
5.3: Device Scale and Platform Reach 102
5.4: The Ten Platform Categories 103

5.5: Cross-Platform Patterns 108
5.6: Defense Across Platforms 109
Chapter 6: The Provider Landscape 111
6.1: Why the Provider Matters 112
6.2: What Makes a Provider an ATG Surface 113
6.3: Provider Reach and Defendability 116
6.4: The Twenty Provider Categories 117
6.5: Cross-Provider Patterns 122
6.6: Defense Across Providers 124
Chapter 7: Content Delivery Networks 126
7.1: Why CDNs Sit Where They Sit 126
7.2: CDN Categories and the Trust Hierarchy 128
7.3: The CDN Acquisition Attack Model 129
7.4: CDN as Risk Multiplier 132
7.5: Lying in Wait 133
7.6: Typosquatted and Impersonating CDN Domains 135
Chapter 8: Third-Party Script Loading: The Direct Trust Problem 137
8.1: Script Loading as Runtime Delegation 137
8.2: The Third-Party Script Inventory Problem 138
8.3: High-Trust Script Categories 139
8.4: Runtime Anarchy 139
8.5: Scale of Exposure Across Application Categories 141
Chapter 9: Open-Source and the Path to the Client Runtime 143
9.1: Why Open-Source Matters for ATG 143
9.2: The Open-Source Open Door 144
9.3: The Path from Package to Rendering Surface 146
9.4: Origin Laundering 147
9.5: Plugin and Extension Marketplaces 148
9.6: No Verification at the Client Runtime Boundary 150

Chapter 10: AI in the Client Runtime 151
10.1: Why AI Matters for ATG 151
10.2: The Path from Model to Client Runtime 152
10.3: The Inference Response as Data Endpoint 153
10.4: The Model as Compositional Surface 154
10.5: Local AI and the Detection Blind Spot 155
10.6: Tailored Deception 155
10.7: Deceiving the Model 157
10.8: No Defense at the Client Runtime AI Boundary 158
Part III - The Attack Mechanism 160
Chapter 11: Placement and Access (P&A) 161
11.1: P&A Defined and the Causal Chain 161
11.2: Levels of Control 162
11.3: P&A as Injection Point in Supply Chains 163
11.4: The Influence and Control Monitoring Imperative 163
11.5: APA and AIO 163
11.6: The Activation Event 164
11.7: The Scope Hierarchy 164
11.8: The APA Vector Landscape 165
11.9: P&A Acquisition Pathways 167
11.10: Operational Characteristics 168
11.11: The Cross-Surface Attack Model 168
Chapter 12: Ownership Opacity: The Visibility Problem 170
12.1: Why Ownership Matters for ATG 170
12.2: The Public-Information Floor 171
12.3: The Adversarial Asset Cycling Pattern 173
12.4: The Ownership-Tracking Capability Gap 174
Part IV - Evidence and Existing Defenses 176
Chapter 13: The Confirmed Historical Record: Summary Reference 179

13.1: What the Confirmed Record Establishes Analytically 179
13.2: ATG-Class Event Categories and Their Structural Significance 180
Chapter 14: Existing Defenses: Partial Measures and Structural Limits 183
14.1: The Structural Limits of Existing Defenses 183
14.2: Content Security Policy: Value and Limits 184
14.3: Subresource Integrity: Value and Limits 185
14.4: Zero Trust: Why ATG Operates Beneath It 186
14.5: Server-Side Rendering as Partial Mitigation 187
14.6: Defenses Outside the Browser, and the Endemic Flaw 188
Chapter 15: Relationship to Existing Frameworks 191
15.1: Why ATG Does Not Appear in CVE/NVD 191
15.2: ATG Mapped to MITRE ATT&CK 192
15.3: ATG and NIST SP 800-161 193
15.4: ATG and the Sectoral Regulatory Frameworks 193
15.5: Standards Gaps ATG Exposes 194
Part V - Reference Methodology 195
Chapter 16: Severity Methodology and Reference Matrix Guide 196
16.1: The Severity Ramp and Color Encoding 196
16.2: Inclusion Criterion and Scope Boundary 197
16.3: The Exclusion of Server-Side Runtime Scripting and SSTI from ATG 198
16.4: The Four Scoring Dimensions 199
16.5: User-Experience Prevalence (Languages Sheet) 200
16.6: Device Scale (Platforms Sheet) 201
16.7: D5 Effects as a Cross-Sheet Score 201
16.8: Critical-Count Summaries 202
16.9: Caveats and Review Cadence 203
16.10: The Workbook, Sheet by Sheet 203
Closing 205

Executive Summary

Attack the Glass (ATG) is a supply-chain injection vulnerability that targets client-side rendering, the step in which a platform assembles what the user sees on the user's own device. It is unmitigated and underappreciated, and attacks that exploit it reach modern technology platforms in every nation, across every industry, and across every type of network, from the open internet to classified systems. These attacks can be used for both large-scale campaigns and precise targeting. They can be deployed from millions of positions across the global digital supply chain with a traffic pattern indistinguishable from legitimate operation, and they leave no forensic evidence in the backend systems, network logs, and audit trails that existing security frameworks monitor. The gap they exploit is client-side rendering itself, and it is present even in platforms built on Zero-Trust architecture.

Client-Side Rendering (CSR) is the technique by which modern technology platforms assemble what the user sees at runtime, on the user's own device. The pattern is universal across the deployed landscape: web browsers, mobile applications, desktop application shells, smart appliances, vehicle infotainment systems, AR and VR runtimes, productivity-tool extensions, and the broader landscape of CSR-capable platforms all build their user interfaces this way. The runtime fetches dozens or hundreds of components from across the digital supply chain, executes the code those components return, and renders the result on the screen. This is where the person meets the system, and it is the step that matters most: everything the user comes to know about the system, and every decision the user makes from it, comes from what appears here. That surface, the last thing standing between the software and the user's eyes, is the Glass. These vulnerabilities attack this moment, the last few milliseconds in which client-side rendering produces what informs the user's understanding and drives the user's decisions. An attack that exploits that moment is an ATG attack.

ATG is not an unprecedented exploit. What is new is the connection between problems the field has so far handled as scattered, unrelated point problems. Its parts are individually familiar: client-side supply-chain security and client-side attack-surface management address third-party script risk; the Magecart intrusions have skimmed data from the rendering layer for a decade; and cloaking, serving one response to a scanner and a different response to a victim, is long established in malvertising and search abuse. What ATG adds is one account that covers all of it. It unifies these into a single supply-chain-injection model, extends that model from the browser across the full landscape of Client Runtimes (mobile, desktop, operational-technology and vehicle interfaces, AR and VR, and productivity and AI runtimes), and names what the point defenses miss: the gap between

where a file is checked and where it runs. ATG is not a single new bug. It is one problem that current defenses meet one surface at a time.

No adequate, complete defense against ATG currently exists; the partial measures deployed today (CSP, SRI, client-side monitoring, Trusted Types) each check what loads, and ATG acts after the load. The cybersecurity frameworks deployed across the internet today are not poorly designed. They were built for a different threat environment: the threat of malicious code making its way into systems where it does not belong. Checking a delivered file against a known hash, network controls, supply-chain attestation (the signed record of how a component was built and by whom), and Zero-Trust architecture each address that threat well. ATG operates in the rendering layer, between the last point those frameworks monitor and the first point that has any consequence for the human at the screen. The most sophisticated form of ATG defeats file-hash checking by design, because the file it arrives in is genuinely unmodified. Behavioral monitoring at the rendering layer would catch it. That defense does not exist at scale.

Defenders are unprepared to respond to and recover from a wide-scale ATG campaign. The damage an ATG-enabled influence operation can do is not confined to the systems it runs on. A well-executed operation could undermine public trust in institutions across nearly every industry, government, and critical system. The same techniques could equally be used to hijack trust and persuade a population to believe and support a chosen narrative. Used at scale, ATG can create widespread distrust, erode public confidence, undermine market integrity, and significantly diminish governmental authority.

Operations targeting public trust at scale are already running. The Doppelganger campaign, Russia-attributed cloned versions of major Western news outlets distributing disinformation across multiple countries since 2022, demonstrates the demand and the scale. Doppelganger relies on visible infrastructure such as spoof domains and separate distribution channels that defenders can sometimes detect and users can sometimes notice. ATG removes that scaffolding. The same effects can be produced through the legitimate trusted sources themselves, with no separate infrastructure for defenders to find or for users to recognize as suspect.

ATG is, on its own, a serious and unmitigated vulnerability. ATG is also a delivery mechanism: it can inject any content, at scale and with precision, into the most trusted technology platforms on Earth. A Doppelganger-class operation running on that infrastructure, paired with AI-generated content tailored per target, becomes a weapon of a different magnitude entirely. The result is a new class of adversarial influence: weapons of mass deception.

THE VULNERABILITY

Modern client applications are not delivered as fixed products. They are assembled at runtime on the user's device from dozens to hundreds of externally sourced components. When a Client Runtime loads a page, an application, an interface, or a screen, it makes network requests to CDN operators, analytics providers, advertising networks, customer-support tools, payment processors, identity providers, AI inference endpoints (an inference endpoint is the address an application calls to get an answer from a model), and a wide range of commercial service integrations, executing whatever code those sources return with the full permissions of the application context. This pattern of runtime assembly is Integration on the Glass (IOTG). It is the industry standard for every significant client application in the world, and it is permanent: the cost pressures and engineering pressures that produced it are built into how modern software is made.

A device trusts a source because of the address it fetched from, not because of what came back. Once a device is configured to trust a domain, it will execute whatever that domain returns, and it never checks whether today's response matches yesterday's, or matches what that domain returns to anyone else making the identical request. The content returned in response to a URL call is determined entirely by the server receiving the request. That server can return different content to different requesters based on any attribute of the incoming request: the requesting device's IP address, the User-Agent string, the referring page, the time of the request, the geographic location of the device, the network through which it connects, or any combination of these attributes. This capability is the legitimate foundation of personalization, A/B testing, and geographic content variation. It is also the technical mechanism through which ATG attacks are delivered.

The device is not choosing. The application author decides which third parties the device will trust, encodes that decision in the code it ships, and the device executes it. The person in front of the screen is not shown that list, is not asked to agree to it, and cannot decline any entry on it without losing the application. The application chooses, and the choice runs on the user's device.

An adversary who controls any component in the digital supply chain of a target application can use this mechanism to deliver substituted content to specific requesters while delivering legitimate content to everyone else. By reading those request attributes, an adversary can aim at the full user population of a global application, at a country or language community, at a segment defined by behavior or demographics, or at one device operated by one named individual, and can serve different content to several of those groups in the same session window. The substituted content can be any type the runtime processes: executable scripts, stylesheets, images, data payloads, model weights, prompt

content. The substitution is delivered through a trusted channel, executed by the runtime without verification, displayed on the Glass without any indication that it differs from what anyone else sees, and leaves no evidence in any backend system, any network log, or any audit trail any existing security framework monitors. The structural condition any ATG operation requires is Adversarial Placement and Access (APA): a party hostile to the dependent systems holding a position somewhere along that supply chain. APA does not come from breaching the target. It comes from holding something the target depends on, often something the target does not know it depends on.

ATG's most sophisticated form is the Two-Stage Attack. A script that is entirely legitimate, verifiably unmodified, and hash-valid carries the attack not through its own code but through the runtime data it fetches at the moment of execution from a data endpoint the adversary holds. The script passes every static integrity check. The attack is in what the script does with the data it receives. This defeats every defense built on checking the delivered file: the file really is clean. The weapon is in the data, not the code.

ATG is not limited to the public internet. Classified networks, military intranets, air-gapped operational technology environments, and private enterprise networks are built from components sourced from the same digital supply chain. A component can be compromised in the supply chain before anyone installs it inside the isolated environment. Isolation applied afterward does nothing about code that was already inside when the isolation began. No network is structurally immune to ATG by isolation alone.

The Threat

The D5 effects framework names the spectrum of outcomes ATG attacks can produce: Deceive, Disrupt, Degrade, Deny, and Destroy. These five effects are not separate attack types, and they are not five degrees of one thing. Each is fully capable alone. They are capabilities of a single weapons framework, applied individually or combined, layered, and sequenced to produce strategic consequences far exceeding the technical complexity of any one effect.

Deceive is the capability at the center of the framework: causing a consumer to accept as true something the system did not intend to show. That consumer is usually a person, manipulated through what they see on the Glass in ways they have no reason to question. It can equally be an automated one. A machine does not judge whether its input is true; it acts on what it is given, which is why falsifying a process value can stop an alarm evaluation from ever escalating, and why a model trained or prompted on a corpus of lies will state those lies as fact. Where the manipulation is delivered to selected recipients, everyone else continues to see the original content, fact-checkers examining the same

source see the accurate version, and the target has no basis for suspecting that their experience differs from anyone else's.

Generative AI extends Deceive into dynamic, contextually adaptive deception. Substituted content is generated in real time from what the target is viewing, producing manipulations that are more convincing, better fitted to the moment, and harder to detect than any pre-prepared substitution. Deepfake video and audio injected into web-based conference platforms, AI-generated modifications to news articles and research documents, and AI-generated alterations tailored to the source materials a targeted journalist or analyst is reviewing are all within the technical scope of an AI-equipped ATG operation. The Hong Kong $25 million video-conference fraud in early 2024 shows this deepfake capability at eight-figure stakes; in that case no conferencing platform was compromised, so it illustrates the payload an ATG operation would carry rather than the supply-chain injection mechanism itself. We call this capability *Tailored Deception*.

Deceive is also where the gravest cases sit. An operator who executes a switching action based on a false grid-state display, a financial officer who executes a transaction based on a false position summary, a system administrator who follows a false security alert's remediation procedure: each is an authorized user taking an authorized action in response to information they had every reason to trust. The manipulation was in the information. The damage that follows is its consequence, and recording it that way keeps the mechanism visible: what failed was not the system, but what the system was believed to have said.

Disrupt interferes with an intended function: the capability is still present and does not do what it is for. A password reset that completes and sends nothing, an alarm control that fires and reaches no one, a report that submits into a queue nobody works. Degrade reduces volume, speed, and scale: everything works, and works worse. Deny removes reach at the Glass: the control is gone, the route does not resolve, the returned set is truncated, and every backend system stays healthy throughout. Destroy is the only one of the five that leaves the Glass: adversary-controlled code reaches through the rendering layer into a backend or a connected system and damages it, so that it cannot function until it is restored or rebuilt. Deny acts on the Glass and Destroy goes through it, and where a person was deceived into taking the damaging action the effect is Deceive with the damage as its consequence, not Destroy.

Each of the four can be aimed at an entire user base, at a country or language community, at a segment defined by behavior or demographics, or at one device, and different groups can be served different content at the same moment. Reach is a property of how an operation is delivered, not of which effect it produces.

The press is the most consequential ATG target. It is the mechanism through which democratic societies form shared understanding, evaluate institutions, and exercise informed judgment. Doppelganger's targeting of major Western news outlets confirms that adversaries already act on that priority. An adversary who can manipulate what journalists see when researching a story, what editors see when reviewing it, and what readers see when consuming it from a trusted platform has achieved precision control over the most influential participants in the information production and distribution chain.

Government and national-security platforms carry the same supply-chain dependencies as every other client application, and legislators, intelligence analysts, election administrators, diplomats, law-enforcement officers, and emergency-management officials decide from what those platforms put on the screen. Defense-industrial-base systems, military command interfaces, and allied coordination platforms carry the same exposure into the most sensitive operational environments on Earth.

Critical-infrastructure operator interfaces, the screens through which human operators monitor and command the physical systems populations depend on for survival, are assembled from the same commercial digital supply chain as every consumer application. Well-run industrial control environments narrow what a runtime in that environment is exposed to. URL pinning, mutual TLS, restricted runtime parsing engines, network segmentation, and operator-workstation hardening each constrain what reaches the rendering layer of an HMI, the human-machine interface an operator watches and acts through, compared with what reaches a commercial browser. None of those measures reach back to the sources the components come from. None of them establishes that a permitted resource returned what the interface expected, and an operator interface exists precisely to render live data it fetches while running. The runtime libraries, framework dependencies, telemetry agents, visualization components, and update channels reach the HMI through the same upstream supply chain that reaches every other client application, and a compromise introduced before the HMI's hardened perimeter is in place is irrelevant to the perimeter that follows. The glass that shows an energy-grid operator the state of the grid they manage is hardened where a social-media feed is not and structurally exposed at the sources its components come from, which the perimeter cannot reach.

The pattern is not confined to a few sectors. It runs across energy and industrial control, healthcare, financial services, transportation, and government and election systems, and it reaches the defense industrial base, communications, and emergency services on the same terms. The common factor is not the sector. It is that each of them puts a person in front of a client-side rendered screen and asks them to decide.

The most strategically significant ATG threat is the compound, long-horizon operation. No single event marks it. Consequences accumulate through individually unremarkable micro-manipulations over weeks, months, or years, gradually corrupting the information environments of the most consequential decision-makers. Those decision-makers never have reason to suspect that what they see differs from what anyone else sees.

The Defense Gap

Existing frameworks are structurally blind to ATG. The allowlist assumption, the foundational premise of network security that trusted addresses return trusted content, is false at the rendering layer. It conflates source authentication, which HTTPS provides, with content integrity, which no deployed framework provides. Zero-Trust Architecture ends at the point of load, before the rendering layer begins. Subresource Integrity verifies the delivered file, not the data that file fetches and acts on while it runs. Adoption has been further limited because keeping those hashes current conflicts with continuous delivery, where the files change on every release. Content Security Policy evaluates sources, not substance. Network monitoring cannot see inside encrypted connections to permitted destinations. Backend integrity monitoring cannot see the gap between accurate data and false display. The CVE framework has no category for attacks that exploit no software flaw.

Runtime code injection is the most demanding dimension of the defense challenge. Pathways include dynamic code evaluation, dynamic script element injection, WebAssembly instantiation from runtime data, Service Worker persistence, AI inference response interpretation, and the data-as-payload Two-Stage Attack. Each requires behavioral observation, not static analysis, for accurate detection. The classification problem is harder still: telling adversarial behavior from legitimate behavior requires knowing both what the code does and what it was supposed to do, and no application publishes a description of its intended runtime behavior in any standard form.

These limits converge because ATG is inherent to client-side rendering: the vulnerability comes from the architecture, and it appears on its own in every runtime that uses the pattern. That is the Endemic Flaw. Defensive maturity is concentrated in the browser; outside the browser, the rendering layer is mostly undefended. The defense must be built where ATG operates: inside the Client Runtime itself, at the rendering layer, in every platform that builds its interface at runtime from the supply chain. Browser extensions and external tools cannot see far enough inside the runtime to detect it.

ATG is real and operational: Polyfill.io, Magecart, XZ Utils, the state-run Great Cannon, and the nine ATG-class event categories documented in the public record establish that. The full scale today cannot be measured: deception, by design, leaves no signal signature-

based detection can find. The architectural conditions for nation-state and other-resourced operations are present, and the prudent posture is to assume the capability is in use.

Closing the gap means reclassifying what counts as malicious. Existing defenses look for malicious code from malicious sources; ATG operates through code and sources no existing test would flag. The behavioral defense that follows does not yet exist. Building it is the work that determines whether the weapons-of-mass-deception scenario remains theoretical or becomes the defining strategic surprise of the next decade.

Part I - The Vulnerability and the Adversary

ATG begins with an architectural choice: applications assemble themselves on the user's device. It becomes a vulnerability when that architecture meets a supply chain. It becomes an operational threat when an adversary holds a position in that supply chain.

Client-Side Rendering is the root architectural pattern that produces the modern Client Runtime: the technique by which every modern technology platform, from web browsers to vehicle infotainment systems, assembles itself at runtime on the user's device. The Client Runtime Boundary names the moment content that has been loaded starts running as code. The Perpetual State Machine names what the assembled application does after that moment: unlike a finite state machine, whose states and transitions are fixed in advance, countable, and testable, it never settles into a fixed set of states.

The ATG vulnerability architecture rests on the Glass and Integration on the Glass, with Supply Chain Injection as the general mechanism, the Two-Stage Attack, the formal definition of ATG, and the D5 effects spectrum. Content Swapping names what changes on the rendered surface, Post-Load and Post-Trust Modification names when it changes, Targeted Delivery names who receives it, and the Integrity Blindspot names why so little of it is seen.

The adversary is described along eight capability dimensions. Further conditions shape what that adversary can do: the commercialization gray zone, the local-AI detection blind spot, and the Placement, Access, Influence, Control causal chain.

Chapter 1: Client-Side Rendering (CSR)

In 2006, an invention sparked a change in how the internet works: the Infinite Scroll. Aza Raskin's design removed the page boundary; as the user reached the bottom of what was loaded, more content appeared, fetched on demand and rendered in place. The pattern spread quickly. Twitter, Facebook, Pinterest, and most of the consumer web adopted variations within a few years. To deliver that experience, applications had to become dynamic and responsive to what each user was doing in real time. This led to the creation of client-side rendering platforms and, fundamentally, changed the way internet-enabled technology is built. It also created a shift in the internet supply chain. The most consequential effect of that shift sits at the layer where the user actually meets the system.

The Client Runtime is the control layer on the client device. Server logic, network paths, and back-end services all matter to the system's correctness; the user only ever experiences the system as it appears in the Client Runtime. It is also the least protected layer in the modern technology stack.

Attack the Glass (ATG), in its most direct framing, is an attack on the Client Runtime. The Client Runtime draws the Glass the user interacts with in real time, and it sits in the one position every other layer of the stack assumes someone else has already secured. That vulnerability is structural: it comes from the technique that produces the modern Client Runtime, Client-Side Rendering.

Client-Side Rendering (CSR) is the technique by which a Client Runtime assembles and modifies its execution at runtime rather than at build time. Older client environments could only present a user experience compiled from fixed source. That limitation was removed decades ago from modern programming languages and runtimes. Two users running the same application on the same device against the same originating source can have different runtime experiences depending on who they are, where they are, what they are doing, and other contextual conditions. Modern applications are not uniform and procedural. They are reactive and dynamic to the individual user.

JavaScript executing in a web browser is the most common and visible expression of CSR, and the web carries by far the largest CSR-capable installed base. CSR is not specific to the browser; the technique drives mobile, desktop-shell, immersive, embedded, and productivity runtimes across the deployed landscape. Wherever a runtime is structured to load or interpret content at runtime that influences what is rendered or what executes, CSR is the technique in use, and the Client Runtime is exposed to ATG.

Not every Client Runtime is exposed. Precompiled-only Client Runtimes (some embedded firmware, certain real-time control loops, certain safety-critical avionics) are out of scope. These Client Runtimes are immune to ATG-style attacks because the language and platform do not support dynamically loading alternative code or modifying user interfaces at runtime.

Two pathways carry the runtime modification that creates that exposure. The first is direct: the runtime loads executable code over the network and incorporates it into the running application. The second is indirect: content interpreted at runtime generates further executable behavior the runtime then runs. Both produce runtime code execution, but only the first pathway looks like "loading code" to a defender.

CSR-capable Client Runtimes are unprotected, unmonitored, and exposed. Thousands of attack vectors target millions of sites, applications, and source repositories that feed those runtimes. The attacks infiltrate Client Runtimes across a wide range of devices and platforms, leave nearly no forensic signature or attribution behind, and meet no runtime behavioral monitoring at the rendering surface. Checking a delivered file against a known hash remains valuable, but that check stops at the moment of delivery and never reaches the running application. The result is a structural blind spot in the integrity of modern computing devices: an open path to modify the information rendered on the screen, the interface the user interacts with, and the decisions the user makes by trusting what they see on the Glass.

Within that exposure, an attack-mode hierarchy holds. The most likely and by far the most dangerous category of ATG attack is the modification of information itself: the data, content, media, and features the user sees. Functional disruption (application failure, alert suppression, behavioral subversion of features) sits above information modification in technical sophistication and below it in frequency, payoff, and damage. The D5 effects framework formalizes this hierarchy.

CSR is not a defect to be patched out. It is the architectural pattern that produced the shift from server-rendered to client-rendered computing, the shift that enabled infinite scroll, reactive interfaces, data-driven user experiences, and the modern application stack. There is no return to server-only rendering at scale. The defensive posture must be purpose-built for the CSR-capable Client Runtime.

1.1: The Client Runtime

The Client Runtime is responsible for assembling, executing, and rendering the user's experience of the system. Everything the user encounters at the device passes through it: what is shown on the screen, which controls work and what they do, what background processes run, what information is collected, and whether alerts and warnings surface or stay silent. Every other layer reaches the user through it. The user only ever experiences the system as the Client Runtime presents it.

Client Runtimes fall into two structurally distinct categories.

The first is the CSR-capable Client Runtime: a runtime that can load, interpret, or generate content at runtime in ways that influence what is rendered or what executes. Modern web browsers, mobile-application runtimes, hybrid desktop shells (Electron, Tauri), AR and VR runtimes, smart-TV applications, vehicle infotainment systems, IoT and operational technology dashboards, and productivity-tool extension hosts all fall into this category. The CSR-capable Client Runtime is where ATG lands.

The second is the precompiled-only Client Runtime: a runtime that loads and executes only precompiled code, with no mechanism for loading or interpreting runtime scripts and no mechanism for content-driven runtime modification. Some embedded firmware, certain real-time control loops, certain safety-critical avionics, and a small set of hardened or minimal runtimes fall into this category. Precompiled-only Client Runtimes are not exposed to ATG. They cannot load runtime scripts, and they cannot modify behavior in response to fetched content.

The distinction is operational, not theoretical. Whether a given platform is in scope depends on what its Client Runtime actually does: whether it loads runtime scripts, accepts runtime configuration, evaluates content as code, or interprets data in ways that alter behavior. A Client Runtime that is technically capable of CSR but configured to load no runtime resources is a low-exposure case. A Client Runtime that loads dozens of runtime resources from many providers is a high-exposure case. The same platform can sit anywhere along that spectrum depending on configuration and use. Exposure is a property of what the Client Runtime actually loads, not of the platform category it belongs to.

Inside the Client Runtime, the Client Runtime Boundary is the moment a fetched file stops being something a defender can open and inspect and starts being code that is running. The supply-chain controls that defenders rely on operate before that boundary. After the boundary, that code holds whatever permissions the host application holds, it can reach for further code of its own, and the conditions ATG needs are all in place. The boundary is the same regardless of platform.

1.2: Client-Side Rendering

Client-Side Rendering is the technique by which final assembly and execution of an application happen on the client device, inside the Client Runtime, rather than on the server. The application as it executes is composed at runtime from sources that may include the base application, scripts and modules fetched at runtime, configuration retrieved at startup, content and data loaded continuously during the session, model weights and prompts in AI-augmented runtimes, and any other resource the runtime is structured to consume. Three primitives are central: the runtime, the rendering surface, and the assembled application.

CSR is a design approach, not an exclusive technique of any single engine. Across all platforms that implement it, user experience and executable behavior are constructed at runtime, not fixed at build time. Mobile-application rendering frameworks, hybrid desktop shells, AR and VR scene graphs, smart-TV runtimes, vehicle infotainment, IoT and OT dashboards, productivity extensions, plugin hosts, and notebook front-ends are all CSR runtimes. JavaScript in the browser is the most common expression because the web carries by far the largest installed base; the browser is not the limit of CSR.

Two pathways carry the runtime modification that distinguishes CSR from server-rendered or precompiled execution.

The first pathway is direct: the runtime loads source code, modules, scripts, or other executable files from the network at runtime and incorporates them into the executing application. This is the pathway most commonly understood as CSR.

The second pathway is indirect: the runtime loads content (data, media, configuration, templates, prompts) that code already present in the runtime then interprets. The interpretation generates, compiles, or otherwise produces further executable behavior, which the runtime then runs. JSON evaluated as code, configuration that declares scripts to load, templates that carry executable expressions, data-driven dispatch tables, and model outputs that drive control flow are all instances of the second pathway. Both pathways produce runtime code execution. The second is the one that defeats the usual line between content and code: inside a CSR runtime, content the application interprets can become code the application runs. Data as Code and Code That Writes Code are the two ways an attacker turns that second pathway into an attack.

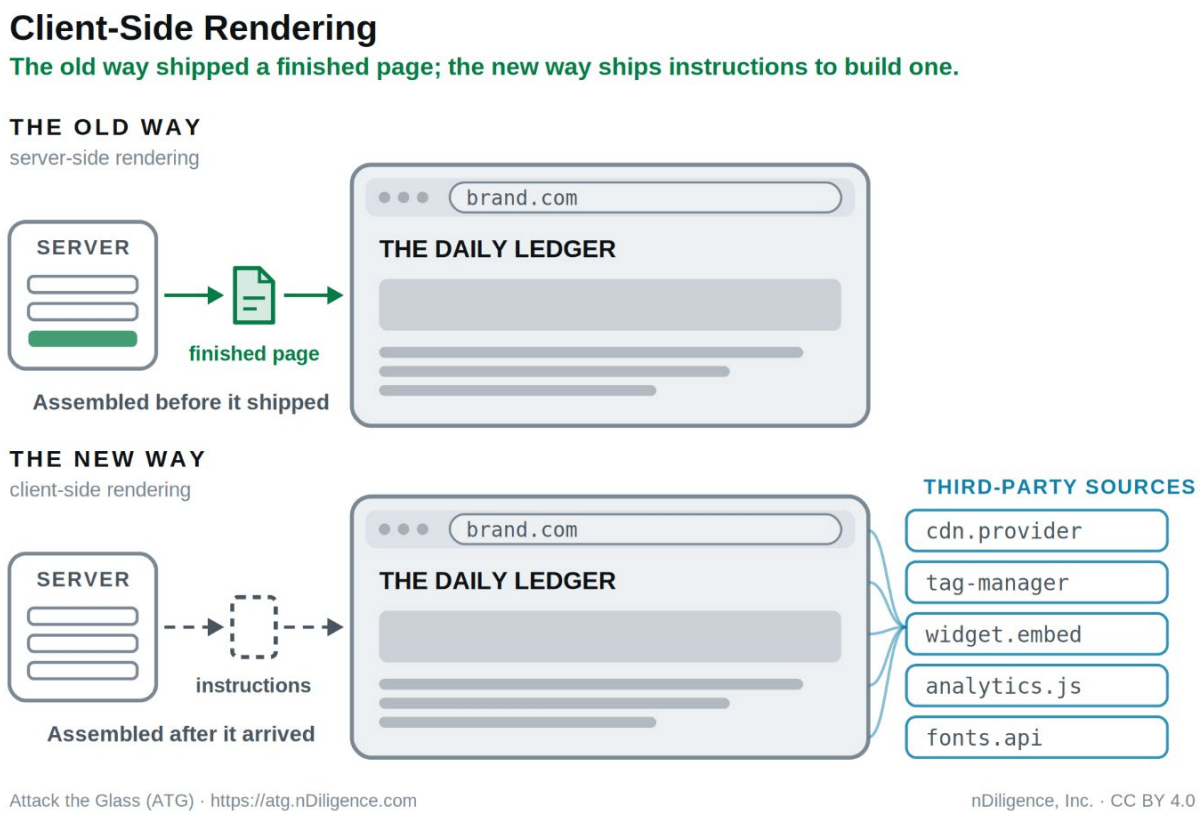

***Figure 1**: Server-side rendering delivers a finished page. Client-side rendering delivers the parts and the instructions to assemble them.*

1.3: The Move from Server to Client

The web migrated first. Around 2010, single-page-application frameworks reached mainstream adoption: Backbone, AngularJS, then React, Vue, Svelte, and the modern framework ecosystem. jQuery had set the stage through the second half of the 2000s as the dominant DOM-manipulation library but was not itself an SPA framework. The same migration carried into mobile through cross-platform frameworks (React Native, Flutter, Xamarin) and hybrid WebView frameworks (Cordova and Capacitor), into the desktop through Electron and Tauri, into immersive computing through AR and VR runtimes, into appliances through smart-TV and signage runtimes, and into productivity through notebook front-ends and extension ecosystems.

Hardware carried the new load. RAM and CPU growth on consumer devices made it practical to render rich, animated interfaces in the runtime without sacrificing user experience. JavaScript engine performance (V8), the rise of Node.js and npm, and the broader maturation of the front-end toolchain compounded the effect. Each generation of capability (richer animations, real-time visualizations, more complex interactivity) drove broader adoption of CSR frameworks. The same pattern pulled more languages into the ecosystem to deliver the experience features the runtime had to provide.

1.4: Why CSR Won

Richer user interfaces and more adaptive user experiences are the obvious technical advantages. CSR provided more dynamic interaction, perceived performance and responsiveness, and the app-like behavior the web had previously lacked. It also simplified operations: deploying static assets to a CDN and letting the client assemble the application removed much of the backend complexity from each release. And it gave development teams single-page-application frameworks, which provided an architecture, tooling, and patterns for building modern web applications. Each of these was a real, decisive improvement. But the transition to CSR was not just a technical advantage; it was also a financial one.

CSR's financial advantage was structural: it moved the cost of rendering and computation off vendor servers and onto the consumer device. The shift drove parallel demand for high-speed home internet, mobile bandwidth, expanded device RAM, and expanded local storage. Every layer of the technology supply chain benefited: application owners operated leaner backends, network operators sold more bandwidth, hardware vendors sold more capable components, and platform companies sold more advanced finished devices at higher margins. The same shift built the high-end consumer hardware market (the flagship smartphone is the most visible example) and the platform-company valuations that depend on it.

Consumers paid for all of it. The cost was spread across billions of users and billions of devices, so it was rarely noticed as a transfer. The *Compute Tax* is that transfer: a tax rather than a bill, an extraction with no line item and no discount, where the user pays more in device, bandwidth, and power and receives the same product at the same price. The provider ships a blueprint rather than a finished product, and the user's device pays to build it. Split across billions, no one sees the cost, or the fortune it hides. That one decision is both what saves the provider money and what leaves the runtime open to attack, which is why the design is unlikely to be given up. CSR won because the cost could be sent to someone who would not see it itemized.

1.5: The Application Blueprint

What a CSR application ships is a blueprint, not a finished product: a plan the device realizes just in time at render. The base application is a thin shell that fetches, assembles, and executes the rest of the application from remote sources at runtime (a bootloader, in engineering terms). The running application is composed from dozens or hundreds of independent sources: scripts, modules, fonts, configuration, feature flags, models, ad units, telemetry. The Just-in-Time App Blueprint is that pattern: the device builds the application at the moment of render, from parts fetched live from sources the owner does

not control. A blueprint is supposed to guarantee the same build every time. This one guarantees the opposite: the application as it executes is different from the application as it was built.

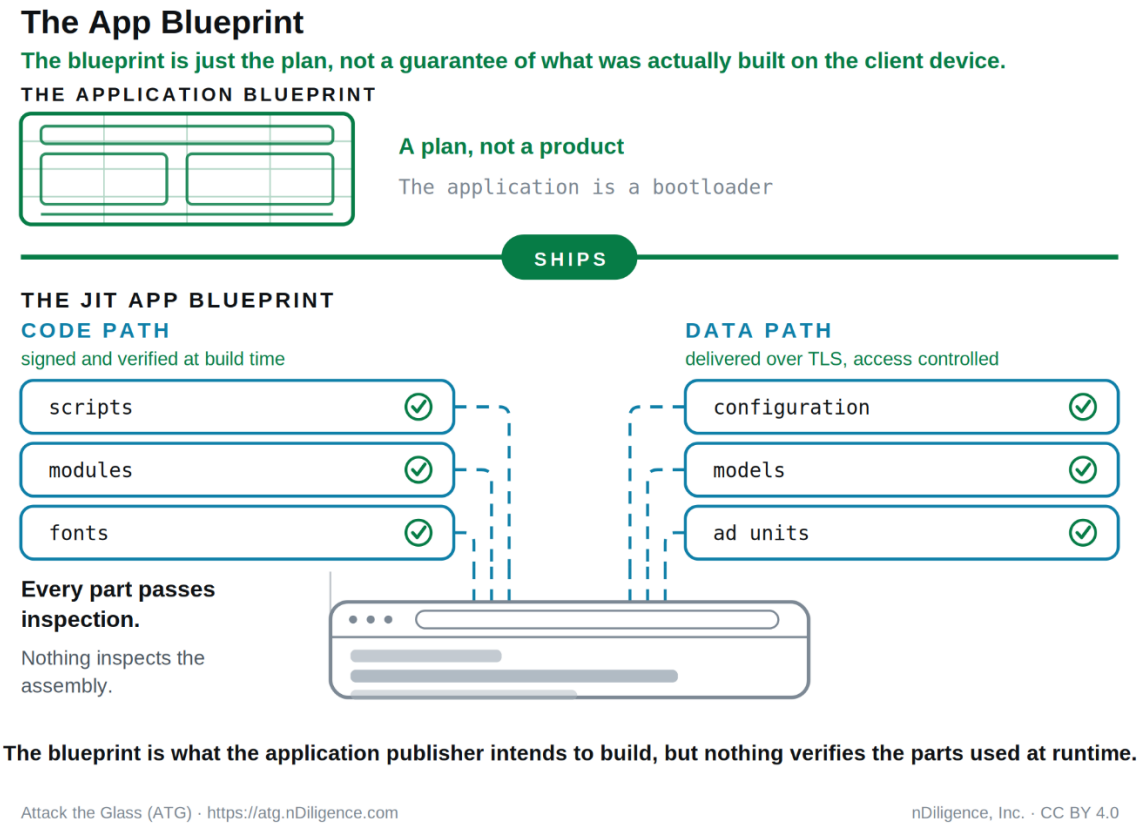

***Figure 2**: What ships is a plan, not a product. The build happens on the reader's device, every time.*

The blueprint is also a trust list, and that part has no counterpart in shipped software. Every remote source the plan names is a party the device will trust at render. The author of that plan decides who those parties are, and the device carries the decision out. The person operating the device is not shown the list, cannot inspect what any entry on it returns, and cannot decline one without losing the application. *Conscripted Trust* names the case where one party grants the trust and a different party carries its consequences.

The client is not powerless, and the distinction matters. Extensions, script blockers, enterprise policy, and network-level filtering can each refuse a source outright, and refusing one that the application depends on usually breaks the application. What none of them can do is accept a source on conditions, or establish that a permitted source returned what it was expected to return. The client can refuse the relationship. It cannot condition it.

Two boundaries keep the claim accurate. Installed software also carries dependency decisions its user did not make, but that dependency set is fixed at install and does not change between launches, so it can in principle be examined; a blueprint's trust set is resolved per render from addresses that may return anything, so there is no build of the application whose trust set could be inspected at all. And server-side rendering delegates in the same way, but composes inside the operator's own boundary, where the operator can observe the result and bears its consequences directly. Delegation itself is not new. What is new is that the delegated relationship now resolves in a place neither the operator nor the user can observe.

Commercial logic reinforced the blueprint pattern. Keeping proprietary code, algorithms, and behavior in the runtime layer (rather than in shipped binaries) raises the cost of theft, reverse engineering, and behavioral monitoring. The vendor gains IP protection on top of the deployment-flexibility benefit. The same pattern enabled the rise of analytic and telemetry frameworks that capture user interactions in real time. Those frameworks became the foundation for engagement-driven design (infinite scroll and its descendants) and the broader behavioral-monitoring ecosystem built on top of CSR runtimes.

1.6: The Perpetual State Machine

Assembly at render implies a moment when assembly finishes: the parts arrive, the application composes, and from then on one built thing is running. Sufficiently interactive applications do not settle that way. The composed application continues to change for as long as the session lasts, and it changes in two ways that have nothing to do with each other.

A user drives the first kind of change. State and workflow are a combinatorial function of the sequence of interactions, the date and time, the platform, the location, the device, and whatever preferences and entitlements the user carries. Two people using the same application on the same day arrive at different rendered states by different paths, and neither path was enumerable in advance. This much is familiar. It is the ordinary behavior of any stateful interface and it is not, on its own, a security property.

The parts drive the second kind, and this one belongs to CSR specifically. The set of parts the runtime assembles is not fixed between renders. What varies from one render to the next is therefore not only the application's state but its function: the same plan, realized from a different collection of parts, is a different program. The owner ships one plan. The device can build it differently on every pass.

The Perpetual State Machine is the application that results. A finite state machine is bounded, enumerable, and testable: its states can be listed and a test can walk them.

Combinatorial input alone does not break that, and it never did; server-rendered applications and native binaries carry long input histories too, and the methods for reasoning about them are mature. What breaks it is the second kind of change. Where the parts are not fixed between renders, the transition function is not fixed either, so there is no stable machine to enumerate and no fixed target for a test to certify. There is not one consistent application. Each render on each machine is a unique instance, potentially never to recur in that exact form again.

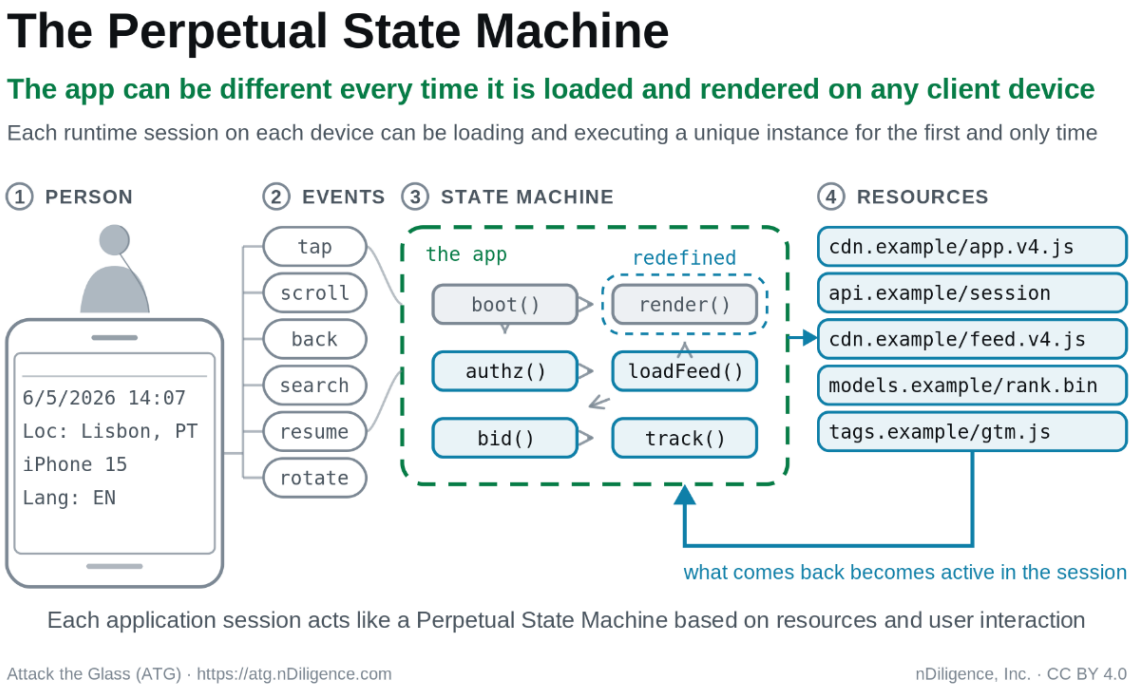

***Figure 3**: Each render is a distinct instance. There is no single assembled application to inspect.*

The claim needs a boundary. A static page that renders once and does nothing further is not a Perpetual State Machine, and neither is a form-and-submit application whose parts are pinned at build time. The description holds where an application is sufficiently interactive and assembles its parts live, which is the prevalent shape of modern client applications rather than a property of all of them. Where it holds, the consequence reaches past testing. A defender comparing a suspect render against a known-good one has no known-good one to reach for, because no two renders were ever required to match.

1.7: Code and Data on Different Paths

CSR architectures separate the delivery of code from the delivery of data. Application logic, frameworks, and scripts travel one path. Content, configuration, and responses travel another. The two are sourced from different providers, arrive at the runtime under different trust assumptions, and meet only inside the runtime where the application finally

executes. They were never written or reviewed together as one piece of software, but both are required for the application to run.

Upstream defenses can be strong. Code is signed and verified at build time; CDN delivery is authenticated; data endpoints use TLS and access control. The final assembly happens inside an untrusted container, where none of those defenses operate.

Every part passes inspection. Nothing inspects the assembly.

1.8: Verification Ends at Delivery

All meaningful verification happens before delivery. Code review, signing, scanning, hashing, dependency analysis, and supply-chain attestation (the signed record of how a component was built and by whom) all operate on files at rest, before delivery to the Client Runtime. Once execution begins inside the Client Runtime, no further verification occurs. The trust granted at delivery is the only trust the runtime ever receives.

Mobile WebViews and JavaScript bridges inherit the gap. So do vehicle infotainment runtimes running embedded browser engines. So do the operator screens of industrial control systems when those screens are built with CSR frameworks, and so do smart-TV applications and productivity-tool extension hosts. The structural property follows from the architecture, not the platform: any runtime that grants trust at delivery and does not continuously re-verify thereafter has the same exposure.

NIST SP 800-207 is guidance, not a standard, and it sets out tenets for Zero Trust Architecture rather than a conformance checklist. The Client Runtime works against those tenets structurally:

- **Continuous verification:** absent after the initial load. Trust is granted once at session start, and the runtime offers no mechanism to re-verify sources, code, or data thereafter.
- **Least privilege at runtime:** loaded resources operate at the host application's permission level. No scoping mechanism restricts a script or data-driven code path to less authority than the page or app it ran inside.
- **Explicit denial of implicit trust extension:** loaded resources implicitly inherit the trust of the runtime that loaded them. The act of loading is the act of trusting.
- **Continuous re-evaluation under changing context:** the runtime makes the trust decision once. Subsequent context changes (new user, different network, elevated privilege, anomalous behavior) do not trigger re-evaluation.

Every Client Runtime inherits the same arrangement: verification stops at delivery, and the trust granted at load is the only trust the runtime ever applies. That trust is usually not the

product of verification at all. In many client-side stacks a library, service, or technology is included because it is popular, well-branded, and widely adopted, not because anyone verified its behavior or intent. *Herd Trust* is inclusion by reputation: the decision to include rests on ubiquity, and no verification of behavior or intent follows. Widespread use feels like herd immunity, but in reality is Herd Trust: adoption does not provide any true safety and the sense of trust is from being a part of a larger group. Trusted by the herd and never verified, the resource is then granted the runtime's full rights at execution. The runtime runs on faith.

Herd Trust explains why an application author selects a source. Conscripted Trust explains why that selection binds someone else. The two compose: a resource is included because the crowd already uses it, and the person in front of the Glass inherits that reasoning without being consulted, without visibility into what the resource returns, and without any way to accept it on narrower terms.

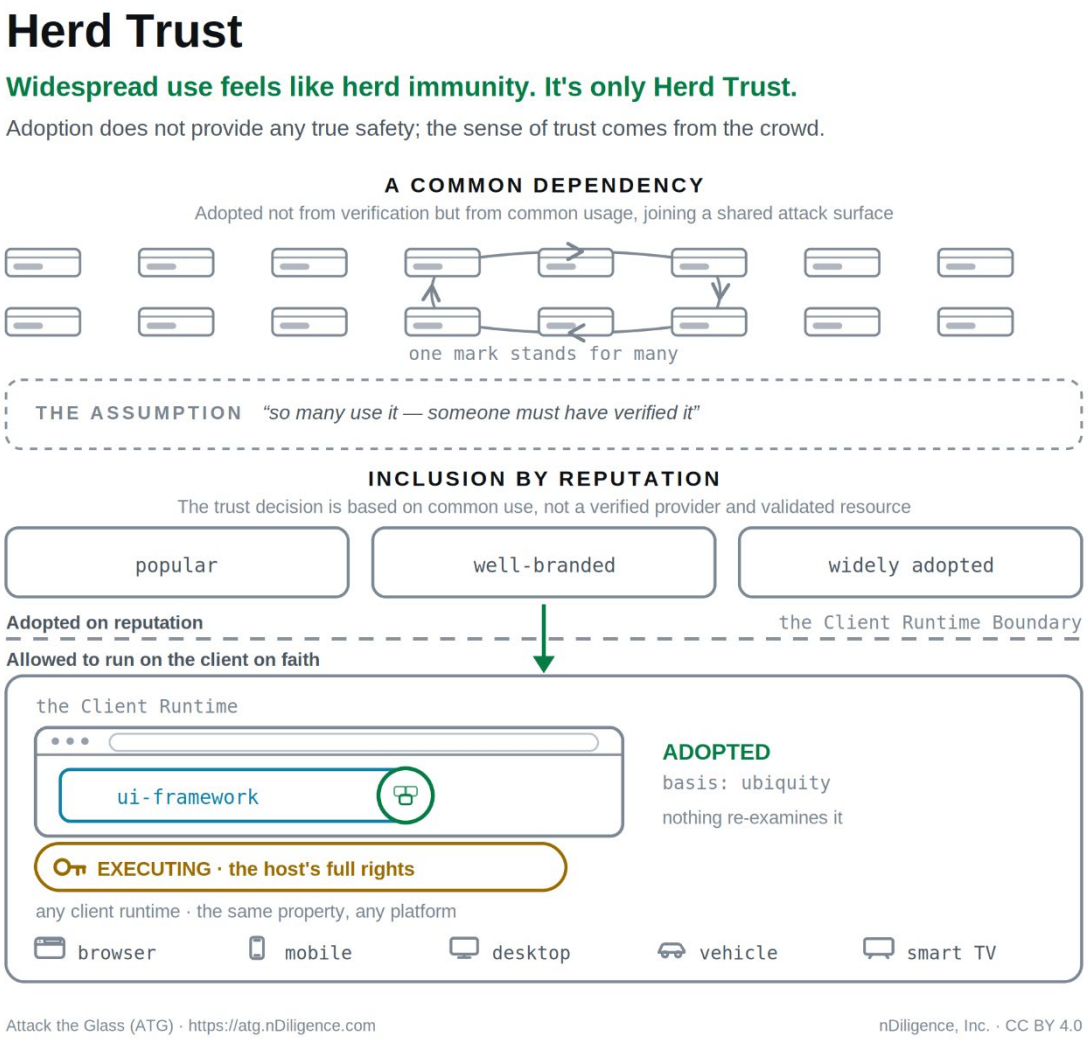

***Figure 4**: Inclusion rests on how widely a component is used, not on anything that verified it.*

1.9: The Allowlist Fallacy

CSR runtimes do not require remote sources to be trusted. A runtime could pin every dependency, refuse to load anything not declared at build time, and reject any data that contained executable instructions. Almost none do. The prevailing implementation pattern is broad runtime trust: scripts can fetch and execute further scripts; fonts and assets are loaded from arbitrary CDNs; configuration endpoints contacted at startup are trusted to influence behavior. Strict-pinning runtimes exist in narrow cases (some embedded contexts, some hardened browser modes); they are exceptions.

The Allowlist Fallacy is the mistaken belief this pattern rests on: that adding a source to an allowlist (a domain, CDN, package, or endpoint approved as trusted) also approves everything that source will later return. An allowlist authorizes an address, not the content it ships, and that address can, by design, return different or mutated content on any later load. An allowlist approves the sender, not the package. The pattern is an implementation choice, not a structural property of CSR, and it is so widespread that the consequences are effectively general. The Manifest Illusion, Data as Code, and Code That Writes Code all depend on it.

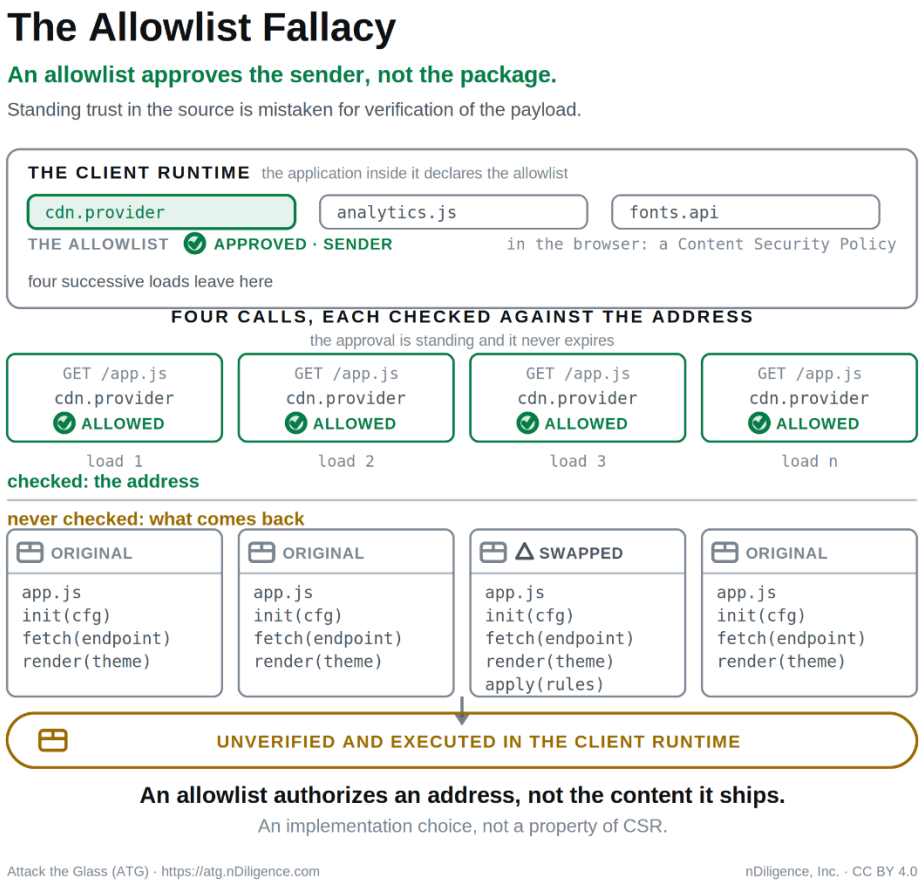

***Figure 5**: An allowlist approves the sender. It does not approve the package.*

1.10: The Manifest Illusion

Runtime loading is transitive. Once execution begins, loaded resources pull in further resources with no obligation to surface what they reach for. A script inserts another script via dynamic <script> injection. A service worker fetches and caches arbitrary URLs. A tag manager, the marketing tool that injects analytics and advertising scripts into a page, loads configuration that itself loads further scripts. A font CSS @import chain recursively pulls in additional fonts and assets. Each operates on the same principle: loading is an action the runtime takes, and the runtime takes it without recording the trust relationship anywhere a defender can inspect.

The mechanism is not browser-specific. A mobile WebView reaches the same way through a JavaScript bridge. A hybrid desktop shell such as Electron does the same wherever its web layer is wired through to Node.js. A vehicle infotainment runtime loads configuration delivered over the air, pulled straight from a server after the vehicle is in service and without a new firmware image, and that configuration fetches further modules. An IoT dashboard loads its UI extensions from a remote registry.

The Manifest Illusion is the structural fact that no complete and accurate list of what actually runs can exist where an application loads code dynamically, generates code at runtime, or fetches parts live. The manifest defenders hold (a build-time package list, a lockfile, an SBOM, or the set of permitted origins in a runtime-enforced CSP allowlist) records what was declared, not the code that actually executes, and the two drift apart with nothing to reconcile them.

Examining the root sources of an application does not reveal the eventual runtime trust graph; the trust chain forms at runtime, and no file a reviewer can open beforehand shows it. You have a manifest from the plans, not a true manifest of what was built and runs.

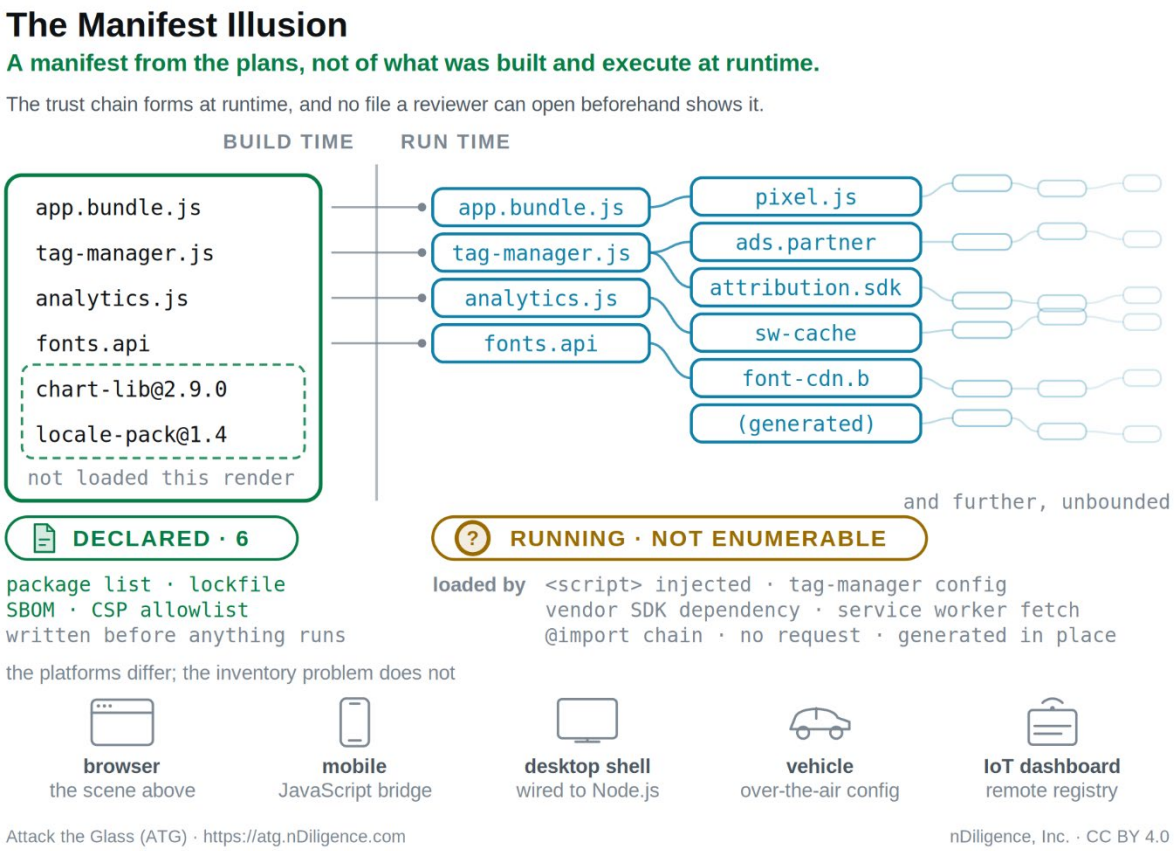

***Figure 6**: The manifest records what was declared. Nothing records what executed, and the two drift apart.*

1.11: Data as Code

CSR runtimes blur the data/code boundary. Configuration declares scripts to load. Templates carry expressions that execute. JSON is evaluated as code. JSONP returns code wrapped as a data response. Dynamic import() consumes module specifiers fed from configuration. Feature-flag services ship code paths as feature toggles. Model outputs drive control flow wherever a model is allowed to act on its own. A response that looks like data is, in many runtimes, a vector for source injection.

The data path becomes a code path without crossing any visible boundary. *That crossing is Data as Code*. The same property holds across the Client Runtime landscape: a vehicle infotainment system's remote configuration channel, a mobile application's feature-flag service, an IoT dashboard's content endpoint, and a productivity tool's plugin manifest each carry the same permeability. The Two-Stage Attack is the operational form of the same permeability, used deliberately by an attacker.

1.12: Code That Writes Code

CSR runtimes routinely allow code to generate code at runtime, compile or interpret it on the fly, execute it, and discard it. eval and the Function constructor produce executable code from a string. new Function(…) builds a function body at runtime from arbitrary input. Dynamic import() resolves module specifiers computed at runtime. WebAssembly compiles from buffers fetched over the network. Runtime template engines turn data into executable functions. Embedded Just-in-Time compilers execute domain-specific code supplied at runtime.

The generated code never appears in any source file, any build output, or any stored asset. It exists only in the moment of execution and leaves no audit surface. A forensic investigator inspecting the running runtime might see it; one inspecting the source code or the stored files will not. *Code That Writes Code* operates the same way in mobile runtimes, desktop shells, embedded JavaScript engines, AR and VR scene-graph engines, and productivity-tool macro engines. Malice Without Malware rests in part on this: the attack leaves nothing behind for an audit to find.

1.13: The Glass Beyond the Browser

CSR runs wherever there is a screen and a CSR-capable Client Runtime. The browser is the largest instance, not the only one. Mobile WebViews and hybrid app shells, Electron and Tauri on the desktop, smart-TV and signage runtimes, vehicle infotainment, AR and VR scene engines, IoT and OT dashboards, notebook front-ends, productivity-tool extension hosts, and plugin marketplaces are each Client Runtimes in their own right.

This breadth is *the Glass Beyond the Browser*: the glass is any surface where a runtime renders for a human. The structural properties and exploitation mechanisms of CSR operate the same way in every one of these runtimes. The consequences differ by platform; the architecture does not. A defensive vocabulary built for a single environment cannot serve all of them.

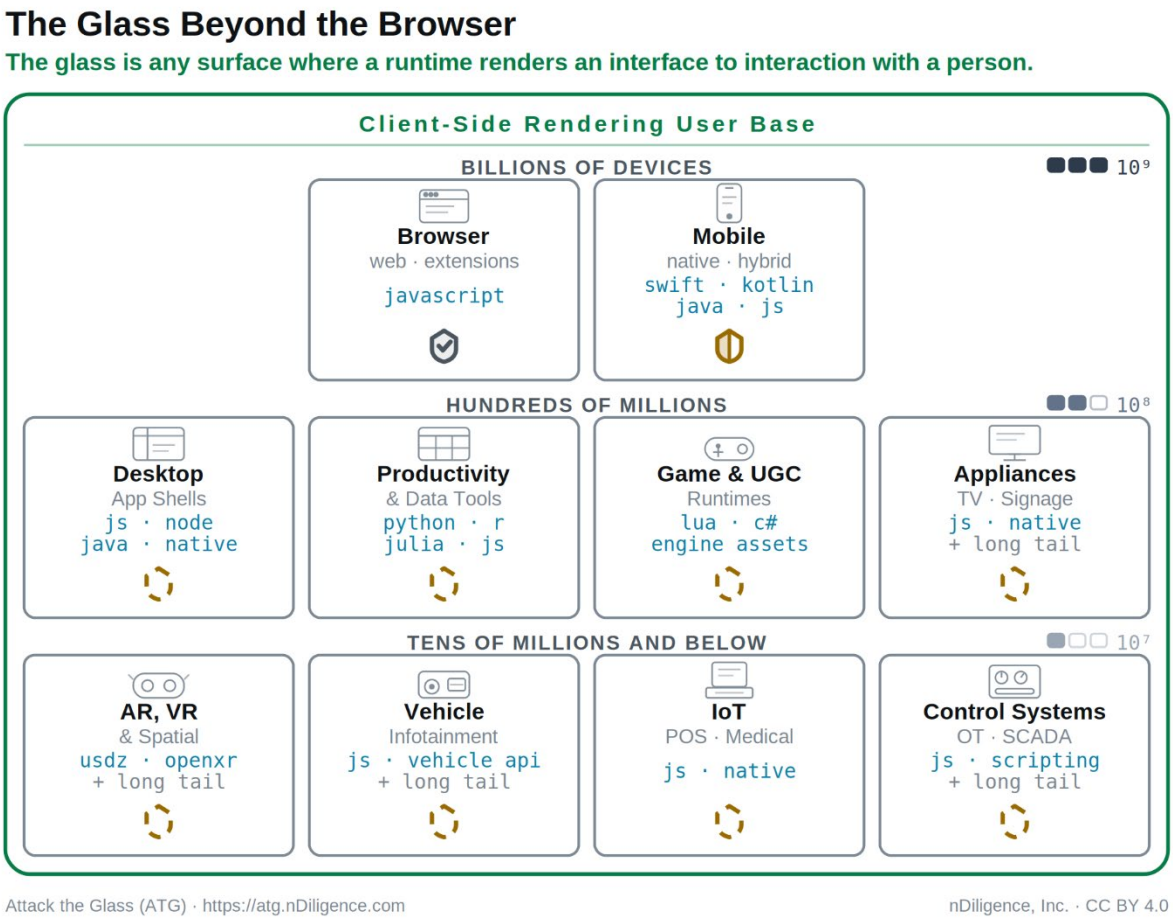

***Figure 7**: Every runtime that renders content is a rendering surface. The browser is one case, not the category.*

1.14: Malice Without Malware

The malware-centric framing has shaped the security industry for three decades, and it has earned its place. Static binary defenses remain a real and necessary line. The categories most relevant to CSR sit outside that framing: manipulation, deception, behavioral steering, supply-chain injection that leaves no altered binary, and runtime trust abuse. None of these is well captured by the term "malware," and the assumption that makes anti-virus work, that there is a fixed file sitting on disk to scan, does not survive a runtime where the loader is opaque, data is code, and code generates code.

A defensive vocabulary that lags the actual threat is itself part of the problem. Anti-virus, signature scanning, and hash-based detection do not detect runtime manipulation of the rendered application, runtime injection of third-party trust, or content-driven generation of executable behavior. What is required is a defensive posture purpose-built for client-side surfaces: client-side attack surface management, runtime behavioral monitoring at the

rendering surface, and a new generation of anti-manipulation and anti-deception capability.

Malice Without Malware names that gap. Malicious code, from a malicious source, is what defenses hunt. ATG is neither. It can produce a malicious outcome with clean, legitimate code from a trusted source, so the tools built to catch bad code from a bad source never fire. The malice is real; the malware is absent. CSR is the foundation of the modern internet, the smartphone economy, the desktop application ecosystem, the AR and VR runtime, the in-vehicle infotainment system, and a growing share of industrial control interfaces. There is no patch for CSR. There is no "out." CSR is the environment in which the modern technology stack lives, and a defensive vocabulary built for a different environment cannot remain the operational center of the security response.

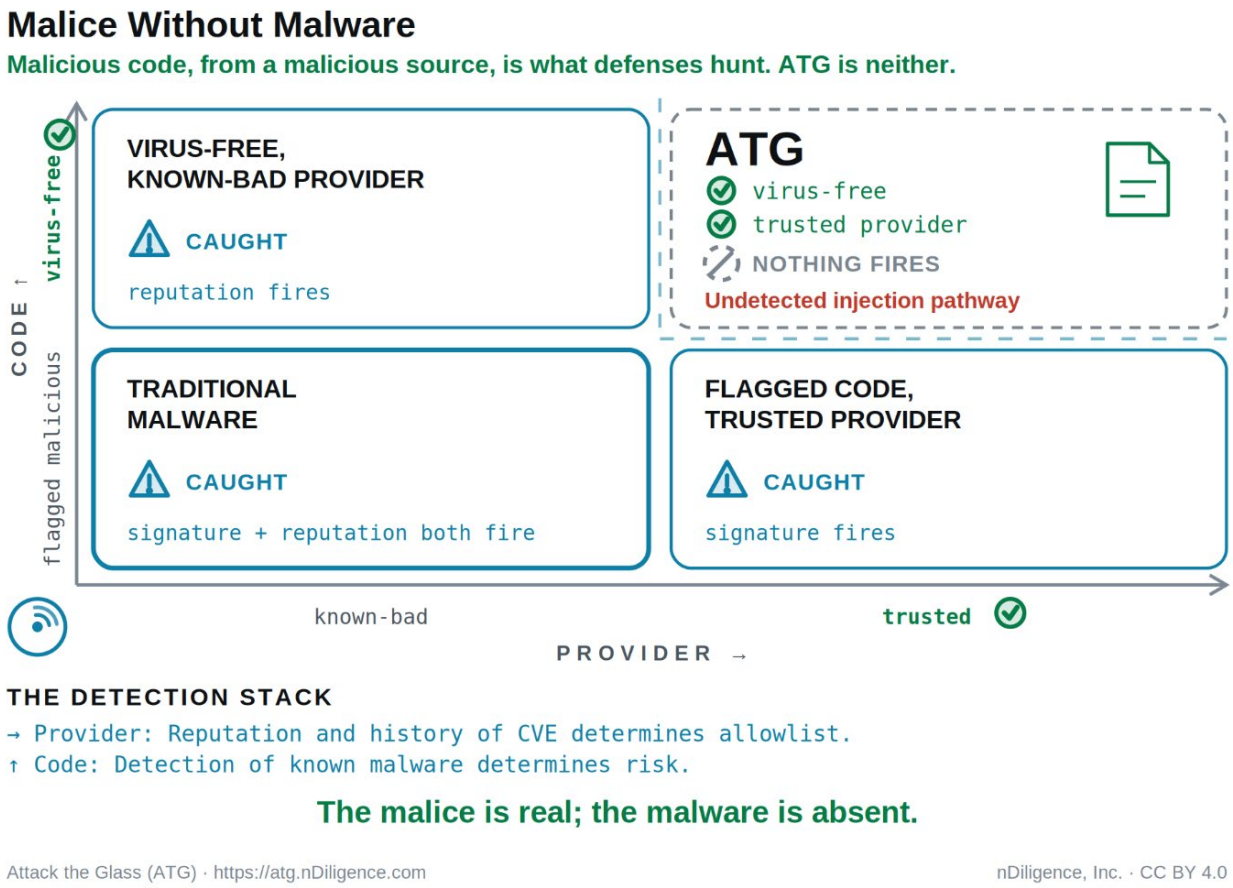

***Figure 8**: No exploit, no malware, no perimeter crossed. The position was acquired and then used.*

1.15: CSR and Client Runtime AI

Client Runtime AI is moving from research curiosity to default capability, and its arrival on consumer devices compounds every property of CSR. The model files loaded into the runtime are fetched at runtime like any other CSR-loaded resource. LoRA adapters (small add-on files that change how a model behaves), prompts, system instructions, and tool definitions delivered to the runtime all become second-stage parts that can be combined subject to the same Manifest Illusion and Data as Code mechanics.

Where a model is allowed to act on its own inside the Client Runtime, it can take actions on the user's behalf with permissions far beyond what a traditional script can claim. Model-driven personalization makes the rendered application functionally unique per user, which makes the manipulation surface harder to detect, audit, or compare across populations.

Client Runtime AI does not introduce a new vulnerability; it amplifies the one already present. CSR is the architecture, and ATG is the vulnerability CSR produces.

Chapter 2: The ATG Vulnerability Architecture

Client-Side Rendering produces a runtime environment that trusts what it loads and does not re-verify. The runtime trusts by default because of the way the architecture works, not because an operator chose a setting. Structural trust by itself is exposure, not vulnerability: the runtime is not vulnerable on its own, and becomes vulnerable when the supply chain feeding it can be shaped by an adversary.

The digital supply chain is unlike any chain humanity has built. Code, content, and configuration flow into Client Runtimes from millions of producers, through a small number of consolidated delivery providers, to billions of devices every day. No single party owns it. No inspection regime governs it. The protection rigor every other consequential supply chain carries (provenance, chain of custody, recall capacity) is largely absent. The runtime accepts what the chain delivers, and the chain has no obligation to deliver what was reviewed.

Attack the Glass (ATG) is a type of supply chain injection vulnerability that targets client-side rendering. It arises at that meeting point: a runtime that trusts what it loads, fed by a supply chain no one has inspected. What defines it is the architecture rather than any one exploit. If an adversary can swap what the runtime receives at the moment of delivery, and the swapped content then runs inside the assembled application, that is ATG.

Information manipulation is the dominant form of ATG attack. Functional disruption (making an application fail, silencing an alert, subverting a feature) takes more technical skill to carry out, and it happens less often, succeeds less often, and does less damage when it does.

2.1: The Internet Is a Supply Chain

The internet is not a communication network. It is a layered supply chain through which code, content, configuration, and behavior flow from many producers and intermediaries to the rendering surface on the user's device. Communication is one flow within that chain, not the chain itself.

Four dimensions set it apart from any prior supply chain humanity has built: scale, decentralization, rate of evolution, and embeddedness.

- **Scale:** billions of users, billions of devices, millions of distinct producers and intermediaries, and a continuous flow of code and content that no other chain matches in volume. The grocery system, the global pharmaceutical chain, the semiconductor industry, the defense materiel pipeline: each is large in its own terms, and each is a fraction of the volume the internet supply chain moves every hour.

- **Decentralization:** no single owner, no single regulator, no single governance authority, no single inspection regime. The internet is assembled across nearly every country and culture and operates simultaneously interconnected and autonomous: nodes cooperate at the protocol level while answering to different jurisdictions, different operators, and different commercial incentives. No prior supply chain of comparable consequence has lacked a central ownership or governance layer.
- **Evolution rate:** the internet at the start of any decade is materially different from the internet at the end. Protocols are added; whole categories of intermediaries (CDNs, model hosts, identity providers, tag managers that load third-party scripts into a page, AI inference services) emerge in years rather than generations; entire delivery pathways come into existence, replace prior pathways, and are themselves replaced. The chain rewrites itself faster than the institutions designed to oversee any one of its segments can adapt.
- **Embeddedness:** the internet now sits inside nearly every aspect of government, commercial, and civil-society activity, deeply integrated into the personal lives of billions of people. Banking, healthcare, transportation, energy, and the daily attention of populations all route through it. Almost no consequential decision in modern life is reachable without traversing some part of the chain.

The internet is the most consequential and transformative invention in human history. It was built through the largest cooperative international effort in history. Client-Side Rendering is the architecture that extends that chain all the way into the runtime on the user's device.

2.2: The Most Important Supply Chain Is the Least Protected

Physical supply chains carry rigor commensurate with their consequence. A pallet of spinach crossing into a major market arrives with documented producer identification, lot codes, harvest dates, cold-chain telemetry, and recall capability that lets a contamination event be traced to a specific field on a specific day. A shipment of pharmaceutical product carries serialized packaging, transaction histories that document every change of custody from manufacturer through distributor to dispenser, and regulatory mechanisms that require irregularities to be reported and acted on. A semiconductor entering a defense application arrives through a trusted-foundry pathway with provenance documentation, anti-counterfeit attestation (a signed record of how the part was built and by whom), and chain-of-custody verification at each handoff. The protection rigor in each case is proportional to what would happen if the chain were compromised.

By contrast, the digital supply chain that delivers code, content, and configuration to billions of Client Runtimes every day carries nothing comparable. There is no provenance requirement on a script loaded from an arbitrary URL. There is no chain of custody on a configuration payload fetched at runtime. There is no inspection regime on a model file downloaded into a runtime. There is no recall mechanism on a third-party script deployed across thousands of sites. The asymmetry is structural and has been well-known for more than a decade. It remains uncorrected.

ATG operates inside that asymmetry. An adversary in the digital supply chain faces none of the documentation, inspection, or attribution requirements an adversary in the physical chain would face. Economics, detection difficulty, and recoverability all favor the adversary. The same is not true in the supply chains the world has decided to protect.

Behind the protection asymmetry sits a deeper one: almost no one knows how this supply chain actually works, who operates it at the relevant layers, or who controls it. Some segments are increasingly being protected: package registries with provenance attestation, software bills of materials, build-system integrity programs, and signed binaries are real progress at the producer end of the chain. Vast sections remain opaque, unguarded, and structurally vulnerable to anyone willing to learn where the gaps are. Effective defense requires an inventory of what is being defended. Vast sections of this chain offer none.

The most important supply chain humanity has ever built is the least protected supply chain humanity has ever built.

2.3: The Digital Supply Chain

The Digital Supply Chain is the system of producers, intermediaries, transformation points, distribution channels, and delivery mechanisms through which executable code, content, and configuration move from their source to the rendering surface inside a Client Runtime. The same general shape governs every Client Runtime on every host platform.

The chain does not carry only files. It carries scripts and modules that execute as code. It carries content (HTML, CSS, JSON, media, ads) the runtime interprets. It carries configuration (feature flags, A/B-test assignments, tag-manager directives, import maps, manifests) that changes behavior or directs further fetches. It carries the AI substrate (model weights, LoRA adapters, prompts, system instructions, tool definitions) that drives any model integrated into the runtime. It carries telemetry specifications that determine what is collected and where it is sent. The distinctions between code, content, and configuration are useful for description. Inside a Client Runtime, all three become executable behavior.

The chain runs through five layers from origin to rendering surface: producers, intermediaries, transformation points, distribution channels, and delivery mechanisms.

- **Producers write the code or create the content.** Application developers, library maintainers, content creators, advertising operations, configuration owners, model trainers, and AI service operators all sit at this layer. The producer layer is where a piece of code or content first gets the name it claims and the behavior it is supposed to have.
- **Intermediaries host and aggregate**. Open-source package registries, source-control hosts, model repositories, ad networks, and configuration-management services all carry code and content forward without creating it. The intermediary layer is the first place a file leaves the producer's control and lands on infrastructure that other parties share.
- **Transformation points change the code between the intermediary stage and the distribution stage.** Build systems, bundlers, transpilers, minifiers, optimizers, CI/CD pipelines, and integration-time composition tools each take one piece of code in and emit a different one. The transformation layer is where what the producer published and what the device eventually runs stop being the same thing.
- **Distribution channels push the transformed code toward the rendering surface.** Content delivery networks, package mirrors, app stores, over-the-air bundle delivery services (which push code updates straight from a server to an already-installed application, without going back through the app store), and tag managers each operate at this layer. The distribution layer is where the largest aggregations of dependent applications meet a single point of delivery.
- **Delivery mechanisms bring the code or content across the trust boundary into the runtime.** HTTP requests for scripts, fonts, configuration, and content; WebSocket and server-sent-event streams; push notifications; over-the-air bundle pulls; and responses from an inference API, the address an application calls to get an answer from a model, are all delivery mechanisms. The delivery layer is where a piece of content enters the running Client Runtime.

The chain ends at the rendering surface. The Client Runtime assembles the application from whatever the delivery layer brings, integrates it with everything already loaded, and executes the result.

Trust at each handoff is structural and largely implicit. Producers trust their own pipelines. Intermediaries trust producers' uploads, with verification rigor that varies from registry to registry. Transformation points trust their inputs, and the outputs they emit are rarely cryptographically pinned to the producer-declared identity. Distribution channels trust the

transformation pipeline that fed them. Delivery mechanisms authenticate the endpoint, not the content. The rendering surface trusts whatever arrives. Once producer-layer trust is established, every subsequent layer extends it without re-verification. The result at the rendering surface is that every check happened before the code ran, and nothing checks it again once it does.

2.4: The Glass: The Final Mile of Every Digital System

Modern life takes place in front of glass. A person wakes to a phone screen, works at a laptop, takes a meeting through a webcam, glances at a vehicle dashboard, watches the evening on a television, and checks a refrigerator's screen for a grocery list. The number of distinct panes most people interact with in a day runs in the dozens; the time spent looking at them runs to most of the day.

Each pane is an aperture into a system the unaided senses cannot reach. A child sees a relative two thousand miles away. A surgeon sees inside their patient. An engineer sees the floor of a building a continent away. Each pane lets a person see around the world or deep into systems at scales the unaided eye cannot see.

The Glass is not a figure of speech here. It is the last component in the delivery chain, the surface on which the finished application is drawn. Every system that produces value for a person finally produces it on a piece of glass. Public billboards, signage in transit hubs, industrial control HMIs, medical-equipment screens, AR headsets, and point-of-sale terminals all carry the same essential property as the consumer panes: a piece of glass with technology behind it. The browser is the most studied case, not the boundary.

The Glass is that final point of contact. The Glass means the rendering surface wherever it appears, not any one device and not any one physical pane. The Glass is the final mile of every digital delivery chain. Whatever provenance, signing, scanning, or verification existed upstream, the system arrives at the user only after passing through the Glass. The Glass is where the assembled application meets the user, where the value of the system materializes, and where the vulnerability lies.

The Glass is also where a person acts. It is the surface on which they see, interact with, and act on the information and features the software puts in front of them, and the acting matters as much as the seeing. A screen that only displayed would expose a reader. A screen that is also the control surface exposes every decision taken through it, which is why the exposure runs from a news page through an administrative console to an operator's panel without changing its shape.

2.5: Integration on the Glass (IOTG)

The integration moment in modern software does not happen in the build pipeline, at the CDN, at the package registry, the bundler, the packaged build output, or any layer where software-supply-chain controls are typically applied. It happens at the Glass, on the user's device, at runtime, after every upstream control has already finished its work.

CSR is the technique that makes this true. The rendered application is composed at runtime from many independent sources: scripts and modules from package registries, content from CDNs, configuration from feature-flag services, ads from ad networks, and model weights or inference responses from AI endpoints. None of these sources sees or coordinates with the others. Each is integrated only when the runtime assembles them inside the Client Runtime.

Integration on the Glass (IOTG) is that moment. IOTG is what CSR produces in operational terms when it meets the Glass: the point at which all the upstream supply-chain pieces become a single executing system. The integration happens on the user's device, inside an untrusted container, with no participant in the chain having authority over the others.

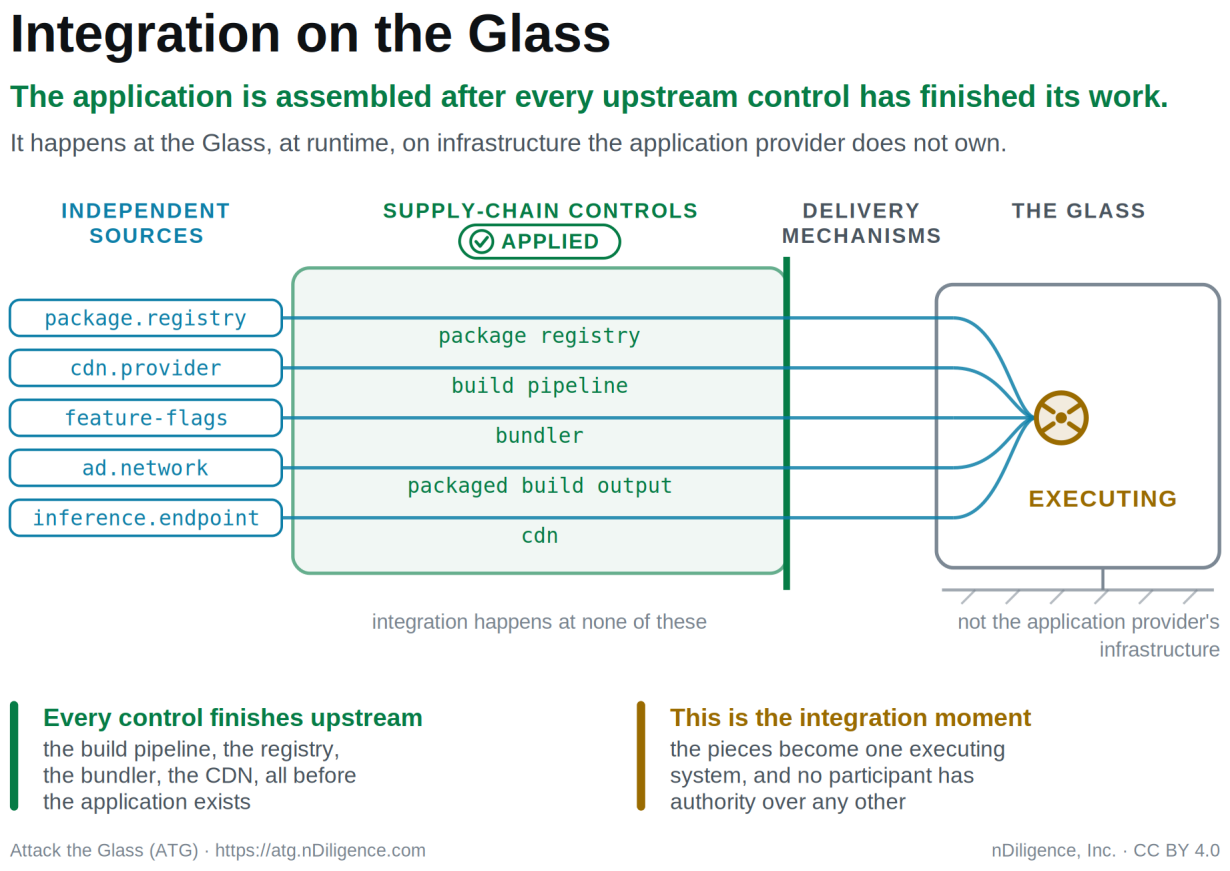

***Figure 9**: The glass is where every supply chain terminates, and where all of them are integrated at once.*

The shift is structural. Server-rendered systems integrated on infrastructure the operator owned. CSR systems integrate on infrastructure the operator does not own and cannot inspect. Every defense that assumed integration happened inside the operator's perimeter is operating against an assumption that no longer holds. IOTG is what replaced it.

2.6: URL Overtrust

A URL is a place. Most cybersecurity work treats URLs as strings of text the runtime uses to fetch resources, and that operational definition is correct as far as it goes. It is also incomplete. A URL is a place within the digital supply chain, and like any place in the physical world, it carries attributes that determine what a relationship with it actually means.

Five attributes determine what a URL means as a place: ownership, jurisdiction, access control, allowed activity, and risk relationship.

- **Ownership:** who controls the space. The owner of yesterday is not necessarily the owner today. Domains change hands, providers are acquired, certificates are reissued, and the operator behind the URL can change without any change to the URL itself.
- **Jurisdiction:** the law that governs the space. A URL hosted in one country answers to that country's legal regime, regardless of where the application loading it operates or where the user lives.
- **Access control:** who can write to the space. This includes who can publish content, who can change configuration, who can rotate keys, and who can reach the underlying infrastructure.
- **Allowed activity:** what the space is permitted to do. Most URL operators are subject to no enforceable constraint on what content they serve, what scripts they push, or what behavior they direct.
- **Risk relationship:** what the application accepts when it loads from that space. Every URL the runtime fetches is a continuing relationship with whoever currently controls the URL, on whatever terms that party chooses, under whatever oversight regime applies to them.

A URL whose ownership is unverified, whose jurisdiction is unknown, whose access control is opaque, whose allowed activity is unconstrained, and whose risk relationship has never been examined is not a passive string. It is a delivery point inside a supply chain.

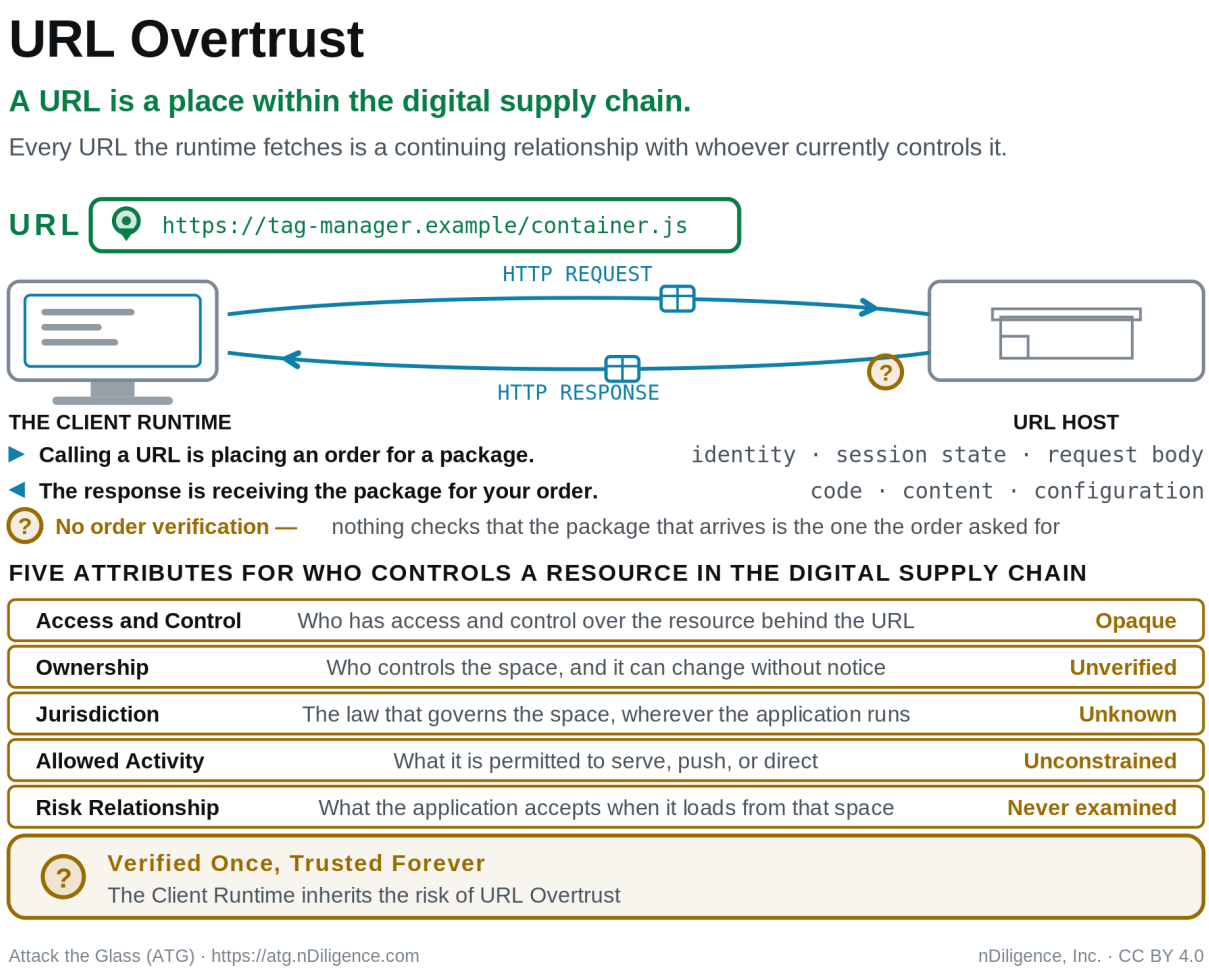

***Figure 10**: A URL is a place. Five attributes decide what loading from it commits you to.*

URLs and the HTTP payloads passed between them operate like a shipping-and-receiving layer. Calling a URL is sending a package: the request carries the application's identifying information, session state, and any data the request body includes. The response is receiving a package: code, content, configuration, or instructions the runtime then acts on. Every URL the runtime touches is a two-way exchange in which the supply chain runs in both directions.

Each URL is therefore a layer in the supply chain. A compromised URL can return a payload that arrives looking like the expected resource and behaves correctly long enough to establish trust. It can equally return one that does damage the moment the runtime processes it. Either is delivered through a URL the application chose to trust at runtime. Neither is detectable by anything in the loading chain that does not inspect the response itself, with knowledge of what the response should be.

Three properties make URL trust an attack surface no current control regime manages: scale, opacity, and payload vastness.

- **Scale:** a modern application loads dozens to hundreds of distinct URLs at runtime, across the producer, intermediary, transformation, distribution, and delivery layers.
- **Opacity:** the ownership and operational control behind any given URL is rarely known to the application's operator and is changeable without notice.
- **Payload vastness:** the aggregate volume of code, content, configuration, and AI assets flowing through these URLs every session is far beyond the inspection capacity of any defensive control that operates at runtime.

Few organizations can answer the question that follows from those properties: what URLs does our application load at runtime, and who controls each one? The honest answer in most cases is that the inventory does not exist.

The addressing and delivery system of the digital supply chain runs without a governance regime, without an inspection regime, and without an inventory underneath it. It treats a URL as a fixed, safe resource when the URL is in fact a position in a supply chain that can, by design, return different content on any request. That is URL Overtrust. The mistake is not that the trust is too generous. It is that the thing being trusted was never a fixed file sitting at rest, so there is nothing stable there to trust.

2.7: The Static Resource Assumption

Industry supply-chain practice rests on a dangerous and widespread fiction: that a resource fetched from a URL is the same resource the next time it is fetched, and that hashing, signing, or reviewing one fetch protects every future fetch from that URL. The fiction is structurally wrong, not occasionally wrong. A URL resolves to a resource that can be generated per request, so the same address can return different content to different requesters; the thing was never a static file to begin with.

In a CSR context, the resource at a given URL can change between fetches without changing the URL, the request headers, the application code, or anything else a defender would inspect. The endpoint controls what it returns. Whoever controls the endpoint at the moment of the response controls what the application receives, regardless of what was published, signed, or reviewed at any earlier moment.

Manufacturing on demand, not mass production, is the closer model. Most readers carry an unstated mental picture in which URL responses come from a fixed-inventory warehouse: a producer made the file some time ago, the file sits on a shelf, and every fetch retrieves the same thing from the shelf. The picture is wrong. URL traffic is on-demand manufacturing. Every response is built freshly when the request arrives, by whoever currently controls the endpoint, with whatever inputs and instructions that controller

chooses to apply at that moment. There is no file on the shelf. What sits at the address is a piece of code that builds the response, and it can be rewritten between any two requests.

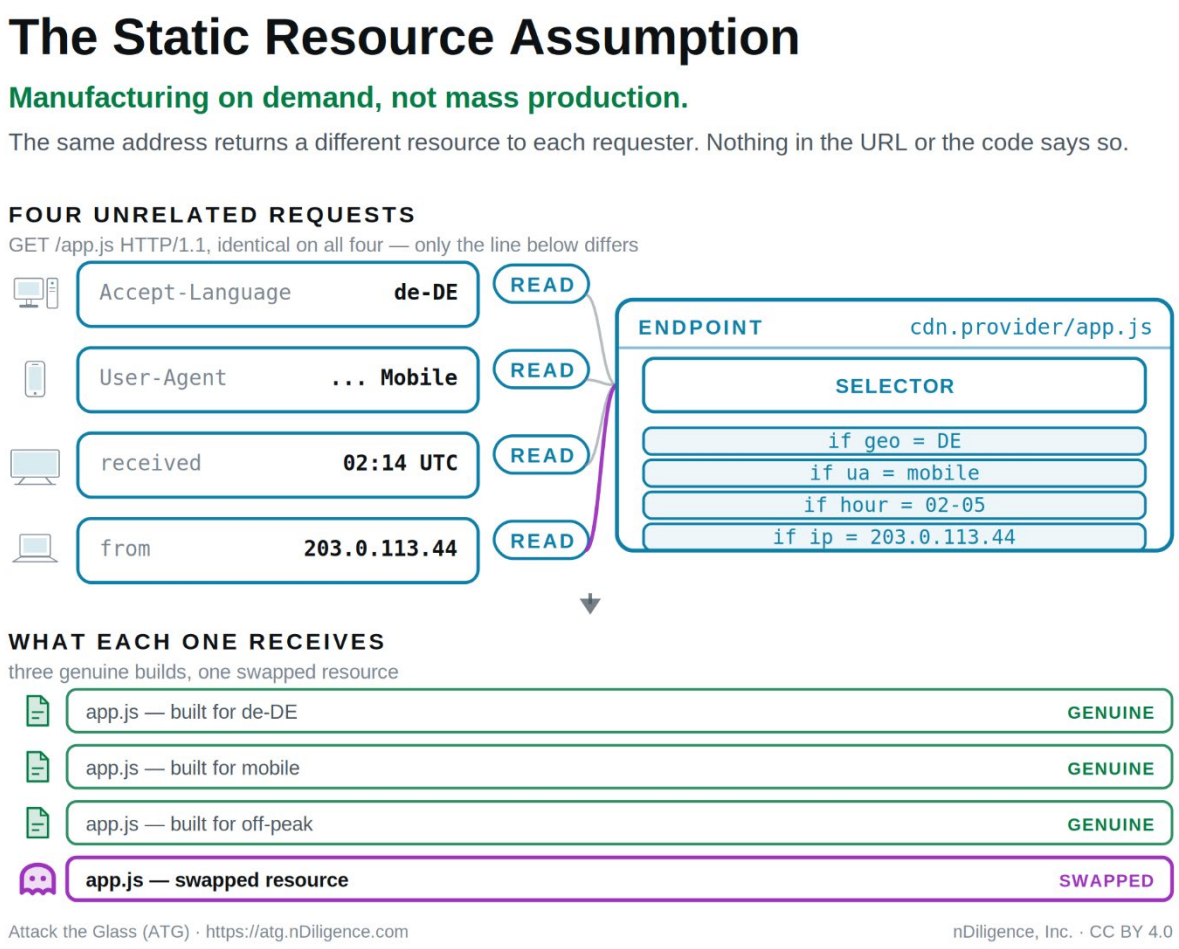

***Figure 11**: The resource is manufactured per request. Checking it once says nothing about the next delivery.*

The Static Resource Assumption underlies most existing software-supply-chain controls (signing, hashing, attestation, software bills of materials, integrity checks at build or release time) so those controls check the file at build or release time and never check the copy the runtime actually fetches. The architectural pattern is that verification finishes before the code runs. The Static Resource Assumption is the belief that pattern leaves defenders holding: that checking once was enough.

Operationally, the Two-Stage Attack rests on this. An adversary who controls the endpoint can serve a clean file to inspectors, scanners, and reviewers, and a malicious one to the runtime under attack, without changing anything a defender can inspect. The on-demand manufacturing model is not a metaphor for the adversary; it is the adversary's working environment.

2.8: The Digital Supernode

Runtime traffic concentrates at a small number of delivery-layer providers. Content delivery networks are the visible case: a handful of operators serve scripts, fonts, configuration, and assets to a substantial fraction of the world's Client Runtimes every day.

Conventional analysis treats a CDN as a chokepoint: one node through which many dependent applications fetch their resources. Treating a CDN as a chokepoint is right as far as it goes and stops short of what is actually there. A CDN is not a single node. It sits at a hub where millions of applications, sites, and organizations all depend on the same point, with millions of source libraries, each with its own producers, intermediaries, transformation points, and distribution channels, feeding into it.

The *Digital Supernode* is a dependency at such a hub: a dominant CDN, a near-universal library, a key service on which millions depend at once. It reaches that position through Herd Trust: the more places that use it, the more trustworthy it looks, and the more places adopt it. That concentration makes it one of the highest-value targets in the supply chain, because controlling one Digital Supernode means controlling what millions of dependent applications receive. Behind the visible chokepoint sits an aggregation of millions of dependent supply chains. Compromise at any one of those underlying chains can be propagated through the centralized node to every application that consumes from it.

Beyond CDNs, the pattern repeats. Widely used JavaScript libraries are nested supply chains: each library bundles many upstream packages, and the same library is loaded into thousands of host applications. Open-source package registries are nested supply chains: a registry is a chokepoint that aggregates millions of independently maintained packages, each with its own dependency tree. Model repositories are nested supply chains: a repository aggregates millions of models, each with its own training data, fine-tuning history, and inheritance chain. Analytics endpoints, tag managers, identity providers, advertising networks, and inference services follow the same shape: a small number of providers each aggregating millions of underlying dependencies and reaching into the same population of Client Runtimes.

Supernodes also sit inside one another. A library that thousands of applications bundle can itself fetch a font service, a polyfill provider, or a telemetry endpoint while it runs, and that provider is then reached by every application that bundled the library, none of which chose it or knows it is there. Dependency chain depth is what makes a position compound: each layer carries the applications above it down to the provider at the bottom, which inherits all of them at once. An application's own inventory records the libraries its developers selected. It does not record what those libraries reach for at runtime, so a provider no one contracted with can sit deeper in the chain than anything the inventory names.

Compromise at any of these layers reaches every dependent application at once, at runtime, inside the IOTG composition, inheriting the trust the host application granted to the endpoint. The reach is not theoretical. It has been demonstrated in confirmed historical events.

Centralization of risk in the delivery layer is not a CDN problem. It is the structural property of every layer at which many applications consume from the same provider. Wherever scale creates aggregation, aggregation creates a Digital Supernode, and the node sits on top of the supply chains it inherits. Own the node, own the millions who trust it.

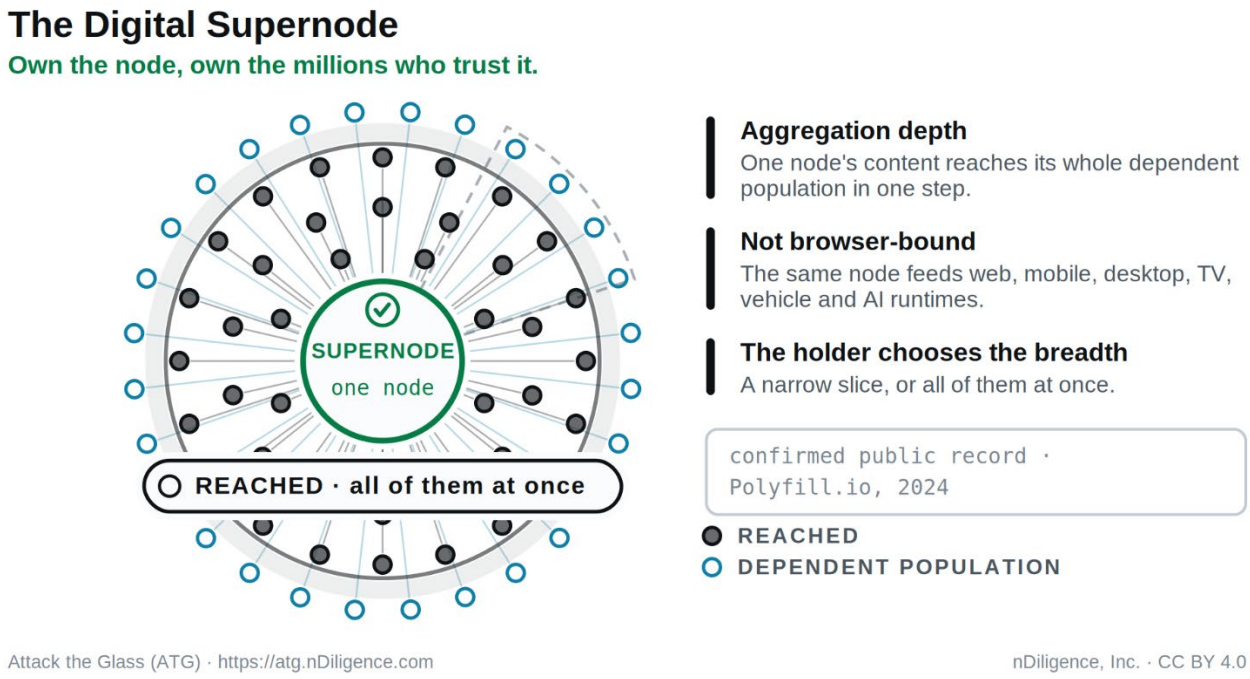

***Figure 12**: One node, millions of dependents. Concentration turns a single position into population-scale reach.*

2.9: The ATG Vulnerability Defined

Attack the Glass (ATG) is a type of supply chain injection vulnerability that targets client-side rendering: an adversary controls, modifies, or substitutes the content, code, or configuration delivered to the rendering surface inside a Client Runtime. What the adversary puts in its place then runs inside the assembled application with the full permissions of the legitimate code. The Client Runtime does not continuously re-verify the sources or content influencing its behavior after the initial trust grant.

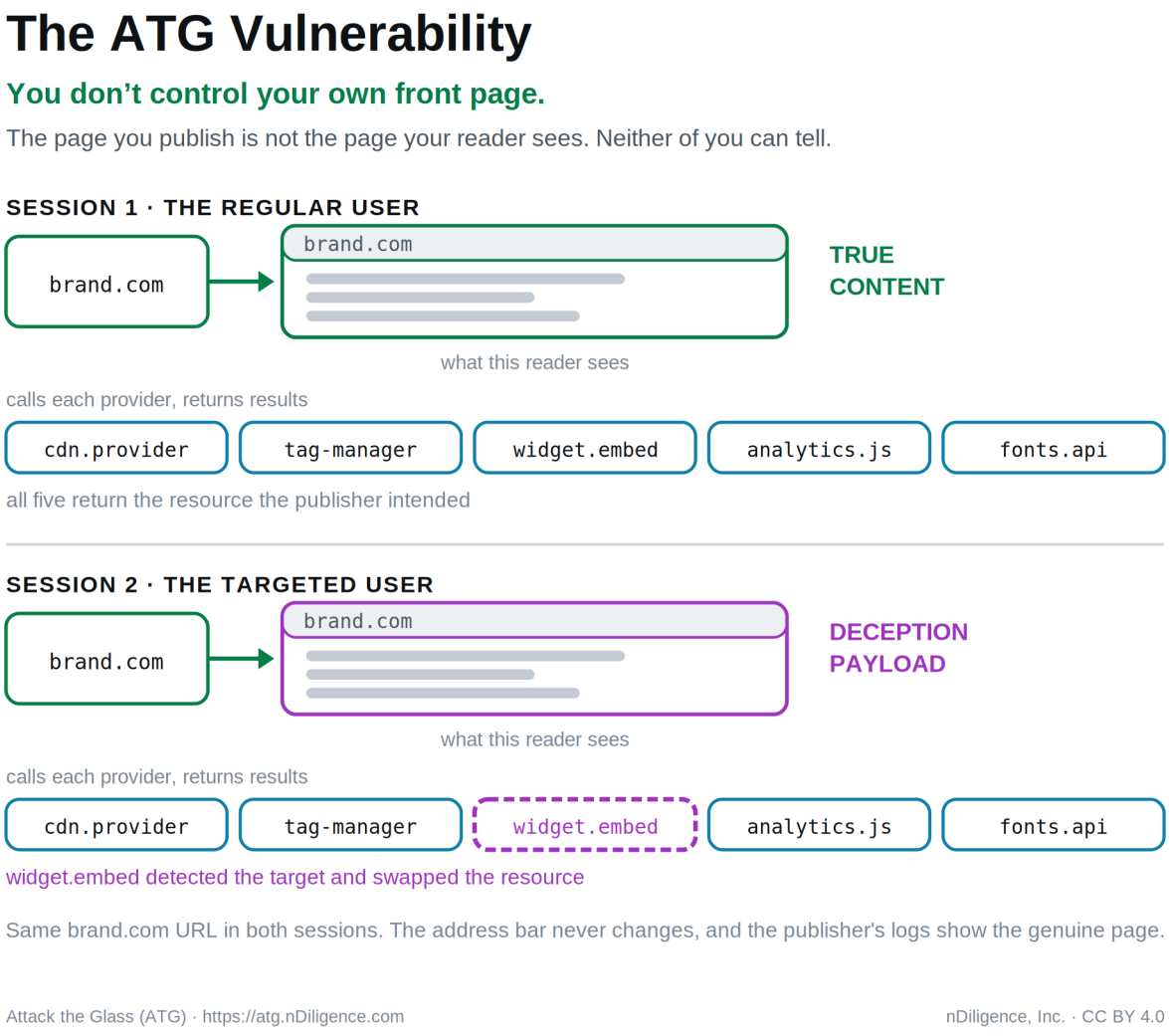

***Figure 13**: The page the publisher serves is not the page the reader receives, and neither of them can tell.*

Three structural parts make the definition precise: what is delivered, what the adversary does, and what makes it a vulnerability.

- **What is delivered:** content, code, or configuration delivered to the rendering surface. This covers both pathways CSR uses. The first is executable code loaded straight into

the runtime. The second is content the runtime interprets (data, media, configuration, templates, prompts) that then drives further execution.

- **What the adversary does:** the adversary controls, modifies, or substitutes what is delivered. Substitution can occur at the cache, in transit, at the configuration source, at the data endpoint, or at the inference response. In every case, the pattern is the same: the swap happens at the moment of delivery, and never touches the copy that was inspected.
- **What makes it a vulnerability:** inside the assembled application, the substituted code runs with the full permissions of the legitimate code, inside a Client Runtime that does not continuously re-verify what it loads after the initial trust grant. The exposure is created by the architecture of the runtime, not by the operational choice of any one operator.

ATG is a client-side supply-chain vulnerability that lets whoever controls one component of a modern application change what its users see, or what the application does for them, and deliver that change through a source the users and the application both trust. Who receives that change is the adversary's choice, and the range runs from every user of the application, through a region, a language community, or a segment defined by behavior or demographics, down to one named individual on one device. It reaches the web browser and the systems beyond it, up to and including critical infrastructure. Operations of this kind can be configured to leave no forensic evidence on the device and to trigger no intrusion detection.

One asymmetry follows directly from the definition. The party that decides which third parties a device will trust is not the party that bears the consequences of the decision.

ATG is not a single attack technique. Different surfaces, different mechanisms, and different attackers produce different versions of it, and the architectural pattern underneath stays the same.

Every Client Runtime inherits the exposure. Browser, mobile-application runtime, hybrid desktop shell, smart-TV runtime, vehicle infotainment, AR and VR scene engine, IoT and operational technology (industrial control) dashboard, productivity-tool extension host, and plugin marketplace are each a Client Runtime, and ATG is the exposure they all share. The architecture creates the exposure, not the host platform.

2.10: Supply Chain Injection

Supply Chain Injection is the structural condition in which adversary-controlled content is introduced into a trusted delivery path. ATG is the specific instance that targets the rendering surface inside a Client Runtime. Substituted content executes there with the full permissions of the legitimate code. Beyond the rendering surface, the same mechanism operates at every layer of the digital supply chain. A compromise anywhere upstream flows forward, through composition, to the glass.

Across every layer, the mechanics share one property: the swap happens when content is delivered, and never touches the copy that was checked. The file that was reviewed, signed, hashed, and approved is not necessarily the file that arrives.

Five patterns illustrate the structure: cache and CDN substitution, network-path substitution, configuration-driven substitution, tag-manager and feature-flag substitution, and inference-response substitution.

- **Cache and CDN substitution:** what the cache hands out differs from what the source published.
- **Network-path substitution:** what arrives over the wire differs from what the source sent.
- **Configuration-driven substitution:** the runtime fetches a different file because configuration delivered through a separate channel told it to.
- **Tag-manager and feature-flag substitution:** the runtime executes different content based on flags delivered through a separate channel.
- **Inference-response substitution:** the model's answer is content the runtime acts on, and it is built fresh for each request, so there is no checked copy to compare it against.

What unifies the patterns is a separation in time. The moment a copy is checked and the moment content is delivered are not the same moment. Static integrity verification (signing, hashing, Subresource Integrity) protects the copy that was checked, not the copy that is delivered. This is why ATG, the rendering-surface instance of Supply Chain Injection, defeats pre-delivery controls: every pre-delivery control inspects the copy that was checked, and the runtime executes the copy that was delivered.

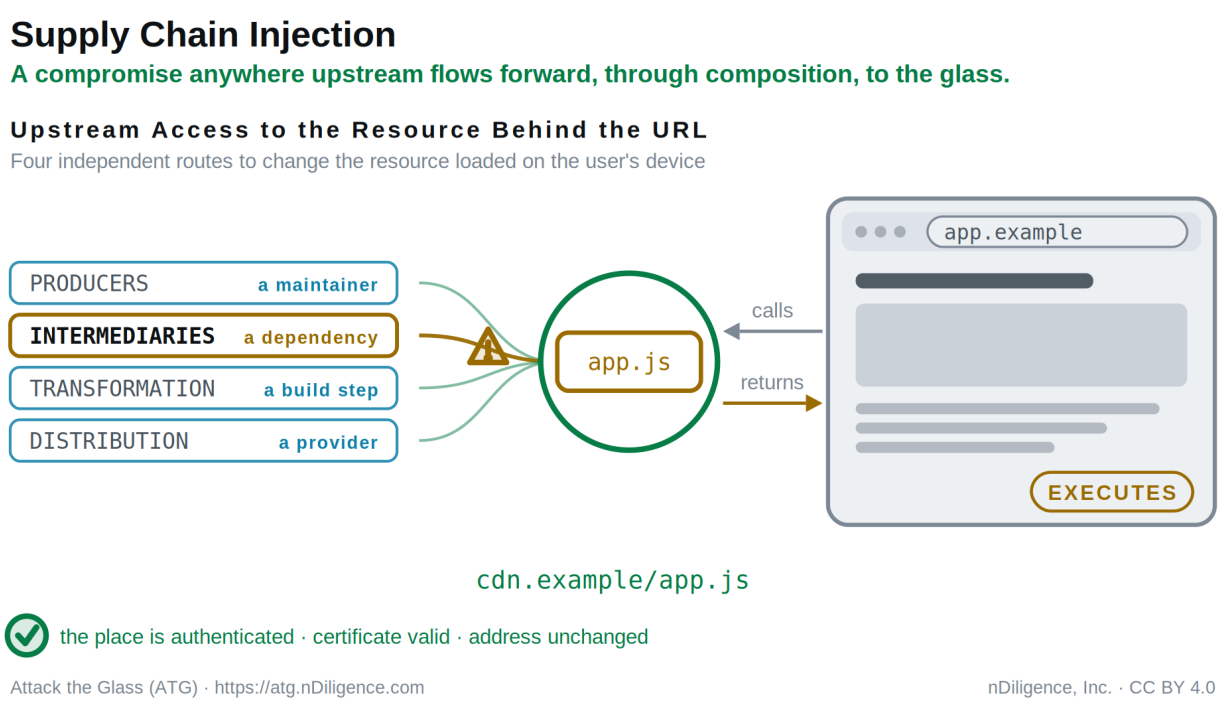

***Figure 14**: Injection travels forward. A position upstream reaches every runtime downstream of it.*

Placement and Access (P&A) is the position from which an adversary can act on the supply chain. Every P&A position is an injection point in this sense. The injection point is structural, not exploit-dependent, and it persists as long as the P&A position persists.

2.11: The Two-Stage Attack

The Two-Stage Attack splits delivery into two parts. The first stage is the visible code: the script, module, or bundle that the runtime fetches at session start and that any reviewer, scanner, or signing process operates on. The first stage is benign by design and contains no payload any pre-delivery control would flag.

Stage two is the data the first-stage code consults at runtime. The first-stage code reaches out to a data endpoint and uses the response to drive its behavior. The data endpoint can be a configuration service, a feature-flag service, an analytics endpoint, an inference API, or any other URL the code is structured to call. The data endpoint is what the adversary controls.

What makes this work is the structural separation of stage one and stage two. Verification of the first stage proves nothing about the second. What a reviewer sees is the bundle; what actually happens is whatever the data endpoint tells the bundle to do at runtime. The separation is the defensive blind spot: the code a defender can inspect is not the thing that decides what runs. The code looks clean because the weapon is hidden in the data it loads.

Obscuring attribution and inspection is part of the design. Resources loaded into the Client Runtime during a session can be nearly impossible to detect as they arrive: they are data, not code, and the loading code is benign. They are nearly impossible to track back to their point of origin afterward: the data endpoint may have stopped serving the malicious content by the time investigators look, and the runtime keeps no audit trail of what each call returned.

Deepfakes generated client-side, runtime media modification, and runtime data manipulation are concrete instances of two-stage payloads. Each loads a benign first-stage element (a media library, a chart renderer, a numeric formatter) that consults a data endpoint for the actual content. On-device AI amplifies the class: the same loading code that fetches an inference response can be steered to produce per-target deceptive media, per-target deceptive numbers, per-target deceptive narratives, and per-target deceptive recommendations, none of which appear in any file a reviewer could inspect. Information manipulation is the dominant ATG attack mode, and the Two-Stage Attack is the operational mechanism through which information manipulation can be delivered today at scale without detection.

The two-stage architecture defeats static integrity verification (signing the file, hashing it, Subresource Integrity, package-level scanning). It does not defeat runtime behavioral monitoring at the rendering layer. Static integrity verification operates on the file as it sits at rest; runtime behavioral monitoring operates on what the runtime is doing in the moment. They are different kinds of control, they are complementary, and the appropriate response to two-stage attacks is the latter deployed alongside the former rather than in place of it.

Underneath the two-stage pattern sits a structural property that holds for any runtime injection, not only the two-stage form. *The Runtime Injection Advantage* is that property. Four features define it:

- **After all pre-delivery verification has passed:** whatever review, signing, scanning, or approval the file went through is complete. The runtime cannot check again, because the copy that was verified is not there; only what was just delivered is, and that is what runs.
- **Per-request and per-target:** runtime injection can deliver different content to different requests, different users, different contexts. The same delivery channel can serve one response to inspectors, scanners, and reviewers, and a different response to the runtime under attack.
- **Difficult to reproduce in forensic analysis:** when investigators try to reproduce an attack after the fact, the adversary may have stopped delivering the malicious content, leaving no evidence in the runtime that anything was wrong.

- **Compatible with arbitrary trust granted to the source:** the runtime granted trust to the source (CDN, package, script host, inference endpoint) at the moment of loading; injection at runtime inherits all of that trust by design.

Together, these four properties make the Runtime Injection Advantage the structural reason ATG-class attacks are difficult to detect with traditional defensive postures and difficult to investigate after the fact. The Two-Stage Attack is the contemporary form of ATG that exploits the advantage most fully. It is the most sophisticated form of the class known to date.

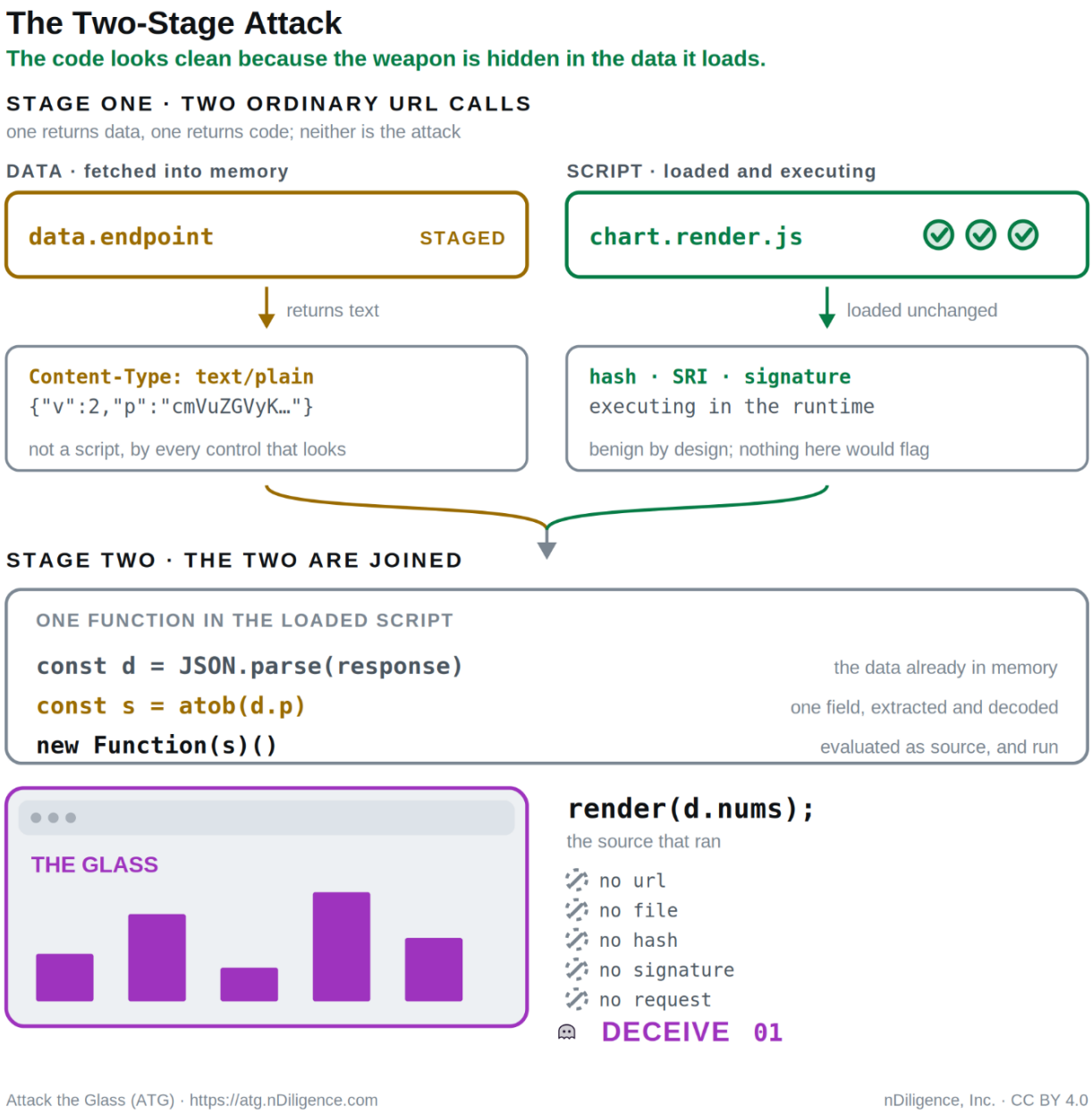

***Figure 15**: The position is the first stage. The activation is the second, and only the second is detectable.*

2.12: Content Swapping

The delivery analysis establishes that an adversary in the supply chain can substitute what the runtime receives. What that buys depends on what is substituted on the rendered surface, and the forms are not interchangeable.

Content Swapping is the family of on-page changes an ATG operation produces: a value, an image, a link, a form field, or a chart rendered as something other than what the application intended. The change can be introduced in the delivered resource, in the data a loaded script consults at runtime, or in the rendered output after both have arrived. The category names the change the user sees, not the channel that carried it, and it is the surface reading of what the D5 analysis calls information manipulation: the same event described by what it does to the page rather than by what it does to the person reading it. Eight forms follow from the architecture. Injected content and link swaps also appear directly in the confirmed record.

- **Data-value swap**: a number, price, quantity, date, status, or identifier renders as a different value than the source returned.
- **Media swap**: an image, video, or audio element renders content other than what the application referenced.
- **Link swap**: a link or action target resolves somewhere other than where the application intended.
- **Injected content**: content the application never referenced appears in the rendered output.
- **Suppressed content**: content the application did reference does not appear.
- **Reordered content**: the sequence of items in a list, feed, or result set differs from the order the source supplied.
- **Form-value swap**: a field value differs between what the user entered and what the application submits, or between what the source supplied and what the user is shown.
- **Chart-value swap**: the series behind a visualization is altered, so the rendered chart is internally consistent and factually wrong.

Most of these do not require the application to fail. The application continues to run correctly in the only sense it can measure: it renders what it was given. What it was given is the thing that changed. That property separates most Content Swapping from functional attacks, and it is why a swap can carry through a session, a workflow, and a decision

without producing an error anyone would investigate. Suppressed content is the partial exception, since removing a referenced element can produce a visible failure.

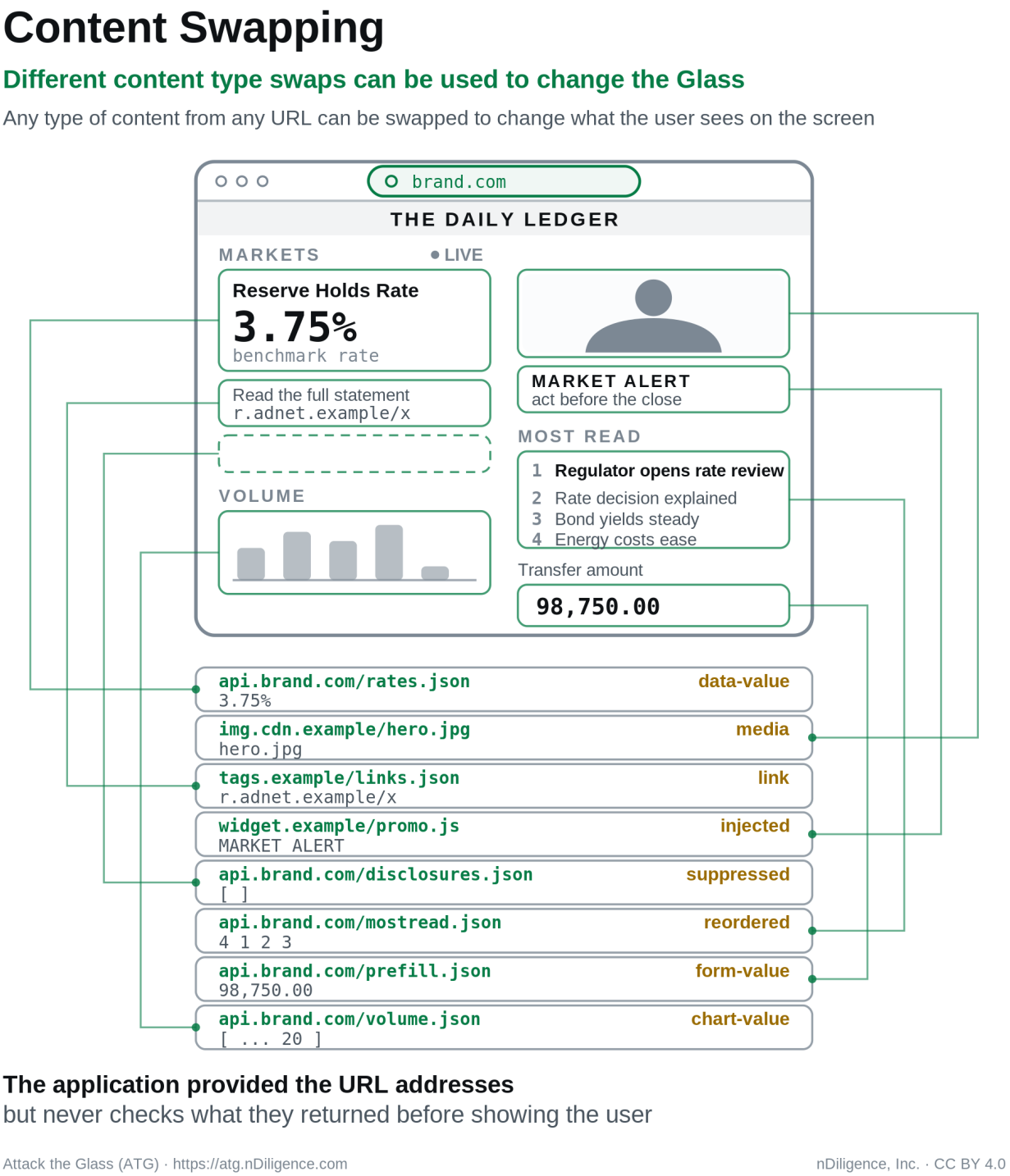

***Figure 16**: What arrives can be substituted, added to, or removed, with the address unchanged throughout.*

The eight are forms of visible change, and they are drawn from the browser and mobile case. Data-value, suppressed, reordered, and chart-value swaps carry across to an operator surface as setpoints and readings, alarm suppression, sequence and priority order, and trend series. Link, media, and form-value swaps map poorly there or not at all. A control-surface taxonomy would not be these eight.

Everything the runtime receives that the application did not reference is an injection, whether it is markup, a value, an image, a stylesheet, or code. The distinction that matters is not code against content but whether the runtime acts on what it receives or only presents it. Presented content reaches its effect by being consumed. Actioned content is interpreted and executed by the runtime and needs nobody present at all. The line between the two is not fixed, which is Data as Code applied to this taxonomy: a data payload is presented until something evaluates it, templates it, binds it to a handler, or supplies it to a model as instruction. Actioned injections carry the widest reach, because they can manufacture any of the eight forms at will, and because they can also produce effects that render nothing at all. A destructive call, an exfiltration, and a silent write each leave the surface looking exactly as it should.

Style Injection is the least recognized actioned form. The presentation layer is altered by a stylesheet the runtime loads or by script that writes style while running, so the rendered surface no longer represents the data behind it. A control can be hidden so the user cannot reach it, a color-coded threshold restyled so it reads as its opposite, a list reordered visually while the source order is untouched, a control taken out of service while remaining visible and pressable, and layout and filter work driven until the surface is unusable. Stylesheets are interpreted and acted on, and they are rarely treated as executable, rarely enumerated in a dependency inventory, and exempt in practice from the integrity reasoning applied to script. The case is the cleanest available statement of the pattern: the data is correct, the render is wrong, and every integrity check on the data passes.

2.13: Post-Load and Post-Trust Modification

Content Swapping catalogs what changes on the rendered surface. When the change happens determines what, if anything, was in a position to notice it. The most consequential timing is the latest one: after the resource has been fetched, after whatever integrity checking was going to run has run, and after the runtime has admitted the content into the executing application.

Post-Load and Post-Trust Modification is that case. Data that arrived over an authenticated connection, passed the verification the application applied, and was accepted as correct is then altered in memory by code already running in the same context. The value the user

sees is no longer the value that arrived. Nothing in the loading path re-examines it, because the loading path finished before the modification occurred.

The mechanism requires no exotic capability. Code executing in a Client Runtime can read and write the global objects, the shared prototypes, and the rendered nodes that other code depends on, which is the ordinary condition of the runtime rather than a defect in any one application. A script that legitimately formats a number can rewrite it. A component that legitimately renders a list can reorder the list after it arrives. A charting library that legitimately draws a series can draw a series other than the one it was handed.

Verification and modification are separated in time, and that separation is what distinguishes this case from a delivery-side swap. A delivery-side swap defeats integrity checking by changing what is delivered. Post-load modification does not need to defeat integrity checking at all. It lets the check pass and changes the result afterward. Everything the check measured was true at the moment it was measured.

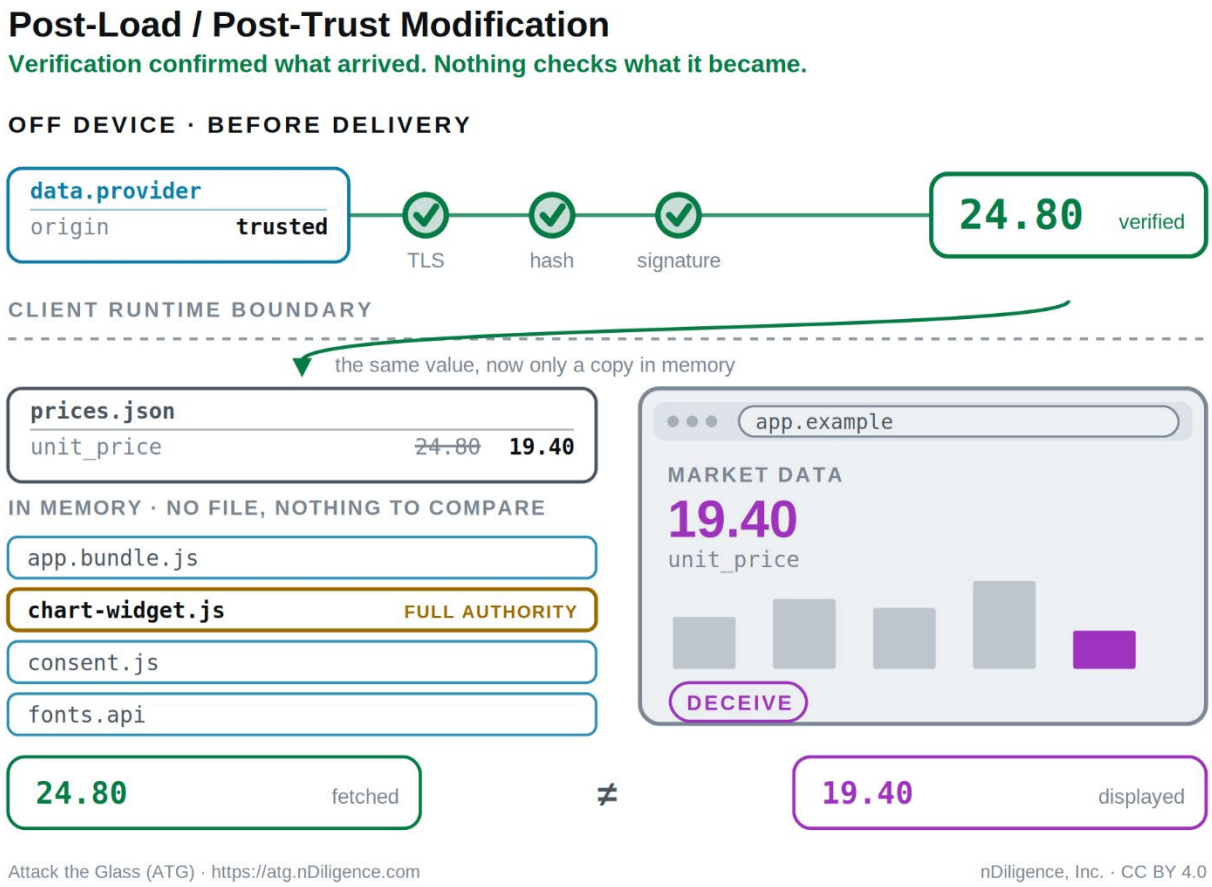

***Figure 17**: Trust is granted at load. Content changes after. The two moments are not the same moment.*

The practical consequence is a matter of order. Changes delivered through this channel run after the existing cybersecurity defenses have done their work, in what the research

calls the post-trust and post-load environment. Every control that operates on delivery has already operated. Every check that was going to run has run and passed. The environment is not one where defenses failed; it is one they have already exited.

2.14: Targeted Delivery

The Runtime Injection Advantage establishes that a delivery channel can serve one response to inspectors and another to the runtime under attack. What decides which requester falls into which group is the operationally interesting part, and it is ordinary commercial infrastructure.

Targeted Delivery is the practice of conditioning what a resource returns on who is asking. The conditions available are the conditions any content-personalization system already uses: source address and inferred geography, device and user-agent characteristics, time of day and session timing, authenticated identity, prior behavior, cohort membership, and experiment-assignment buckets. An operation that can express its target as a rule in that vocabulary can deliver a swap to the target population and clean content to everyone else, from the same URL, in the same session window.

The middle of that range is where an influence operation lives, and it is worth working through rather than asserting. One position inside one advertising or analytics provider can serve, in the same hour and from the same address: the genuine article to most readers; a version with one paragraph altered to readers whose address resolves to a particular country; a different alteration again to readers whose prior behavior places them in a commercial or political segment; and the genuine article once more to any requester whose characteristics look like a scanner, a fact-checker, or a newsroom. No group can see another group's version, and the publisher's own servers were never touched.

Precision down to a single device is the extreme of the range rather than its center. It demonstrates how fine the targeting can be, and an operation seeking reach would not use it. The operations that matter at population scale are the ones that segment: one position, one trusted source, different content to different groups, simultaneously.

Investigation after the fact meets the difficulty from the other direction. The population that could confirm the swap is the population that received it, and that population had no reason to know it had. None of this requires the adversary to hide. It requires only that the adversary answer a different question than the one the defender is asking.

Deciding who receives a swap and actually delivering the swap are two separate things. The rule can be resolved by the same personalization and advertising infrastructure the commercialization analysis describes, and that infrastructure is not the vector. The swap is

carried by code running in the application's own context. The same rules that decide which advertisement a person sees can decide which person receives a swap.

Narrowing the target set is deliberate on the adversary's part, and it makes detection harder rather than limiting the harm. A swap delivered to a thousand people out of a hundred million is harder to detect than one delivered to all of them, and it is worth more to an operation that only ever cared about the thousand.

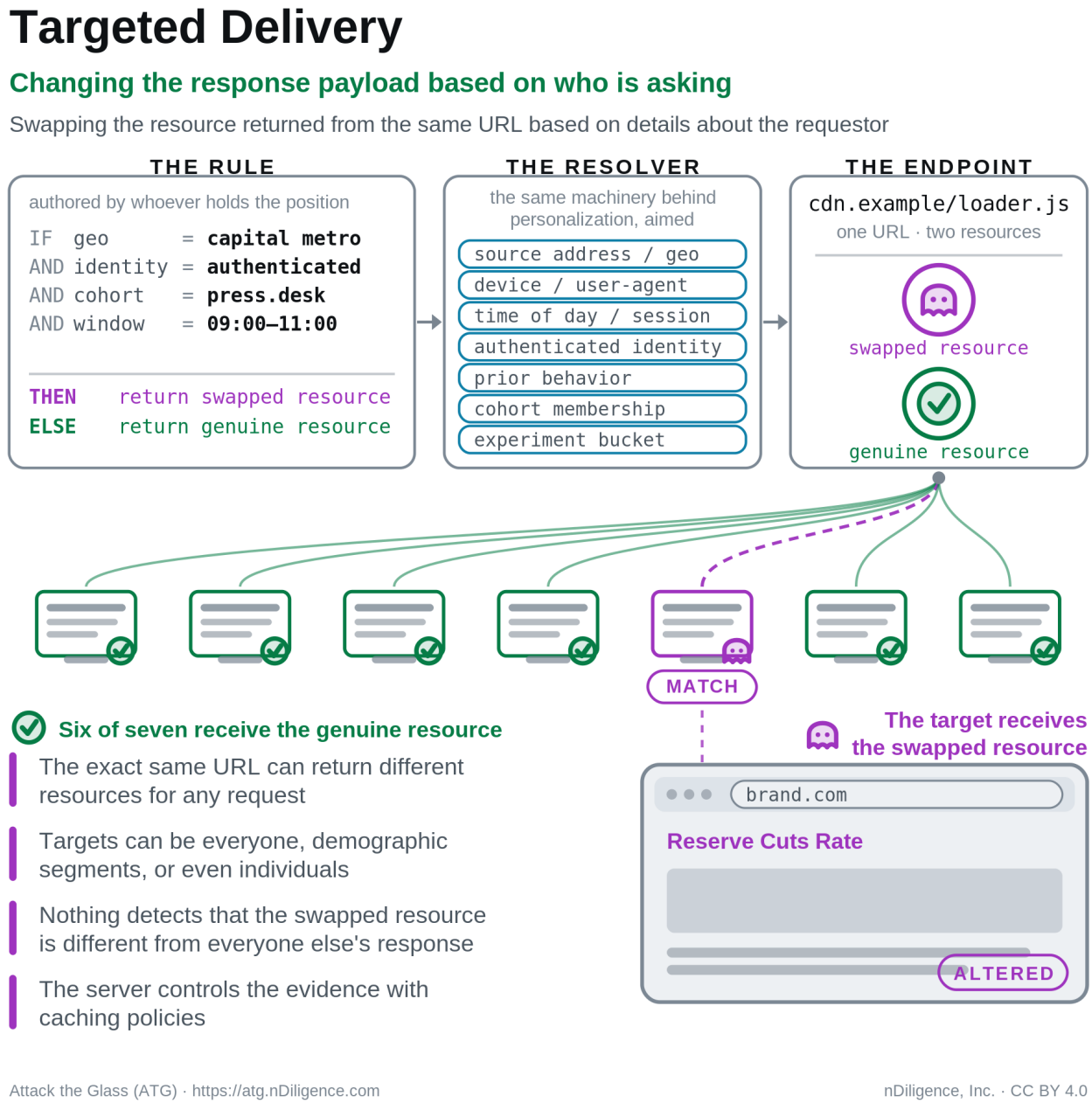

***Figure 18**: Delivery can be conditioned per recipient, so no two observers necessarily see the same thing.*

2.15: The Integrity Blindspot

In the common case, no party can establish at the moment it matters whether the content in front of them is what it should be. That inability is not a gap in anyone's tooling. It follows from the architecture, and it applies both while the content executes and afterward.

The Integrity Blindspot is that condition. Where no authoritative reference exists for what a given resource should be at the moment it is delivered, there is nothing to compare the delivered resource against, either live or later. The invisibility is structural rather than an oversight in any one product: monitoring inside the rendering boundary, where the substitution occurs, is not deployed at meaningful scale in the current landscape.

- **The Detection Blindspot:** nothing verifies, at the moment of execution, that a fetched resource matches an authoritative reference. Subresource Integrity is the closest the platform comes, and the existing-defenses analysis sets out where its coverage ends. The verification that does exist runs before execution and covers the file as it was delivered, not what it does once it runs.
- **The Forensic Blindspot:** nothing is recorded that would let the substitution be reconstructed later. The substituted resource existed in memory for the duration of a session and is gone when the tab or the application closes. Unless traffic is intercepted at a proxy that terminates TLS, a network monitor records only metadata, timing, size, and destination, never the content itself. In the post-load and targeted-delivery cases there is often no file left to scan and no distinct domain to block, and the substitution leaves no entry in the logs the defender keeps.
- **The Missing Source of Truth:** the application owner declares what a resource is at build time, and nothing keeps that declaration current at the moment of delivery. The Detection Blindspot and the Forensic Blindspot both follow from this. A live check has nothing to measure against, and an investigation after the fact has nothing to reconstruct toward.

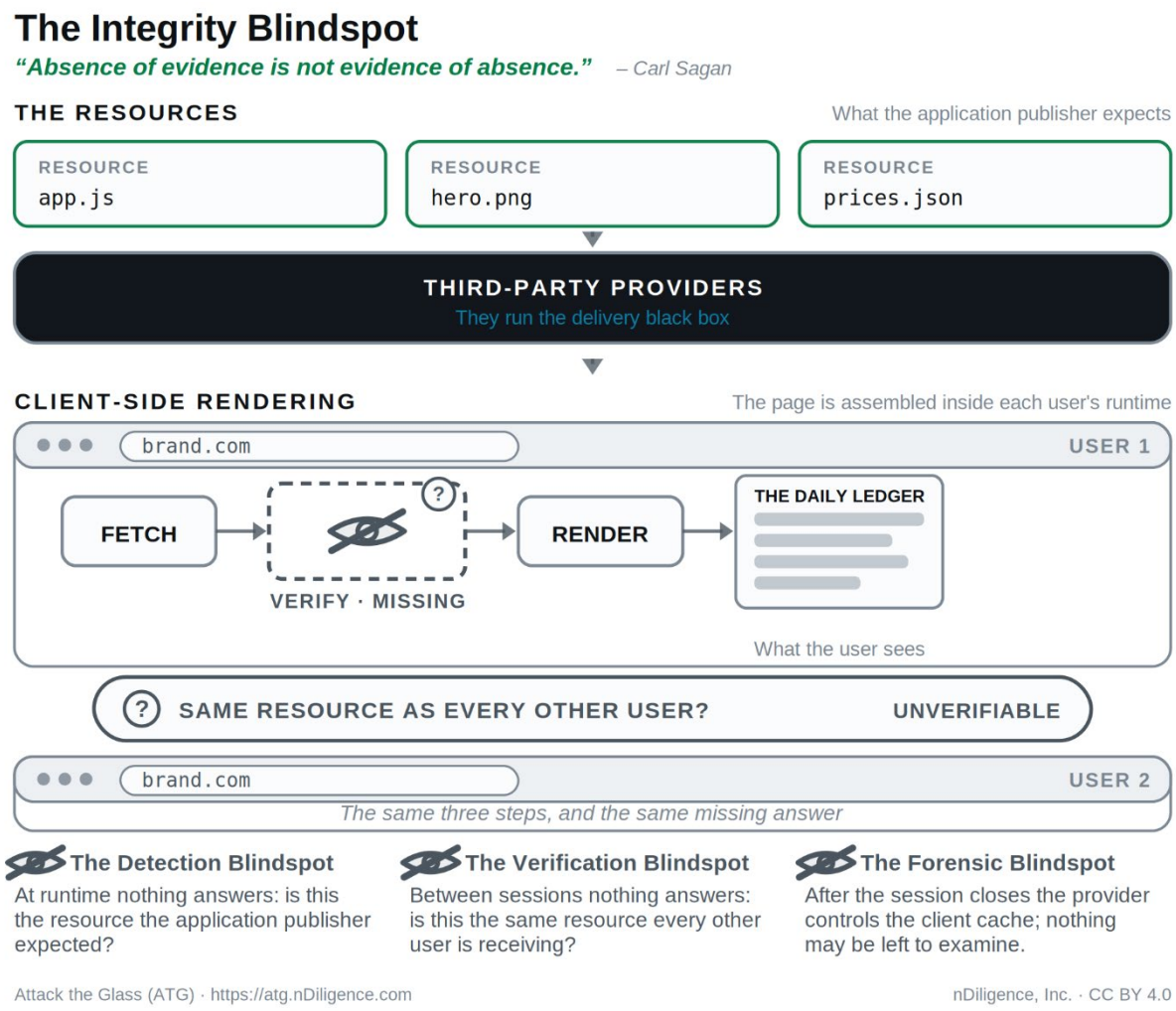

***Figure 19**: Nothing verifies at execution, and nothing is recorded afterwards to reconstruct what ran.*

Remote Cache Control closes the forensic route deliberately. Response cache directives are set by the server that answers the request, and clients honor them by default, so the party serving a resource decides how long, and whether, the client keeps it. This is the same serving-side authority the CDN analysis describes, applied to a different purpose. That server can send a substituted resource once, mark it no-store so the client's HTTP cache keeps nothing, and serve clean content to every request after that. Served to the target, the resource lives in memory for the session and is never written to the cache an investigator would go looking in. The cache is clean because it was never used.

The condition is not permanent. It holds where no live reference for runtime resources is deployed and no monitoring operates inside the rendering boundary, which is the common case today rather than a property of the architecture. An application that pins every resource, declares an integrity hash for each, and admits no dynamically generated code has a reference to check against. Few applications are built that way, and the ones that are

give up the capabilities that made CSR worth adopting. Runtime behavioral monitoring at the rendering layer is the complementary control, and the existing-defenses analysis sets out where it stands today.

Absence of evidence is not evidence of absence. A blind spot is precisely a place where not seeing something is not the same as its not being there. The instrumentation that would produce evidence of a swap at the rendering surface is not deployed at population scale, so the absence of that evidence carries no information about whether swaps are occurring. Every claim about the operational prevalence of ATG runs into this, including the claims in this whitepaper.

2.16: ATG Applies to All Networks, Not Just the Internet

Public-internet framing is the default mental model for software supply-chain vulnerabilities. ATG does not stop at the public network. The vulnerability class travels with the architectural pattern. Wherever a Client Runtime consumes content from any external source, ATG applies, regardless of which network connects the runtime to that source.

Air-gapped enclaves, classified networks, OT segments, vehicle networks, industrial control networks, and private corporate networks all host Client Runtimes. Those Client Runtimes fetch scripts, configuration, content, and data from upstream sources within their respective networks. The networks are different; the runtime is the same. An air gap does not eliminate the supply chain; it brings it inside the wire. The pattern is the Trojan Horse: high walls and isolation do not protect against what you carry inside the perimeter yourself, and where the vulnerability predates the isolation, air-gapping can simply do the adversary's work of getting it inside. Isolating a network relocates the supply chain rather than removing it.

Equating protected networks with the public internet would overstate the risk inside protected environments. The digital supply chain inside a classified network or a hardened OT segment is significantly smaller and more controlled than the public web. Access control is rigorous, nodes are validated, the inventory of upstream sources is tractable, and the adversary set is different from the population that operates against the public internet. The protected-network differential is real.

Protected networks are not immune. The Glass inside those networks is not currently defended against ATG; ATG is not yet a recognized attack vector for which industry has built defenses. Even the most secure classified networks are built on commercial hardware, run commercial operating systems, and host largely commercial software platforms. The runtime structures are the same as on the public internet; the differences are the supply-chain inventory and the adversary set.

ATG applies inside protected networks because the architectural pattern is present. The risk inside a protected network differs from the risk on the public internet because the supply chain is smaller, the controls on access are tighter, and the adversaries with positions inside that supply chain are a smaller, different group. The protected-network case is not exempt; it is differently exposed.

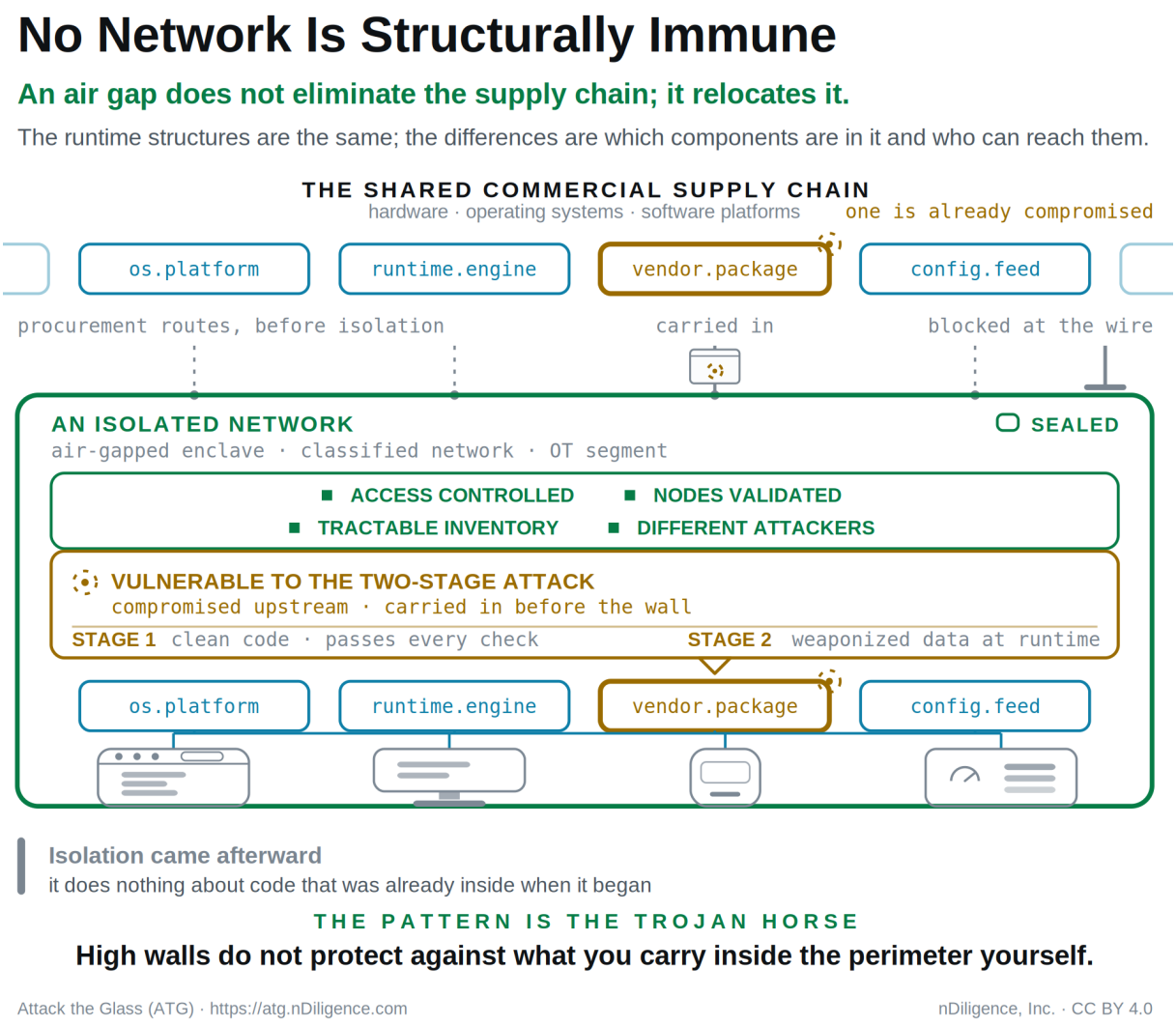

***Figure 20**: Air-gapped, private and classified networks run the same runtimes and inherit the same gap.*

2.17: D5 Effects

Operational consequence is what an attack accomplishes, distinct from how the attack is delivered. ATG can produce five classes of operational consequence, the D5 effects. The five are distinct outcomes rather than five degrees of one outcome: each is fully capable alone, combinations are ordinary rather than exceptional, and the 01 to 05 ordering is a stable index rather than a severity ranking. Every effect is scored together with the object it

acted on and the reach it was delivered to, and neither of those belongs inside the definition of the effect itself.

- **Deceive:** the consumer accepts as authoritative content that does not match what the system intended, and acts on it. False prices, doctored images, manipulated alerts, fabricated recommendations, and per-target deceptive content all sit here. The consumer may be a person or an automated one: an alarm evaluation, a control loop, an agent, or a model reads what it is given and behaves accordingly, without judging whether it is true.
- **Disrupt:** an intended function is interfered with. The function is present and does not do what it is for. Workflows fail, alerts stay silent, controls miss, features behave incorrectly. The boundary against Deceive is what was consumed: a false value that the function processes correctly is Deceive, because the function worked; a malformed value or an altered code path that stops the function achieving its purpose is Disrupt.
- **Degrade:** volume, speed, scale, or response time declines below intended levels without anything failing. The system does everything it is supposed to do and does it worse. The mechanism is induced work rather than substituted meaning: a delay inserted ahead of every outgoing request, forced repaints driven by mutating bound values, listeners and polling added to the runtime, payloads sized past what the renderer can handle. Reduced accuracy is not Degrade. Content that is wrong is Deceive, and content that is withheld or truncated is Deny.
- **Deny:** something is made unavailable or inaccessible at the rendering layer. The control is removed from the interface, the navigation point is deleted, the returned set is truncated, authentication fails, transactions block, services become unreachable, content cannot be loaded. Deny runs from a single control up to an entire application and stays Deny at every scale, because what makes it Deny is that the effect is achieved at the Glass.
- **Destroy:** adversary-controlled code reaches through the rendering layer into a backend or connected system and damages it, so that it cannot function until it is restored or rebuilt. Data and the structures holding it are destroyed, configuration or credentials the system runs on are removed, or a connected system the application drives is damaged. Two conditions are required and neither is optional: the damage occurs beyond the Glass, and the adversary's code causes it. Where a person was deceived into taking the damaging action, the effect is Deceive and the damage is its consequence. Whether a backup exists measures the operator's resilience and belongs to the consequence rather than to the effect: dropping every table is Destroy whether or not the tables can be restored.

Each effect is achievable on any Client Runtime once supply-chain control is established. The full spectrum is the operational expression of the Malice Without Malware argument: D5 attacks weaponize features of the platform and the language to produce effects that current detection systems do not recognize as malicious behavior. The result is an *uncontested operating environment* in which truly malicious activity proceeds undetected because it does not match any signature or behavioral pattern the defender's tools were built to flag.

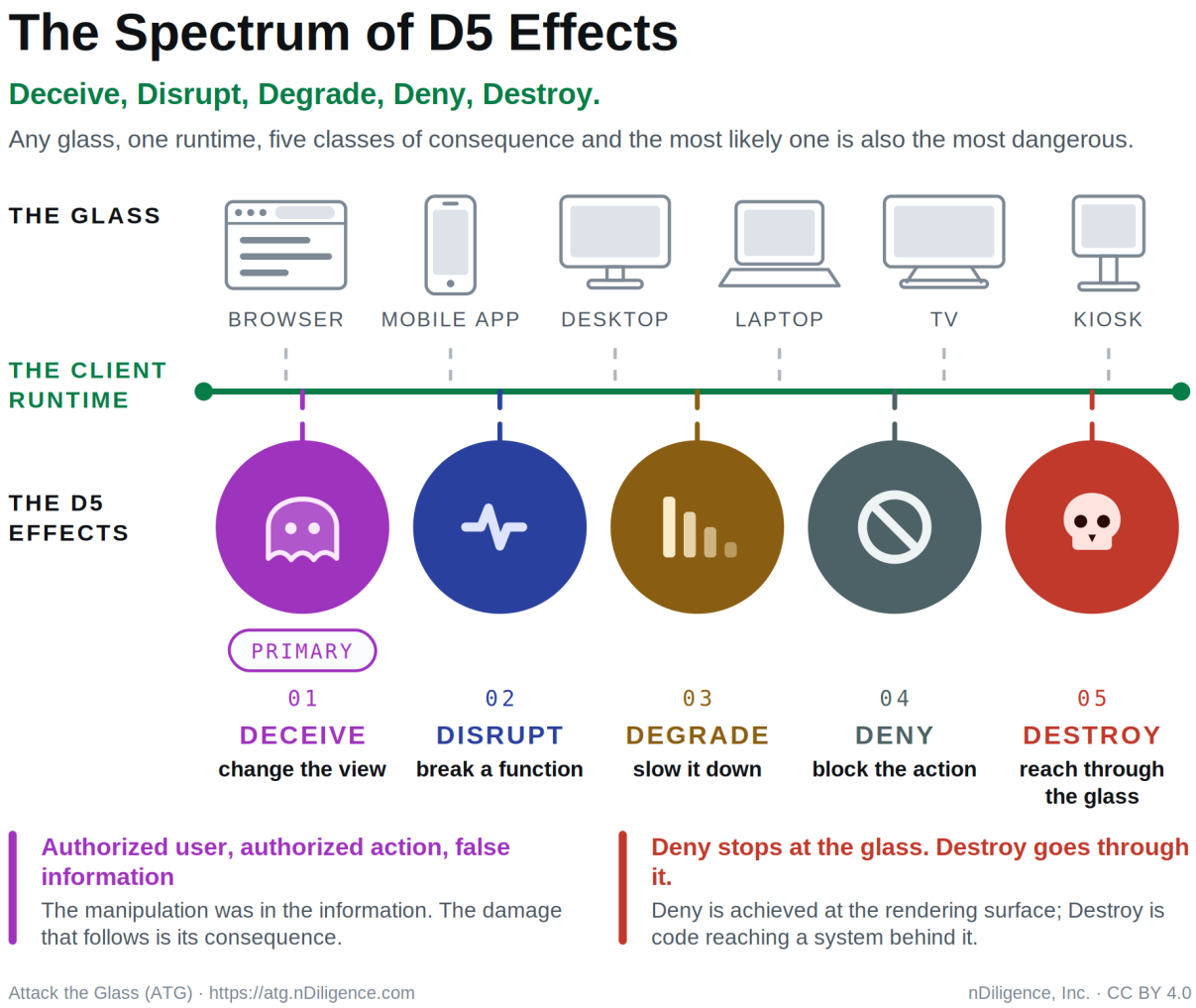

***Figure 21**: Five effects an injected resource can produce. The set is an index, not an escalation.*

Consequence and delivery are different axes. D5 names the consequence; the Two-Stage Attack, supply chain injection, and runtime injection generally name the delivery side. A delivery mechanism produces an effect, and the effect produced is the D5 score for that mechanism.

What a given effect is worth, and whether it is available at all, is set by the authority of the session the runtime is holding. A swap on a retail checkout reaches a customer's belief and

a customer's transaction, because that session has no destructive path to aim. The same swap on an administrative console reaches whatever that console is authorized to reach: a database administration interface can destroy tables, a cloud console can delete an account's storage, an identity console can revoke a directory. The adversary's capability is identical in both cases; the session's authority is not. This is the D5 expression of what Placement, Access, Influence, Control already argues, that control follows position rather than actor size. The corollary deserves stating plainly: the interfaces that operate infrastructure, from database tools and cloud consoles to CI/CD interfaces and identity administration, are client-side rendered applications assembled at runtime from the same commercial supply chain as any other.

ATG grants an adversary nothing the target application could not already do. It substitutes what the application works from. The effects available against a given system therefore depend on that system's function, deployment, users, and connections, and not every system has a pathway to every effect. A read-only reference surface has no Destroy pathway from the Glass and little worth disrupting. An operator console has all five. An administrative surface is a trade rather than a prize: a higher ceiling, a smaller user base, and often a more controlled dependency set to reach through.

Two operations outside ATG's inclusion criterion show where the line between Deceive and Destroy falls. They share a mechanism and score differently. Stuxnet was an attack on the interface: monitoring systems were shown recorded normal process values while centrifuges ran outside their tolerances. No person was deceived into acting. The automated evaluation was supplied false data so that it would not intervene, and the adversary's code reached through into the connected system and did the damage. That is Deceive against an automated consumer and Destroy against a connected system, on one operation. Operation Orchard in September 2007 scores differently. The Syrian air-defense system reportedly presented a picture that did not match the sky, and the al-Kibar site was then destroyed by aircraft ordnance. The deception was the same class of act, worked on the same kind of consumer, and produced no Destroy at all, because nothing reached through the Glass and a separate kinetic means did the damage. Severity did not decide the difference, and neither did the presence of physical destruction. What decided it was whether the code went through.

Neither case satisfies the inclusion criterion. Neither environment is a CSR-capable Client Runtime: a PLC and HMI pairing is not one, and an air-defense sensor network is further from one still. Neither belongs in the confirmed ATG record. Both are precedents for how the five effects are told apart, and that is the only work they do. Israel acknowledged the Orchard strike in March 2018; the electronic method is reported rather than confirmed and is carried as Presumed.

Within the D5 spectrum, one effect class dominates. Information manipulation is the modification of the data, content, media, and features the user sees. Within ATG, information manipulation is the easiest to execute, the most rewarding to run, and the most dangerous category of attack. *The information-manipulation hierarchy* is that ordering, and three independent dimensions support it.

- **Easiest to execute**: information manipulation requires the smallest adversary capability footprint. Most of the Content Swapping forms (a swapped numeric field, a media element, a price, an alert text, a recommendation) produce an effect without disrupting normal operation, without crashing the application, and often without leaving any structural sign. The execution barrier is the lowest in the spectrum.
- **Most rewarding to run**: information manipulation produces the highest expected value per unit of adversary effort. The same placement and access position that enables a single information-modification effect enables a population of targeted variations across users, contexts, and time windows. The economics favor information manipulation across criminal motivation (fraud), state motivation (influence), commercial motivation (competitive sabotage), and insider motivation (selective harm).
- **Most dangerous:** information manipulation operates beneath the user's verification capacity. The user trusts what the Glass shows. When the Glass shows a different number, a different image, a different sentence, a different recommendation, or a different alert state than the system intended, the user makes a decision based on the manipulated information without any signal that something is wrong. The damage compounds because the trust that produces the wrong decision is the same trust that protects the wrong decision from review.

Functional disruption (Disrupt, Degrade, and Deny effects against application behavior; alert suppression; feature subversion) sits above information manipulation in technical sophistication and below it in frequency, payoff, and damage. Destroy effects are the rarest, because they require a path through the Glass into a system that can be damaged, and most sessions do not hold one. Second-order outcomes that flow from information manipulation producing wrong decisions are consequences of Deceive and are recorded against Deceive.

A category whose dominant exposure is information manipulation sits in a higher-priority defensive position than a category whose dominant exposure is functional disruption.

2.18: Scope and Exclusions

The inclusion criterion defines ATG's scope, and it is deliberately conservative: a vulnerability is in scope for ATG if and only if it manifests through code or content execution inside a CSR-capable Client Runtime at the rendering surface. The severity-methodology analysis applies the same rule.

Two exclusions follow from the criteria: server-side runtime scripting and precompiled-only Client Runtimes.

Server-side runtime scripting is the principal exclusion. Server-side template injection (SSTI), remote code execution in server-rendered languages, and similar server-side execution patterns are not in scope for ATG. The execution environment is different. A server is not a CSR-capable Client Runtime; the rendering surface of a server-side rendering (SSR) application sits behind the operator's defensive perimeter; and the existing frameworks for server-side vulnerability cataloging cover the server-side cases reasonably well already.

Precompiled-only Client Runtimes are the structural exclusion. They are out of scope structurally, not by analytical choice.

Both exclusions are of scope, not of relevance. The D5 effects framework applies equally to server-side environments. Malice Without Malware applies across tiers. Any form of supply chain injection can produce D5 effects on any tier in a system architecture, and a server-side compromise that produces deception, disruption, degradation, denial, or destruction is a serious matter regardless of whether ATG names it.

ATG is about the unprotected client-side rendering layer inside the Client Runtime. Server-side supply chains may be more secure and more closely monitored, in part because they sit inside the traditional defensive boundary of modern cybersecurity frameworks. The client side does not. Where the traditional defensive boundary stops, the analytical focus picks up.

2.19: ATG Beyond the Browser

Beyond the browser, the Glass is everywhere. Client Runtimes render it through many non-browser hosts, and ATG applies in each of them. Six categories illustrate the breadth.

Mobile OTA bundle delivery. CodePush, Capgo, Appflow, and similar services let mobile applications fetch and apply JavaScript or React Native bundles at runtime, replacing the application's executing code without going through the app store. The Glass is the mobile screen; the Client Runtime is the application's JavaScript runtime; the supply chain is the

OTA service and whatever the bundle inherits from. ATG applies directly: the bundle is built by the over-the-air service at the moment it is fetched.

User-generated-content (UGC) interactive runtimes. Roblox, Unity's AssetBundle pipeline, and similar systems load user-generated assets into a runtime that integrates them into the running game. What an asset can carry differs by platform: Roblox assets carry Lua scripts directly, while Unity's manual records that AssetBundles cannot contain executable code and that the reachable path is altered serialized data exploiting the application or the runtime. The Glass is the gameplay screen; the runtime is the engine; the supply chain is whatever the user-generated asset depends on. The user trusts what the engine renders; the engine renders what the asset directs.

Productivity add-ins. Office.js add-ins inside Excel, Word, and PowerPoint; Jupyter notebook frontends; and the broader productivity-extension ecosystem each provide a Client Runtime inside an application typically not classified as a runtime host. The Glass is the document or notebook surface; the runtime is the embedded JavaScript or kernel-language runtime; the supply chain is the registry and CDN the add-in fetches from.

Spatial-asset auto-render. USDZ AR Quick Look on iOS renders a 3D scene from a URL the moment the user taps the link, with no step at which the user chooses to execute it as code. The Glass is the AR-rendered space; the runtime is the spatial-asset interpreter; the supply chain is the URL host serving the USDZ file. The structural property is that visual rendering is itself execution: the asset is interpreted, and interpretation produces what the user sees.

Plugin and extension marketplaces. IntelliJ Platform plugins, VSIX extensions for Visual Studio, Blender add-ons, and Anki add-ons each install code into a host application and run that code with the host's privileges. The Glass is the application UI; the runtime is the host's plugin runtime; the supply chain is the marketplace and the plugin's own dependencies. A compromised plugin executes inside the host with the host's trust.

IoT and control-system dashboards. The HMI of a smart-home appliance, the dashboard of an industrial control system, the screen of a medical device, and the touchscreen of a vehicle infotainment system are Client Runtimes running on devices typically not perceived as runtimes. The Glass in these cases sits in front of consequence: a mis-rendered alert, a deceptive status indicator, a manipulated control reading. The supply chains are smaller than the public web's, but ATG applies in the same structural form.

Each category is a Client Runtime rendering the Glass through a non-browser host. The full taxonomy spans languages, platforms, and providers. ATG is structural, not browser-bound.

One of those categories carries the highest concentration of function on the Glass anywhere, and it behaves differently from the rest. Industrial HMIs, embedded appliance interfaces, vehicle systems treated as controls rather than as media, building management panels, and medical device interfaces render a control layer rather than a reading surface. What appears on them is bound to actuation, timing, sequencing, interlocks, alarms, setpoints, and modes. The effect profile shifts accordingly. Disrupt, which is narrow against a page whose purpose is to be read, becomes the widest and most consequential effect where there is that much function to interfere with. Where a system carries real-time deadlines, the boundary between Degrade and Disrupt moves as well, because latency that reads as an annoyance in a browser is a missed deadline on a control loop, and a missed deadline is a functional failure rather than a slower success. Deny is harder to achieve against a fixed panel with little navigation, and more absolute when achieved, because there is no second device to try. Destroy has the shortest path of any surface class, because the Glass sits closest to the system it drives.

2.20: ATX: Beyond the Glass

Beyond the Glass, the architectural pattern that produces ATG generalizes. ATX (*Attack the X*) is the broader framework, where X is any human-computer interaction surface. The Glass is one X. Audio surfaces, haptic surfaces, voice interaction surfaces, gesture and sensor-driven interfaces, and emerging modalities such as brain-computer interfaces and robotic actuators are other instances.

ATX is forward-looking research framing. The current operational threat is overwhelmingly the Glass; non-Glass surfaces are emerging and operationally insignificant by comparison today. The enumeration below flags published attack vectors for future research, not for any expansion of operational scope.

Audio surfaces have received the most published research outside the Glass. Voice-assistant injection attacks place commands the human ear cannot detect into the system the assistant trusts: DolphinAttack and similar ultrasonic-injection research, SurfingAttack and other solid-surface conduction techniques, and commands hidden in music or background audio all sit here. Audio deepfakes drive executive-impersonation wire fraud, family-distress scams, and voice-authentication bypass. In-vehicle audio manipulation, smart-speaker hijacking, hearable-device exploitation, VoIP call modification, and streaming-audio injections round out the surface. The dominant D5 effect across the audio surface is Deceive, with Disrupt reachable through specific subvariants. Destroy is reachable where an injected command executes against a connected system; where a person is misled into acting, the effect is Deceive and the harm is its consequence.

Haptic surfaces are growing as wearables, controllers, and accessibility devices proliferate. Spoofed haptic confirmations falsify transaction success or biometric capture. Game-controller haptic manipulation deceives players in gambling and esports contexts. Wearable false notifications fabricate health alerts and trigger panic responses. Accessibility-device exploitation can misdirect or fail to alert visually impaired users. Steering-wheel haptic deception falsifies lane-drift or collision warnings or suppresses legitimate ones. Deceive dominates. Haptic misdirection that produces physical action is Deceive with a physical consequence rather than Destroy: a person acted, and nothing reached through the surface into another system.

Voice interaction surfaces overlap with audio but use voice as the input modality. Voice-driven UI manipulation hijacks kiosk, vehicle, and smart-home command flows. Voice authentication bypass combines deepfake audio with replay or injection. IVR-system manipulation targets phone-based voice-response systems through model compromise or middleware injection. Deceive is the dominant effect; Destroy appears at the consequence level via account takeover.

Gesture and sensor-driven interfaces bypass traditional UI inspection entirely. Motion-control deception manipulates Vision Pro, Wii, Kinect, and similar input through environmental injections or model compromise. Eye-tracking manipulation targets gaze-based authentication and AR/VR intent capture. Biometric spoofing covers fingerprint replay, face spoofing, and iris spoofing. Health-sensor data poisoning influences medical decisions, fitness records, insurance underwriting, and continuous-glucose-monitor readings. Environmental-sensor manipulation targets smart-home heating, security alarms, door unlocks, and HVAC controls. Deceive, Disrupt, and Degrade all appear. Physical harm that follows a person acting on manipulated sensor output is Deceive with a physical consequence; Destroy applies where the manipulation drives a connected system directly, as in environmental controls acting with no person in the loop.

Emerging surfaces are largely without confirmed operational attacks today. Brain-computer interfaces (Neuralink, Synchron, and the broader invasive and non-invasive BCI category) sit at the early-research stage; the theoretical D5 reach spans Deceive (false sensory input), Disrupt (interrupted intent translation), and Destroy (induced motor activity or harmful neural signaling). Robotic actuators with HCI (humanoid robots, collaborative robots, autonomous service robots) carry direct Destroy potential through compromised commands. Digital olfaction is nascent. The prospective attack delivers a false scent that misrepresents environmental safety or danger.

Across non-Glass ATX surfaces today, Deceive is the dominant achievable effect. Disrupt is second most common (jamming voice commands, blocking sensor inputs). Degrade is

present but rarer (degraded voice recognition through targeted poisoning, gradual sensor drift). Deny is present in authentication and emergency-notification contexts. Destroy is present where an injected instruction executes against a connected system with no person in the loop, as in voice commands triggering destructive actions and robotic actuator compromise. Account takeover through biometric bypass is Deceive against the authenticating system, and physical harm that follows a person acting on manipulated sensor output is Deceive with a physical consequence.

Malice Without Malware anticipates this surface expansion. The defensive vocabulary built for binary malware does not capture the audio, haptic, voice, gesture, sensor, or emerging-modality cases any better than it captures the Glass cases. The same architectural reasoning applies: each surface is a runtime that grants trust to upstream content, and each grant of trust without continuous re-verification creates the conditions for D5 effects.

ATX is the horizon. The Glass is the operational threat today. As non-Glass surfaces mature operationally, the same structural reasoning will extend to them.

Chapter 3: The Adversary: Capability and Operations

Vulnerabilities and adversaries are different problems. A vulnerability is a property of an architecture: it is there whether or not anyone uses it. An adversary is a person or group that decides to use it, picks a target, and picks a moment.

We characterize adversary capability across eight dimensions: scope of effect, persistence, pre-operational intelligence required, attribution difficulty, reversibility, resource investment, time horizon, and detection difficulty. The conventional single-axis ranking (script-kiddie at one end, nation-state at the other) collapses these into one scale and ties each point on the scale to the actor type historically associated with it. This single-axis approach conflates "what an adversary can technically do" with "what type of actor has historically done it." Rating an operation on each of the eight dimensions separately lets a defender describe what an adversary can do without first deciding what kind of adversary it is, which matters because the two no longer track each other. For example, a small, purely commercial team can hold persistence over long periods of time undetected without the resource investment and support traditionally only available to nation-state operations.

Two shifts mark the present moment. The first is the *commercialization gray zone*: the same technical capability is available to a broadening buyer base, and whether any given operation counts as legitimate depends on buyer identity and intent rather than on anything the technical analysis can reliably establish. Commercial spyware took that route over the last decade: capability that only governments had built became a product with a customer list. ATG-class techniques are on the same route now. The second is the *AI Amplification Factor*. An adversary can now have a model generate a different piece of manipulated content for every target, matched to that person's situation at that moment, and that changes what an ATG operation costs to run. When the AI runs on the user's device, part of the operational signature moves inside, where defenders cannot see it from outside. This is the *local-AI detection blind spot*. The combination of commercialization and AI amplification distinguishes the present moment from previous threat eras.

Adversary scope extends to the full Client Runtime landscape, not the browser alone. The adversary who works against a browser script dependency is the same adversary who works against a vehicle infotainment update channel, a smart-TV plugin marketplace, a productivity-tool extension store, or a mobile over-the-air update pipeline: code pulled straight from a server after the app is installed, without going back through the app store.

Placement, Access, Influence, Control describes how an adversary turns a capability into a running operation.

3.1: ATG Adversary Capability

Eight dimensions characterize ATG adversary capability. Any one operation sits somewhere on all eight at once; it does not get a single score.

Scope of effect: the size and type of population, infrastructure, or system an operation reaches. The range runs from a single targeted device or user, through a specific organization or supply-chain segment, to an entire infrastructure layer. Position on this dimension is determined by where in the digital supply chain the adversary holds placement and what passes through that position. A compromise of a major identity provider reaches every application that authenticates through it; a per-victim deception operation reaches one. Reach is not the same thing as targeting precision. An operation can pass through millions of devices while aiming at one person, so a large number here says nothing about how selective the operation was. Reach is also not the same thing as consequence. A single recipient can carry an entire operation when that recipient's decisions carry institutional weight: a head of government, a chief executive, or a commander directing a force each reads from one screen, and an operation that reaches that screen has reached everything it needed. A small number here therefore says nothing about what the operation was worth.

Persistence: the duration over which a capability remains operational once established. The range runs from transient (a single payload deployment that ends with delivery), through campaign-scoped (operational across a defined window of weeks or months), to indefinite (a position the adversary holds and re-uses across many operations over years). Persistence counts only the time after the position is working. Time horizon, a separate dimension, counts the whole arc from first planning to exit. Position on this dimension is determined by detection-and-removal pressure, the resilience of the underlying placement, and the adversary's discipline in operating quietly enough to remain in place.

Pre-operational intelligence: the volume and specificity of target-specific intelligence the operation requires before activation. The range runs from no target-specific intelligence (an operation that delivers the same payload to every reachable victim), through population-segment intelligence (deception calibrated to a demographic, language community, or device class), to individualized intelligence (deception assembled per target from aggregated data on that specific person). How much an operation needs depends on what it is trying to do and on the intelligence sources the adversary can draw on. AI makes the individually targeted end of the range much cheaper to reach.

Attribution difficulty: the analytical effort required to trace an operation back to the operator who launched it. The range runs from straightforward (clear technical signatures, named infrastructure, public attribution within weeks), through moderate (multi-month

investigation across several reporting institutions before attribution stabilizes), to structural (nothing is recording at the rendering surface, so the evidence that would identify the operator is never captured in the first place). Position on this dimension is determined by operational tradecraft, infrastructure compartmentalization, and what defenders can see at the layer where the operation runs.

Reversibility: the extent to which an operation's effects can be undone after detection. The range runs from fully reversible (a payload removed from a CDN ends the operation), through partially reversible (the immediate payload is removed but the data already exfiltrated, the inferences already drawn, or the decisions already taken cannot be reversed), to fundamentally irreversible (information delivered, beliefs shifted, financial transactions executed, or physical-world actions taken based on the manipulated surface). Position on this dimension is largely determined by the D5 effect being produced; manipulation effects sit toward the irreversible end of the range, because the manipulation has already affected the user's actions by the time the payload is removed.

Resource investment: the investment the adversary is required to make to acquire and operate the capability, across financial, technical, and operational categories. Financial cost runs from negligible (recruiting or co-opting an existing position-holder) to multi-million-dollar (commercial acquisition of a delivery-layer entity). Technical cost runs from modest (off-the-shelf tooling) to advanced (custom runtime instrumentation, AI-augmented payload generation). Operational cost runs from low (a single insider operating from inside an existing role) to high (a coordinated infrastructure of dormant assets staged across jurisdictions). An operation therefore carries three separate cost figures, and a low figure in one category says nothing about the other two. Intelligence cost is treated separately under Pre-operational intelligence required.

Time horizon: the total arc from planning and position acquisition through activation and exit. The range runs from short (weeks from compromise to activation, typical of opportunistic exploitation of a known vulnerability), through campaign-scale (months), to long-horizon (years between the acquisition of a placement and the activation that monetizes it). Time horizon includes the pre-operational positioning that Persistence does not measure: a long-horizon operation with transient persistence spends years staging a payload that runs once. Where an operation sits depends on how valuable the position is, how disciplined the operator is, and how well the operator can keep the position looking ordinary while waiting.

Detection difficulty: the level at which the operation can be detected, given the instrumentation that exists. The range runs from signature-detectable (the payload matches a known indicator and trips a static control), through behavioral-detectable (the

payload's effect on the runtime is anomalous and would be flagged by a monitor watching the right signal), to structurally undetectable (no instrumentation exists at the layer the operation runs at, so the signal that would trigger detection is never captured). Position on this dimension is largely determined by what the operation does and where it does it; an operation that runs entirely inside a Client Runtime on the user's device sits near the structurally undetectable end of the range with current instrumentation.

The dimensions are analytically distinct, not strictly independent. Where an operation sits on one dimension narrows where it can sit on the others without fixing it. What describes the operation is the whole set of eight positions.

Capability is not bound to actor type. Precision-targeted operations are most often run by nation-state actors, but the technical capability ceiling is reachable by any sufficiently resourced and patient adversary. Well-funded criminal organizations, hedge funds, large corporations, contracted private intelligence firms, and political operations all command the resources, the procurement pathways, and the discipline to acquire and operate placement at the levels associated historically with state actors. Motivations, risk calculus, objectives, and operational signatures differ across types of adversarial actors. The technical capability ceiling does not.

Resources are not the only thing that decides what an adversary can do. Where the adversary sits decides at least as much. An employee with relatively few personal resources, working in a high-leverage supply-chain position, can hold ATG capability that exceeds what most external adversaries could acquire through any combination of access points. A developer inside a widely deployed analytics SDK reaches every site that loads the SDK. A maintainer of a major CDN service reaches every dependent application that fetches from it. An engineer inside a common API library reaches every integration that calls it. A developer inside a mobile over-the-air update pipeline reaches every device on the channel. A maintainer of a vehicle infotainment update feed reaches every vehicle on that platform. The scale of insider-enabled capability is proportional to the organizational reach of the insider's employer.

A motivated external actor can target employment or personnel in a high-leverage position as a deliberate strategy for establishing placement. The insider category extends beyond direct employees to board members, consultants, advisors, contractors, and third-party vendors. Every position in the supply chain that holds placement and access is a potential line of infiltration.

ATG capability is not bound by traditional actor categories or resource requirements.

State capability in this class is not only inferred. The Great Cannon, documented by the Citizen Lab in 2015, sat alongside China's national filtering infrastructure and intercepted requests bound for servers hosting widely used analytics and advertising scripts. For a selected fraction of requesters, chosen by address, it dropped the request before it reached the intended host and answered with substituted JavaScript. Browsers outside China executed that code and were enlisted, without their operators' knowledge, in denial-of-service attacks against GreatFire.org and GitHub. Attribution rested on co-location with the filtering system across multiple network links, a shared side-channel indicating common code, and target selection consistent with the state's censorship objectives.

Two qualifications keep the case honest. Its placement route was on-path interception rather than compromise of a supply-chain component, so it is not evidence for the acquisition or maintainer routes. And the intercepted scripts were served over unencrypted HTTP; the researchers who documented the system named HTTPS as the remedy, and the operation as run in 2015 would not repeat unchanged against an encrypted path. What it establishes is the rest of the chain. Content substituted for the resource an application referenced, delivered to requesters selected by attribute, executing inside ordinary Client Runtimes at population scale, invisible to the user, to the site operator, and to the provider whose script was impersonated. The operation was run by a state, in public, in 2015, and it was documented at the time.

3.2: The Commercialization Gray Zone

The commercial spyware industry is the established precedent. Over the last decade and a half, tools that only governments had built became products that a widening set of private customers could buy. The Citizen Lab and Amnesty International produced the technical attribution work; the 2021 US Entity List action and the 2024 Pall Mall Process are the policy response. The trajectory from state-only capability to commercially-available-product is now well documented.

The same dynamic is in motion for ATG-relevant capabilities. Six commercial pathways put this capability within reach of buyers who are not governments.

Commercial CDN acquisition. Acquiring a content delivery network, or a domain or organization with established CDN customers, is a commercial transaction with no special security review. The 2024 Funnull and cdn.polyfill.io case is the example: a domain previously serving a widely used JavaScript polyfill service was acquired and subsequently used to deliver second-stage payloads to the dependent sites. The acquisition itself was the attack vector. The same pathway applies to specialty CDNs, regional CDNs, content edge networks, and the long tail of small but widely integrated delivery providers.

Behavioral-profiling-as-a-service. The ad-tech and digital-marketing industries operate continuously running infrastructure for collecting per-user behavioral telemetry, segmenting populations, and targeting content to individuals. The same infrastructure, viewed from the adversary's perspective, is pre-operational intelligence collection at scale across potentially large populations. Buying access to a behavioral-profiling provider, or to the data products derived from one, is a commercial transaction. Surveillance-industry data brokers do related work. Whether any given dataset is marketing data or pre-operational intelligence depends on the buyer.

Contracted private intelligence and influence operations. Private intelligence firms, opposition-research operations, and contracted influence-campaign operators sell capabilities that include the discovery, surveillance, and targeted manipulation of specific persons, organizations, or populations. The contracted operator's clients are the actors with the strategic interest; the operator carries the operational capability and the technical infrastructure on the clients' behalf. Several public cases over the last decade document this pattern.

Red-team and adversary-emulation services. The market for offensive-security tooling and contracted adversary emulation has matured into an industry, and several providers explicitly include supply-chain compromise and runtime-injection techniques in their playbooks. The legitimate use case is defensive testing under contract. The same tooling and the same operational tradecraft are available to anyone who can afford the contract or replicate the techniques.

Open-source ATG tooling. Public tooling now exists to assist with several stages of an ATG operation, from identifying vulnerable third-party scripts on a target site to generating manipulated content for delivery to specific user segments. The tooling carries no buyer screening. The barrier to entry is technical literacy, not capital.

Ad-tech and tag-management infrastructure. The infrastructure that delivers scripts (the advertising industry calls them tags) and configuration to billions of pages every day is built the same way an adversary would build infrastructure to deliver a second-stage payload. The risk is the ad network's own loader script, which runs in the page itself with the same powers as the site's own code, not the advertisement, which commonly renders in a separate frame and where malicious creatives are a long-recognized problem with established countermeasures. The integration patterns, the trust relationships, and the runtime behavior are all the same. Whether the operation is legitimate is a matter of who is operating the infrastructure and what payload is being delivered, neither of which is visible at the rendering surface. Targeted Delivery runs on the same rules that decide which advertisement a person sees.

Each of these vectors reaches across the Client Runtime landscape, not the browser alone. Mobile attribution and over-the-air update providers, vehicle infotainment update aggregators, IoT management dashboards, productivity-tool extension marketplaces, and AR/VR asset distributors are each following the same commercial pattern as the web script-delivery and CDN businesses. The same vectors that operate against websites operate against phones, vehicles, appliances, and headsets, and the same buyers can transact in those markets.

The Cost of Free is a related pattern. Many technologies positioned as "free" establish placement and access as their actual purpose. A free analytics library obtains visit history that is monetized as marketing data and that doubles as pre-operational intelligence on the populations it observes. A free open-source utility establishes placement in many systems where it is adopted, providing both an attack surface and an intelligence-collection channel. The "free" positioning is itself a placement strategy, and the resulting placement is a foundation for both pre-operational intelligence collection and direct exploitation. The placement a free component establishes can change hands with the component, including to gray-zone actors who monetize the access it carries.

Build, Buy, Influence, Coerce, and Exploit are five ways to end up holding a supply-chain position, each with a different cost and a different operational signature.

- **Build:** establish a technology services company organically and operate the position from inside it. High time horizon, high resource investment, low attribution risk because the position is genuinely the operator's.
- **Buy:** acquire an existing entity by commercial purchase, as in the Funnull and cdn.polyfill.io case. Quick relative to building; cost depends on the size and visibility of the target.
- **Influence:** acquire effective control of an entity through investment, board seats, partnership, joint venture, or analogous mechanisms. Less visible than acquisition; the entity's nominal ownership does not reflect operational control.
- **Coerce:** apply hostile pressure to an entity or its position-holders through extortion, blackmail, threats, or analogous tools. Compatible with operational discipline; the coerced position can be operated for years if the pressure is maintained.
- **Exploit:** break into the entity technically, using ordinary cyber intrusion tradecraft. Breaking in is only one of the five ways to end up holding the position.

The pathways are composable. A single position may be acquired through a combination: an exploitation foothold that is later converted into an influence position through hiring, or a financial acquisition followed by coercion of the existing personnel.

Supply-chain depth amplifies the leverage of any acquisition. Software A integrates library B, which depends on web service C; influence or control of C reaches A through the transitive dependency chain. The leverage is greatest when the chain passes assets through (data, media, code), because the deeper position can substitute, modify, or instrument those assets before they reach the rendering surface. Two conditions let a deep position be used this way. First, checking stops before running: integrity checks happen before delivery, and nothing checks the pieces again once the runtime starts assembling them. Second, the Just-in-Time App Blueprint: the runtime assembles whatever the chain delivers without re-establishing trust at the point where each piece joins. When an adversary holds the deeper position, those conditions enable transitive control.

The deeper a position sits in the chain, the more systems it reaches and the harder it is to see from the surface. A surface-side dependency review that ends at the first hop does not reveal what the third or fourth hop is doing.

The technical capability is the same regardless of who buys it or how the position was acquired. Whether an operation is legitimate depends on buyer identity and intent, neither of which the technical analysis can reliably establish.

Actor identity no longer predicts capability. Capability no longer predicts actor identity. The threat model that relied on either prediction was built for a buyer set that no longer exists.

3.3: ATG Operational Patterns

ATG operations draw from a broader range of supply-chain adversary tradecraft. Long-term infrastructure pre-positioning, trust-laundering through legitimate-looking infrastructure, supply-chain foot-holding via mergers and acquisitions, multi-step infrastructure resilience, public-cloud service abuse, geographic and jurisdictional staging, asset cycling, and operational compartmentalization are all in active use and all documented in the public supply-chain security literature.

Three of these patterns matter most for ATG, because they decide what a defender is able to detect and what a defender will miss.

Dormancy and activation timing. Dormancy and activation is the most important pattern. Adversaries acquire a position, operate it legitimately for an extended period that can run from months to years, and activate at the moment that maximizes operational value. During the dormancy phase the position is indistinguishable from a benign service, by design: traffic shapes, response patterns, deployment cadence, and update behavior all conform to the benign category the position belongs to. Activation is the only detectable event. The shape is the one the Two-Stage Attack has at the resource level, moved up to the

operation: everything a defender can inspect is benign, and what makes it an attack arrives afterwards.

Defensive postures built for active-attack signatures miss the dormancy window entirely, and the dormancy window is where the position is acquired, hardened, instrumented, and woven into legitimate dependency graphs. By the time the activation event arrives, the position is already trusted by every dependent runtime, every reviewing analyst, and every static integrity check the dependent applications carry. Detection at activation is not the same as defense; the activation often is the irreversible event. Defending against a dormancy-and-activation operation requires detecting the dormancy phase, which by construction looks benign.

The cdn.polyfill.io domain served a legitimate Polyfill library under several successive operators, then changed ownership and began delivering second-stage payloads. The XZ Utils contributor infiltration followed the same arc inside an open-source project: years of credibility-building, then activation.

Commercial acquisition as APA pathway. *Adversarial Placement and Access (APA)* names the case where the party holding a supply-chain position is hostile to the systems depending on it. In the cases where the buyer purchases a supply-chain position that feeds content into a Client Runtime, through an ordinary commercial transaction that raises no security alert. The Funnull and cdn.polyfill.io case is one instance of the broader commercialization-gray-zone pattern. The acquisition is the attack. No exploit is run; no credential is stolen; no perimeter is crossed. The pattern is not browser-specific. Mobile over-the-air update services, vehicle infotainment update aggregators, productivity-tool extension marketplaces, and IoT management providers can all be bought, and each of them feeds a Client Runtime somewhere in the platform landscape.

The security review run during a corporate acquisition rarely asks what the buyer will do with the position once they own it. Buyer-due-diligence frameworks for digital-infrastructure transactions are not built around the adversarial use of the position.

Insider and positional-access leverage at provider scale. An insider in a high-leverage position holds population-scale ATG capability at low resource investment. The pattern is different from external compromise because the access already exists. No exploit is needed; the operator was hired, contracted, advised onto the board, or installed through one of the other pathways the supply chain runs through.

Detection of the insider pattern requires monitoring the operator-side activity at the provider, and the provider's own internal monitoring is the only practical vantage point. External defenders see only the provider's externally observable behavior, which under the dormancy-and-activation pattern is benign by construction. The insider's operational

signature is internal; the externally visible signal arrives only at activation, and often only after the irreversible event has already happened.

3.4: The AI Amplification Factor

Generative AI does not create a new kind of weakness. It amplifies the one already at work: supply chain injection that lands at client-side rendering. The change is one of kind, not just of volume: it changes what an ATG operation costs the adversary to run, and it moves part of what the operation would leave visible to a layer existing defenses do not reach.

Four economic shifts mark the change.

- **Pre-operational intelligence cost falls.** Per-target precision becomes economically feasible at volumes the analyst-driven approach could not reach. A model can pull together information about one person from scattered sources (resume sites, LinkedIn profiles, social media content, public records, leaked datasets) and assemble a profile of that person at machine speed.
- **Content production cost approaches zero.** Each deception target can receive tailored content: language, vocabulary, voice, context, and media. Producing one more tailored deception costs no more than one more request to a model.
- **Detection-evasion improves.** Generated content is harder to signature-match because the per-target variation is built into the generation process. Static signatures, hash matches, and template heuristics all degrade against content that was assembled for one target and not seen anywhere else.
- **Operational scale increases.** Operations that previously required analyst time per target can now run at population scale. How many people an adversary can reach is now limited by what the compute costs, not by how many analysts they employ.

All four shifts come from one capability: multi-source target intelligence aggregation. The system ingests information about a target from disparate sources and optimizes message, voice, vocabulary, and context for the specific user, group, or demographic. The aggregation step previously required human analyst time and was the rate-limiting cost in precision-targeted operations. AI completes it at machine speed and at population scale, and the result is per-target deception that defender detection systems were not designed to handle.

The D5 effect Deceive is where the change bites hardest. Content built for one person at one moment is a different problem from swapping in one fake page that everyone sees: there is no shared copy for anyone to compare against. A defender inspecting one user's experience cannot reliably tell, even with full inspection authority, that the content arriving

on that user's screen was assembled to match that user's profile and would not have arrived in that form for another user. Information manipulation is the most likely and most dangerous ATG attack mode. AI amplification operates most directly on that mode and compounds every dimension of it.

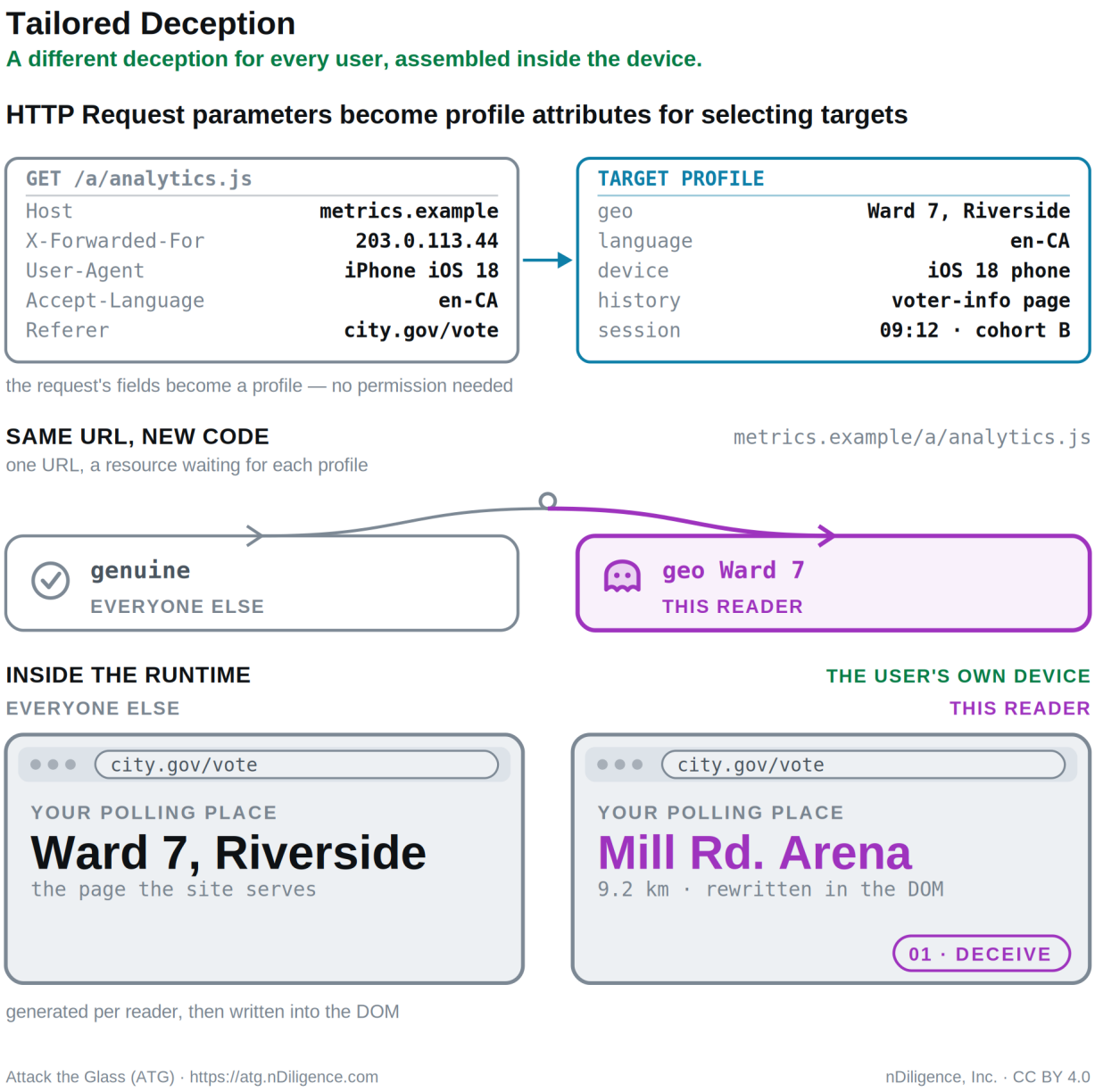

***Figure 22**: Per-target content leaves no shared copy for anyone to compare against.*

The Hong Kong synthetic-video conference fraud is another illustrative case. An employee authorized a major financial transfer after a video conference with what the employee believed to be the company's CFO. The video conference was synthetic. AI-generated per-context deception is operational reality today, not a forecast.

The local-AI detection blind spot is the second qualitative change. When the adversarial intelligence work happens on the target's device using the device's own integrated AI, there is no outbound network traffic to fingerprint. Detection systems built on network-level monitoring of suspicious calls to remote AI services miss the activity entirely. The operational signature moves inside the device, where defenders cannot see it from outside.

Local AI is reaching multiple Client Runtimes at once. Phones, laptops, vehicles, AR/VR headsets, smart appliances, and productivity tools are each gaining on-device AI capability through different pathways and on different schedules. The detection blind spot is therefore not a desktop-browser problem; it is reaching every Client Runtime gaining on-device AI.

AI capability is commercializing on the same terms: the tooling, the profile-building it performs, and the content it produces are all for sale, and a buyer's purpose is no more visible here than it is in the older ATG markets. The capability that produces tailored deception is in market alongside the placement that delivers it.

Detection systems built for static-payload signature matching will not scale to per-target AI-generated content, and detection systems built for network-level fingerprinting will miss the local-AI case. AI-augmented behavioral detection at the rendering layer is the architectural requirement. Current Client Runtimes do not provide it, and no instrument inside the runtime can see a model that is running on the device itself. The boundary is unprotected.

AI shifted ATG's adversary economics. Local AI moved the operational signature into the device. The defender's instruments predate both.

3.5: Placement, Access, Influence, and Control

Placement, Access, Influence, Control is the mechanism every ATG adversary operates through. Capability profile, actor type, and AI amplification shape how the mechanism is used; they do not replace it. P&A combines Placement and Access.

Placement is where the actor sits in the supply chain.

Access is what the actor can do from that position. A maintainer of an open-source package has Placement in the project; a maintainer who can publish releases has Access.

An OEM inside a vehicle infotainment update channel has Placement; an OEM whose engineers can ship updates has Access. Every ATG operation requires both.

Influence is what P&A produces. When P&A is exercised repeatedly and broadly, the systems, content, and decisions downstream of the position bend toward the operator's

intent. Influence is the cumulative effect of operating the position through repeated cycles of payload, observation, and adjustment.

Control is sustained Influence at population scale: when the operator's decisions effectively determine system behavior for the population, Influence has crossed into Control. There is no line where one becomes the other; the difference is how much of the population is affected and how consistently.

The causal chain is P&A → Influence → Control. Each stage is a quantitative escalation of the previous one. The chain is not specific to ATG; it operates in every supply chain. ATG is the chain at work at the rendering surface inside Client Runtimes. APA is when an actor in a P&A position acts against the systems that depend on the position.

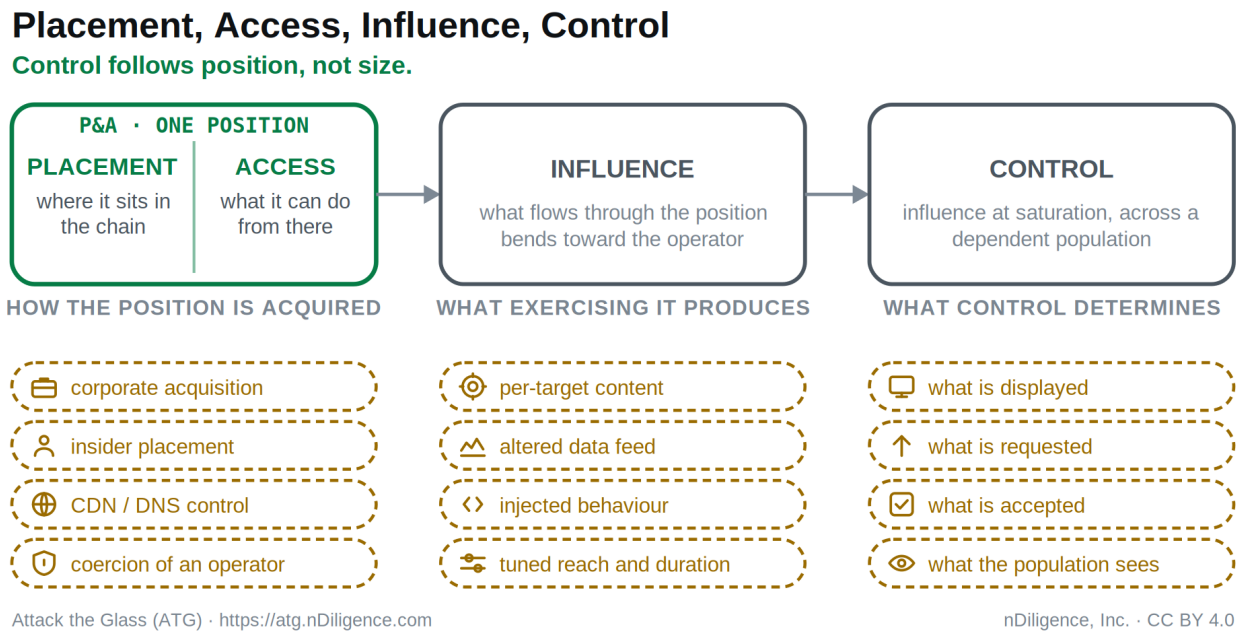

***Figure 23**: Placement and access produce influence. Sustained influence at population scale is control.*

Adversarial Influence Operation (AIO) is the activity by which an APA holder uses the position against those systems to create effects and outcomes.

Part II - The Surface Landscape

ATG does not operate from a single place. An ATG operation can start in any language the runtime executes, on any platform that runs one, through any provider that delivers content into one, and from any supply-chain position that touches those three.

Languages and Runtimes (Chapter 4): what running code is permitted to do, and how the powers a language grants decide what an adversary gains by taking a position that ships that code.

The Platform Landscape (Chapter 5): where the runtime lives, the device-scale picture, and the ten platform categories that span the deployed Client Runtime landscape.

The Provider Landscape (Chapter 6): who delivers content into the runtime, the five ways an adversary gets in, and the cross-provider patterns that recur across the deployed landscape.

Content Delivery Networks are where the most sites depend on the fewest providers, and buying one of those providers turns that concentration into a working attack.

Third-party script loading is the trust problem in its most direct form: a site tells the browser to run whatever a third party sends it, and Runtime Anarchy is what that looks like one script at a time.

Open source reaches the Client Runtime through the Open-Source Open Door, the acquisition route with the lowest barrier, and through Origin Laundering.

AI in the Client Runtime makes every one of these properties worse at the same time, and Tailored Deception is the result.

Placement and Access (P&A), the APA Vector Landscape, the seven acquisition pathways, and the attack model run across all of these surfaces.

Ownership Opacity and Adversarial Asset Cycling name the four things a defender would need to know about who owns a provider and currently cannot find out.

Chapter 4: Languages and Runtimes

Whatever an adversary places into a Client Runtime, the language hosting that runtime determines what the placement can do. Eval, dynamic loading, runtime compilation, in-memory state modification, network interception, privileged API access: these are not adversary techniques. They are properties of the language and its runtime, available to whatever code happens to be executing. An adversary who acquires placement inherits whatever the language permits.

Language, platform, and provider vary independently of one another. The platform is the runtime that hosts the language; the providers are the supply chain that delivers content into that runtime. Knowing which language an application is written in does not tell you which platforms host it; knowing the platform does not tell you which providers it loads content from; and knowing the providers does not tell you which languages their code and content are written in.

JavaScript in the browser is the loudest case because the web is the largest installed base that assembles what the user sees on the device itself (client-side rendering, or CSR), but the same analysis applies to Swift on iOS, Kotlin on Android, Lua in user-generated content (UGC) runtimes, Python in notebook and productivity front-ends, and the long tail of languages that build screens on the device in vehicles, appliances, control systems, and augmented and virtual reality headsets.

Seventeen language-level capabilities, split into two groups, describe what a language permits. Eight code-introduction capabilities govern how adversarial code or content arrives in the runtime; nine runtime-manipulation capabilities govern what arriving code can do once it is there. Which of these seventeen capabilities a language supports determines its level of risk.

A language's contribution to the ATG attack surface depends on its presence in user-experience surfaces across the deployed Client Runtime landscape, not only on what its runtime permits. Capability and reach together produce a spectrum of risk.

Browser-specific primitives that constrain a subset of the seventeen capabilities do not transfer as code to other Client Runtimes. Architectural patterns can translate even where implementing code cannot. Three patterns translate: SSoT (a single source of truth) for runtime resources, content-swap detection, and in-memory anti-manipulation.

4.1: Why the Language Matters

Of the three dimensions, language is the one that determines what is technically possible inside the runtime. The platform constrains where the runtime ships and how it is updated. The provider constrains what content arrives and from whom. The language constrains

what the runtime is allowed to do when content arrives and execution begins. Take a runtime that does not permit dynamic evaluation of code from a string: an adversary who places a string of code into that runtime cannot execute it. Take a runtime that does not permit modification of a peer's in-memory state: an adversary placed alongside a sensitive code path cannot reach into that path. The boundaries of the attack surface are language-level boundaries. Platforms and providers determine reach and delivery; the language determines what reach and delivery can produce.

The Manifest Illusion, Data as Code, and Code That Writes Code are language-level properties. The Manifest Illusion arises in a runtime to the extent that the language permits dynamic loading of code from sources determined at runtime. Data as Code exists to the extent that the language permits content (a string, an image, a configuration file, a manifest) to be coerced into executable behavior, whether through eval, embedded scripting, unsafe deserialization, or auto-fetched declarative content that interprets at load time. Code That Writes Code exists to the extent that the language permits running code to compile, evaluate, or otherwise produce further executable behavior at runtime. These are not abstract architectural patterns. They are conditions a language and its runtime either satisfy or do not satisfy, and a language that satisfies all three gives an adversary the full toolkit for ATG operations inside the runtime that hosts it.

That picture extends across every Client Runtime in scope, not the browser-hosted JavaScript runtime alone. Lua in a Roblox-style UGC runtime supports dynamic loading and runtime evaluation; an adversary who places a Lua module into that runtime inherits the same toolkit a JavaScript adversary inherits in a browser, calibrated to the Lua runtime's API surface. Hermes, the JavaScript engine inside React Native mobile applications, carries that same JavaScript model into those applications. The Python kernel inside a Jupyter notebook supports import-time and exec-time code introduction with full access to the host process. The renderer for USDZ, Apple's 3D scene format, interprets scene files whose declarative content carries behavior with it; on iOS that renderer is the AR Quick Look viewer. Each is a Client Runtime; each has its own capability profile; each is exploitable through the language-level conditions its runtime satisfies.

4.2: What Languages Allow

A language's contribution to the ATG attack surface is the set of capabilities its runtime makes available to running code. Seventeen capabilities, eight code-introduction and nine runtime-manipulation, cover the language-level conditions ATG operations require. The two groups are interdependent. Code-introduction determines whether an adversary can arrive. Runtime-manipulation determines what arriving code can do.

Code-introduction capabilities. The eight answer a single question: how does adversarial code or content arrive in the runtime in a form the runtime will execute? Each is a distinct mechanism by which a Client Runtime expands its executing code base after the initial load. Languages and their runtimes vary in which subset of the eight they support, and the more they support the larger the surface through which an adversary can introduce executable behavior.

- **eval:** dynamic evaluation of code from a string. The string can originate anywhere the language permits a string to come from, including a network response, a stored value, an attribute on a markup element, or a piece of user-generated content. JavaScript's eval and Function constructor are the examples in browsers and in JavaScript-engine-based mobile and desktop runtimes. Lua's loadstring plays the equivalent role in UGC runtimes such as Roblox. Python's exec and eval operate in Jupyter kernels, productivity tools, and any embedded Python interpreter. The capability is the most direct realization of Data as Code: any input the runtime evaluates becomes executable code.
- **dynamic load:** runtime loading of code modules from sources determined at runtime. The destination is decided after the application starts rather than fixed at build time. ES module dynamic import() in browsers, Node.js require() paths constructed at runtime, Hermes module loading inside React Native, Java reflection-driven class loading inside JVM plugin hosts, native dynamic library loading inside desktop applications, and loaders for over-the-air (OTA) bundles inside mobile applications, meaning code updates pulled straight from a server after install without going back through the app store, all instantiate the capability. The capability is the realization of the Manifest Illusion: the runtime fetches and executes code whose source and integrity are decided at fetch time, not at build time.
- **runtime compile:** compilation of source code on the fly into executable form. The runtime takes source text and produces in-process executable behavior without a separate build step. Browser JavaScript engines compile every script the runtime loads. Lua compiles loadstring inputs at call time. Python compiles imports and exec inputs at call time. WebAssembly runtimes that accept module bytes from arbitrary sources compile them on receipt. The capability is the upstream condition for Code That Writes Code: a runtime that can compile source at runtime can be made to produce further executable behavior from any string the running code generates or fetches.
- **embedded scripting:** the host language permits scripted execution inside data or markup that the runtime treats as content. HTML's <script> tag is the common example. SVG scripting inside vector graphics, MathML script execution, PDF JavaScript inside document viewers, and macro languages embedded in productivity-tool

documents all operate the same way. USDZ behavior nodes inside AR scene graphs are the declarative case: no script runs, and the interpreted content still directs what the scene does. The runtime parses content, the content carries scripts, and the runtime executes the scripts as part of rendering. The capability is why content channels are also code channels in many Client Runtimes.

- **unsafe deserialization:** deserialization formats whose parsing logic can produce code execution. Python pickle, Java serialization, .NET BinaryFormatter, YAML loaders that instantiate arbitrary classes, and a long list of model-distribution formats in AI runtimes all fall into the category. The runtime takes a serialized blob and reconstructs in-process objects whose construction can invoke arbitrary code. The capability turns the routine ingestion of saved state, model weights, document content, and configuration data into a code-execution channel.
- **plugin marketplace:** the language ecosystem includes a plugin or extension distribution channel that a runtime loads from. Browser extension stores, IDE plugin marketplaces, mobile app extension surfaces, smart-TV app stores, vehicle infotainment marketplaces, productivity-tool add-in galleries, and game-engine asset stores all expose the same structural pattern: third-party code distributed through a centralized channel and loaded into the runtime with the same permissions as the application's own code. One compromised publisher account therefore reaches every runtime that installed the plugin.
- **auto-fetched declarative content:** declarative content that the runtime auto-fetches and interprets at load time, where interpretation produces runtime behavior. CSS that fetches fonts, images, and additional stylesheets; web manifests; configuration files that specify dependencies to resolve; AI prompt templates that pull other prompts; action manifests for models that are allowed to act on their own; localization bundles that contain templated content. The boundary between data and code dissolves at the interpretation step. The capability is a quieter realization of Data as Code than eval: the runtime is not asked to execute a string, but the declarative content it interprets ends up directing executable behavior.
- **hot-reload or OTA update:** runtime code update without application restart. React Native CodePush, Expo OTA, Capacitor live updates, auto-update of the display bundles inside desktop application shells such as Electron, mobile app on-demand bundle delivery, vehicle infotainment over-the-air feature updates, and game-engine live patches all fit. The runtime accepts a new version of executable code and swaps it in while the application is running. The capability is why placement at an OTA pipeline reaches every device on the channel without re-deployment, and the reason mobile

and embedded Client Runtimes are not less exposed than browsers despite shipping precompiled binaries.

Runtime-manipulation capabilities. The nine answer a different question: once code is executing inside the runtime, what is it allowed to do? Each names a class of effect that running code can produce in the surrounding runtime, in other code's state, in the rendering pipeline, in the network layer, in storage, in privileged APIs, in background execution contexts, and in user input handling. A language's runtime-manipulation profile is what determines the D5 effects an adversary can produce once placement and code introduction have succeeded.

- **in-memory state modification:** modifying objects, prototypes, variables, or function bindings of other code in the running process. DOM manipulation in browsers, prototype pollution in JavaScript engines, monkey patching in Python and Ruby, native object swizzling in Objective-C and Swift, JVM bytecode manipulation in plugin hosts, and arbitrary state mutation in shared-memory runtimes are all instances. The capability is a primary lever for information manipulation: an adversary that can rewrite a DOM node, an answer returned by a model, or a rendered string can produce a deception that the user reads as legitimate output of the host application.
- **data injection between code paths:** one code path inserting data into another's flow. The injection target may be a function call, a message channel, an event handler, an inter-component message bus, an inter-frame communication channel, an inter-process pipe, or an embedded data-flow graph. Cross-component injection in single-page applications, postMessage abuse between iframes, intent injection in Android, URL-scheme handler abuse on iOS, message-port hijacking in WebViews, and inter-app sharing abuse on embedded platforms all operate through the capability. The capability enables information manipulation at the seam between trusted code paths, where each path treats the other's output as input it does not need to re-validate.
- **media manipulation:** modifying images, video, or audio in memory or before render. Canvas-level pixel manipulation, WebGL shader manipulation, decoded video frame editing, audio buffer rewriting, pre-display image processing in AR/VR rendering pipelines, and on-device synthesis of substitute media are instances. The capability sits at the operational center of on-device tailored deception.
- **rendering cycle control:** forcing or delaying repaint, forcing reload, manipulating the render loop, modifying the scene graph, intercepting layout passes, or rewriting the rendered output between layout and presentation. Browser requestAnimationFrame hijacking, scene-graph mutation in game engines and AR/VR runtimes, render-pass manipulation in mobile UI frameworks, and HUD-overlay manipulation in vehicle

infotainment all operate through the capability. The capability allows information manipulation to land at the moment of presentation rather than at the moment of computation.

- **network interception:** intercepting and modifying network requests and responses originating from the runtime. fetch and XHR hijacking through prototype patching, Service Worker interception in browsers, WebSocket manipulation, native HTTP-client interception in mobile and desktop runtimes, request-rewriting middleware in productivity-tool plugin hosts, and on-device proxy hooks in embedded environments are all instances. The capability turns the runtime into the man-in-the-middle for its own outbound traffic.
- **persistent storage manipulation:** reading and writing persistent storage that survives session restart. Cookies, localStorage, sessionStorage, IndexedDB, cache storage, native key-value stores, embedded SQLite databases, file-system access surfaces, and platform-specific secure storage all fall into the category. An adversary with persistent storage access can implant state that re-establishes presence on the next session, exfiltrate accumulated data on a scheduled cadence, or rewrite values that the application trusts on subsequent loads.
- **privileged runtime API access:** using sensitive runtime APIs that grant capability beyond rendering. Browser-side examples include the clipboard API, geolocation, the camera and microphone, notifications, the file system access API, push notifications, and payment APIs. Mobile and embedded Client Runtimes add device-specific surfaces: Bluetooth, NFC, SMS, contacts, calendar, telephony, biometric APIs, sensor and actuator APIs in IoT and vehicle Client Runtimes, and hardware-access APIs in control-system Client Runtimes. The capability transforms a foothold inside the runtime into capability that reaches into the physical world.
- **background execution context manipulation:** spawning or hijacking execution contexts that operate without user awareness. Web Workers, Service Workers, browser background timers, requestIdleCallback, native background tasks on iOS and Android, embedded scheduler tasks in vehicle and IoT runtimes, and headless contexts in productivity-tool plugin hosts are all instances. The capability decouples adversarial activity from user-facing rendering, which is operationally significant because most user-side detection signals are tied to user-facing rendering.
- **user input event control:** intercepting, suppressing, or fabricating user input events. Click, key, touch, gesture, sensor-input, form submission, voice, gaze, and controller input events all fit. The capability allows running code to perform actions on the user's behalf without the user originating them, and to suppress evidence that the actions

occurred. In spatial Client Runtimes (AR/VR, automotive HUD), gaze and gesture event control extends the capability into modalities where user awareness of the manipulation is lower than in keyboard-and-mouse interaction.

A language may have any subset of the seventeen, and where it sits across the full set is what shapes its risk. Together the capabilities cover the language-level conditions for the Manifest Illusion, Data as Code, and Code That Writes Code, and for the full D5 effects spectrum an injected resource can produce. Several runtime-manipulation capabilities (in-memory state modification, media manipulation, rendering-cycle control, data injection between code paths) are direct levers for information manipulation, the dominant ATG attack mode.

Several of the runtime-manipulation capabilities have established defenses inside the browser. The defenses are browser-specific.

4.3: User Experience Attack Surface by Language

A language scores against a second axis: whether it actually reaches user-experience surfaces in deployed Client Runtimes. A language with all seventeen capabilities but rare presence in user-facing rendering produces a small attack surface in practice. A language with fewer capabilities but global presence in user-experience surfaces produces a large one. A language's reach into user-experience surfaces is its User Experience Attack Surface.

User-experience presence runs along a four-step ramp: global, common, sometimes, rare. The ramp is scored across all Client Runtime platforms in scope, not the browser alone.

- **Global:** the language is dominant in user-facing rendering across many Client Runtime platforms. JavaScript, HTML, and CSS rate global. They appear across browsers, mobile WebViews and hybrid frameworks (React Native, Capacitor, Cordova), desktop application shells (Electron, Tauri), smart-TV runtimes, vehicle infotainment systems, smart-home interfaces, productivity-tool plugin hosts, and the long tail of platforms that embed a JavaScript engine for user-experience rendering. A global rating means a single language's capability profile reaches user-experience surfaces across the deployed landscape.
- **Common:** the language is dominant in user-facing rendering inside a particular Client Runtime platform category and absent or rare outside it. Swift and Objective-C rate common (global within iOS-class Client Runtimes, rare elsewhere). Kotlin and Java rate common (global within Android-class Client Runtimes, rare elsewhere). Lua rates common (global within several UGC runtimes and embedded scripting contexts, rare in mainstream consumer rendering). C# rates common (game engines and the .NET

application shell, less elsewhere). The common rating identifies a language whose user-experience reach is bounded by a platform category and is nonetheless very large within that category.

- **Sometimes:** the language appears in user-facing rendering in specific contexts but not across a platform category. Python rates sometimes: present in Jupyter notebooks, several productivity-tool front-ends, scientific-computing dashboards, and an increasing set of AI-tooling user interfaces, but not in mainstream consumer mobile, browser, or smart-TV rendering. USDZ rates sometimes: present in iOS AR Quick Look and adjacent AR contexts, not across general application surfaces. The sometimes rating identifies a language whose presence in user-experience surfaces is real and growing but bounded by use-case rather than by platform category.
- **Rare:** the language appears in user-facing rendering only in narrowly defined contexts. C and C++ rate rare in user-facing rendering despite their high capability profiles, because most user-experience rendering on top of C and C++ runtimes is delegated to languages further up the stack. Rust rates rare for the same reason in current deployment, with caveats for the emerging practice of compiling rendering code to WebAssembly. Assembly and bytecode languages typically rate rare. The rare rating does not mean the language is unimportant in the Client Runtime landscape; it means the language's presence in user-experience surfaces is narrow.

JavaScript sits at the high end of both: most of the seventeen capabilities and a global user-experience presence, which is the reason JavaScript is the dominant case. C and C++ sit at high capability but rare user-experience presence, and they contribute to the attack surface mostly indirectly, through the runtimes they host rather than through user-facing rendering they perform directly. Python sits at high capability but sometimes user-experience presence; the contribution is significant in the contexts where Python is the rendering language and limited elsewhere. The two-axis read is what matters, not a single-number ranking.

Risk is a spectrum across the two-axis read, not a binary of safe versus unsafe languages. True immunity in the CSR sense is rare, and where it exists it almost always means the runtime falls outside the ATG scope entirely: a precompiled-only Client Runtime that does not load runtime resources, a server-rendered static HTML page with no client-side scripting, a native compiled binary with no remote runtime loading, a terminal application with no Glass surface. Those runtimes are genuinely immune. A runtime that loads nothing at render sits outside the scope of the ATG vulnerability. What makes ATG consequential is not that immunity is unreachable but that reaching it means giving up runtime assembly, which is the capability that made CSR worth adopting. Within the boundary, low-risk

profiles exist (few capabilities present, low user-experience presence, common in restricted embedded UIs, some kiosk runtimes, and certain appliance dashboards), but they are positions at the low end of the same spectrum, not separate categories of safety.

4.4: Notable Language Profiles

Five language profiles show how the two-axis read produces operational analysis. Per-language details live in the Reference Matrix Languages sheet and the printable matrix sheets in the back matter.

Language	eval	dyn load	emb script	OTA	mem mod	net intercept	priv API	UX
JavaScript	Y	Y	Y	Y	Y	Y	Y	Global
Swift / Obj-C	L	Y	L	Y	Y	Y	Y	Common (iOS)
Kotlin / Java	L	Y	L	Y	Y	Y	Y	Common (Android)
Lua	Y	Y	Y	L	Y	L	L	Common (UGC)
Python	Y	Y	L	L	Y	Y	Y	Sometimes
USDZ	N	N	Y	Y	Y	N	L	Sometimes (AR)
C / C++	N	Y	N	N	Y	Y	Y	Rare

Y: capability present in normal runtime use. L: present in limited or restricted form. N: capability not present in normal use. Capability columns: eval, dynamic load, embedded scripting, hot-reload or OTA update, in-memory state modification, network interception, privileged runtime API access. UX: User Experience Attack Surface presence per the four-step ramp.

JavaScript sits at the high end of both axes. The language carries every code-introduction capability that matters in a Client Runtime (eval, dynamic load, runtime compile, embedded scripting, plugin marketplace, auto-fetched declarative content, and hot-reload or OTA update) and every runtime-manipulation capability (DOM and prototype manipulation, data injection, media manipulation, render-cycle control, network interception, persistent storage manipulation, privileged API access, background execution contexts, and user input event control). User-experience presence is global: browsers, hybrid mobile frameworks, desktop application shells, smart-TV runtimes, vehicle infotainment, smart-home interfaces, and productivity-tool plugin hosts all rely on a JavaScript engine for user-facing rendering. The combination is the reason JavaScript is the dominant ATG language profile and the reason most existing defensive primitives (Content Security Policy, Subresource Integrity, Trusted Types, the Permissions API) target it. The defensive coverage in the browser is real; outside the browser, the same JavaScript runtime engines (JavaScriptCore, Hermes, V8 in Electron, embedded V8 in vehicle and smart-TV systems) carry the capability set without the matching defensive primitives.

Swift and Objective-C sit at moderate code-introduction and high runtime-manipulation, with common user-experience presence inside iOS-class Client Runtimes. eval-style dynamic evaluation is restricted; dynamic load operates through framework-loading and plugin-host APIs; embedded scripting comes through WebView-hosted content rather than native script tags; hot-reload arrives through OTA bundle delivery channels. Once code executes, the runtime-manipulation surface is wide: native object swizzling, in-memory state modification, full network-stack interception, and access to the device's privileged APIs (camera, microphone, location, contacts, biometrics, sensor and actuator surfaces). The profile is illustrative of the native-mobile case: code-introduction is gated harder than in JavaScript, but runtime manipulation, privileged-API reach, and user-input-event control are at least as wide. Where the language exposes a JavaScript runtime through embedded WebViews or JavaScriptCore, the JavaScript profile applies inside that runtime.

Lua sits at high code-introduction inside the runtimes that host it and moderate runtime-manipulation, with common user-experience presence in UGC and embedded scripting contexts. eval (loadstring), runtime compile, dynamic load, and embedded scripting are all present. Hot-reload is platform-dependent. Runtime-manipulation is bounded by the host runtime's API surface: in a UGC runtime such as Roblox, Lua scripts can manipulate game-world state, intercept user input events, and modify the rendered scene, while network interception and privileged-API access are gated by the host platform's sandbox model. The profile is illustrative of the embedded-scripting case: code-introduction is wide because Lua is purpose-built as an embeddable scripting layer, while runtime manipulation depends on what the host runtime exposes. The UGC variant of the profile is operationally significant because the user-content channel is also the code-distribution channel; arbitrary Lua arrives through user-uploaded experiences and is executed at runtime by every user who joins.

Python sits at high code-introduction and high runtime-manipulation, with sometimes user-experience presence. The language carries eval, exec, dynamic import, runtime compilation, plugin marketplaces (PyPI plus IDE and notebook extension surfaces), and unsafe deserialization (pickle). Runtime-manipulation includes monkey patching, full network access through the host process, persistent storage access through file-system surfaces, and privileged-API access mediated by whatever the host environment exposes. User-experience presence is bounded: Jupyter notebooks, productivity-tool front-ends, scientific-computing dashboards, and an increasing set of AI-tooling user interfaces are the primary Client Runtimes where Python performs user-facing rendering. The profile is illustrative of the high-capability low-prevalence case: where Python is the rendering language, the contribution to the ATG attack surface is large; where it is not, the contribution arrives through the files the language produces (notebook outputs, generated

reports, files produced by AI pipelines) that are then rendered in another language's runtime.

USDZ sits at limited code-introduction, moderate runtime-manipulation, and sometimes user-experience presence in iOS AR Quick Look and adjacent AR contexts. USDZ carries no scripts. It carries a declarative behaviors schema of triggers and actions that the AR runtime interprets while it renders, so eval and dynamic load have no equivalent here. The asset still directs what the scene does: Data as Code without a script engine. Hot-reload arrives through asset replacement in the AR session. Runtime-manipulation is at the rendering layer: scene-graph mutation, media manipulation in the AR rendering pipeline, and rendering-cycle control. Network interception and privileged-API access are largely gated by the host AR runtime. The profile is illustrative of the spatial-runtime case: the capability set is narrower than JavaScript's but operates at the point of presentation, where the user has little basis for doubting what is shown; manipulation at the AR rendering layer reaches a user-experience surface where confidence in the rendered output is elevated by the immersive context.

JavaScript shows what a global presence on top of a wide capability set produces. Swift shows what gated code-introduction with full native-mobile runtime reach looks like. Lua shows what a purpose-built embedded scripting layer brings into the host runtime's capability frame. Python shows the high-capability, sometimes-presence case where the contribution is large in the contexts the language renders and indirect everywhere else. USDZ shows the spatial-runtime case where the surface is narrower and the presentation context elevates the consequence.

4.5: Defense Across Languages

Each browser-specific defense is built for the browser as a runtime. Content Security Policy works in a browser. It does not work in a Lua interpreter, a Python kernel, a USDZ scene-graph renderer, a Hermes runtime inside a React Native application, an embedded JavaScript engine inside a vehicle infotainment system, a Java or Kotlin runtime on Android, a Swift runtime on iOS, a C or C++ runtime in embedded firmware, a JVM language in a desktop plugin host, or a language runtime embedded in a smart-home hub, an IoT dashboard, or a productivity-tool extension host. Subresource Integrity works for browser script loading and for nothing else. The Permissions API gates sensitive APIs in browsers and has no equivalent elsewhere. Trusted Types constrains DOM injection sinks in browsers and applies only to the DOM. The browser-specific defense ecosystem is genuinely valuable inside the browser; outside the browser, none of the four can be lifted as code into another runtime.

Each of those defenses targets a capability that exists in every Client Runtime, not only in browsers. Network interception is a runtime-manipulation capability of every runtime that issues network requests. Privileged API access is a property of every runtime that exposes APIs to running code, and the privileged-API surface is wider in mobile, embedded, vehicle, and IoT Client Runtimes than in browsers. Static script integrity is something every runtime that loads scripts can pin, and most do not. DOM injection sinks are a browser-specific case of in-memory state modification, and other runtimes have analogous sinks in their own state-modification surfaces. The capability is the same in every runtime, but the defense is browser-only, and the gap widens as the capability set grows.

Three architectural patterns are reusable across Client Runtimes.

- **SSoT for runtime resources:** a service that maintains the reference for what a runtime should be loading (scripts, data, media, CSS, fonts, configuration, model weights, prompts, plugin manifests, and any other resource the runtime fetches and applies) and detects deviations from the reference at load or execution time. The SSoT pattern is implementable in any Client Runtime: in a browser as a runtime-resource manifest service that observes fetch and dynamic import; in a React Native application as a bundle-and-asset manifest service that observes the OTA update channel and the Hermes module loader; in a Roblox-style UGC runtime as an experience-asset manifest service that observes the script-loader and the asset cache; in a Jupyter kernel as a notebook-asset manifest service that observes import, exec, and pickle deserialization; in a vehicle infotainment system as a manifest service that observes the OTA pipeline and the runtime's resource cache. The implementing code is different in each runtime. The architecture is the same.
- **Content-swap detection:** the detection problem of identifying content arriving at the runtime that differs from the authorized content is shared across runtimes. Methods of detecting the swap (cryptographic comparison against an SSoT reference, behavioral profiling of expected content shape, anomaly detection at the load boundary, runtime monitoring of content as it is interpreted) are reusable as methods even when the code that implements each method is runtime-specific. The cross-runtime work is the method library; the per-runtime work is the integration into each runtime's resource-loading and resource-applying paths.
- **In-memory anti-manipulation:** detecting modifications to runtime state, to monitored objects, and to rendering pipelines is a pattern that applies across every runtime that exposes the runtime-manipulation capabilities. The steps carry across: declare which regions of state are monitored, instrument whatever loads or modifies those regions,

and detect departures from what was declared. The implementing code differs by runtime. The design is recognizable across them.

All three patterns are the language-level expression of Malice Without Malware. Defenses purpose-built for Client Runtimes have to come from the runtime itself: detecting manipulation, detecting deception, and detecting attempts to steer behavior. At the language level, the three patterns are what those defenses look like.

Chapter 5: The Platform Landscape

A language describes what running code is permitted to do. The platform describes what surrounds the running code: the channel that delivered it, the isolation around the runtime, the device APIs the runtime exposes, and the population of devices receiving the same delivery. A JavaScript engine carries the same language-level capability profile in a browser, in a desktop application shell such as Electron, in a smart television, and in a vehicle infotainment dashboard. The four ATG profiles those four hosts produce are not the same.

Platforms, languages, and providers vary independently of one another: a single platform category usually hosts several language runtimes, the same language usually appears on several platform categories, and knowing the platform or the languages does not tell you which providers deliver content into its runtimes.

ATG operates against Client Runtimes that assemble what the user sees on the device itself (client-side rendering, or CSR); Client Runtimes that run only code compiled before shipping are out of scope. Most modern platform deployments are CSR-capable, but the variation is real and runs inside almost every category. Mainstream consumer browsers, mainstream mobile phones, mainstream smart televisions, mainstream vehicle infotainment systems, mainstream IoT management dashboards, and mainstream productivity-tool extension hosts are all CSR-capable. Specialized deployments inside the same categories, including hardened embedded firmware, safety-critical avionics, and certain real-time control loops, may be precompiled-only and out of ATG scope.

Eight conditions describe what a platform contributes to ATG exposure. Each shapes how adversarial code arrives, what surrounds the running code once it executes, and what reach the runtime exposes beyond itself. Where a platform sits across the eight is what shapes its risk.

Conditions are the first axis. The second is device scale, treated as a blast-radius indicator: how many devices a successful platform-level compromise reaches, in order-of-magnitude terms rather than as a unique-device count. Billions of devices for the dominant browser-and-mobile categories. Hundreds of millions for desktop and game runtimes. Tens of millions and below for specialized surfaces.

Ten platform categories host Client Runtimes at meaningful scale, from browser and mobile at the top through desktop, productivity, and the non-browser surface categories.

Five patterns recur across the categories.

Implementing code does not translate from one platform to another. A verifier built for React Native CodePush, which delivers over-the-air (OTA) updates (code pulled straight from a server after install, without going back through the app store), does not run on a

vehicle infotainment update channel; a browser-shell Permissions API does not exist inside the renderer of a desktop application shell such as Electron, or inside a smart-TV application; nothing equivalent to CSP exists for a renderer of USDZ files, Apple's 3D scene format, or for a Lua runtime hosting user-generated content (UGC). Architectural patterns can translate even where implementing code cannot. SSoT (a single source of truth) for runtime resources, content-swap detection, and in-memory anti-manipulation apply at the platform level, with platform-specific implementation challenges added to each, and the defensive-maturity falloff outside the browser is the gap the patterns are designed to close.

5.1: Why the Platform Matters

Of the three dimensions, the platform is the one that determines where the runtime lives and what surrounds it. The language constrains what running code is allowed to do. The provider constrains what content arrives and from whom. The platform constrains the channel that delivered the content, the isolation around the running code, the device APIs the runtime can reach, and the population of devices a single delivery reaches.

Once an adversary has acquired placement on a Client Runtime, the platform's conditions determine what the placement reaches. An OTA bundle channel determines whether the placement reaches a single device or every device on the channel. Runtime isolation determines whether the placement is contained inside its declared scope or escapes into surrounding application and operating-system state. The device APIs determine whether the placement reaches the camera, the biometrics subsystem, the vehicle telemetry interface, or the industrial actuator surface. None of these are language properties; none are provider properties. They belong to the platform.

Every platform category contains deployments on both sides of the CSR-capable / precompiled-only line, and many individual devices contain both. The mainstream CSR-capable cases include consumer browsers, mobile phones running hybrid frameworks and OTA bundles, smart televisions, vehicle infotainment systems, IoT management dashboards, and productivity-tool extension hosts. On the precompiled-only side: hardened embedded firmware in some appliances, safety-critical avionics, and certain real-time control loops in industrial process equipment. The per-device split is itself a feature of several categories. A vehicle's infotainment runtime is CSR-capable; the safety-critical engine and brake electronic control units (ECUs) on the same vehicle are precompiled-only; ATG operates on the infotainment side. The same pattern recurs in IoT, where the management dashboard is CSR-capable and the underlying real-time control firmware is precompiled-only, and in control systems, where the human-machine interface is CSR-capable and the programmable logic controllers and remote terminal units behind it are precompiled-only. ATG scope follows the runtime, not the device. A runtime sits

outside that scope only where three conditions are all enforced: every runtime resource is pinned to its content rather than to its address or its certificate, no code is generated or evaluated at runtime, and no runtime-fetched data drives what is rendered or what executes. Meeting the first two and failing the third is the condition the Two-Stage Attack is built for, because the first-stage code passes every check made before delivery, and what gets swapped rides in the data. Across the browser, mobile, desktop-shell, appliance, productivity and user-generated-content categories, the CSR-capable case is the mainstream deployment. In IoT, vehicle and control-system categories it is the interface layer rather than the whole estate, and those interfaces are the ones most likely to pin their sources and least able to stop fetching the live data they exist to render.

5.2: What Makes a Platform an ATG Surface

A platform's contribution to the ATG attack surface is the set of conditions it imposes on the runtime it hosts. Eight conditions cover the ground ATG operations require beyond the language layer.

- **Update and distribution channel:** how new code reaches the platform: app store review channels (iOS App Store, Google Play, Microsoft Store, smart-TV app stores, vehicle infotainment marketplaces); OTA bundle delivery (CodePush, Capgo, Expo OTA, vehicle infotainment OTA, smart-TV firmware OTA); web auto-update (consumer browsers, SaaS clients); manual install (desktop installers, sideloaded APKs); and developer-mode sideload. Each channel is a Placement and Access (P&A) vector with its own review depth, signing requirements, publisher identity verification, and population of accepting devices. The same compromised package distributed through an app store with active human review reaches a different population, on a different timeline, than the same package distributed through OTA on a fleet-update channel.
- **Runtime isolation:** whether the platform sandboxes running code or runs it with broad system privilege. Browsers sandbox the renderer process; iOS and Android sandbox each app to its own address space; container runtimes isolate workloads. Desktop applications run with the user's privilege level and can reach the file system, network, and many platform APIs. IoT firmware commonly runs with full system privilege, and certain embedded human-machine interfaces have no isolation between rendering code and control-system access. In Electron, the split between the rendering process and the surrounding shell separates rendering from Node.js access to the machine, but that boundary holds only where the application is configured to keep the renderer away from Node.js, and it is frequently misconfigured. Isolation determines whether placement at the rendering layer can reach state belonging to the host application or the operating system around it.

- **Plugin and extension hosting:** whether the platform admits third-party plugins, extensions, add-ins, or asset packs as a first-class feature, and through which marketplace structure. Browser extension stores (Chrome Web Store, Firefox AMO, Edge Add-ons), IDE plugin marketplaces (JetBrains Plugin Repository, VSIX (the Visual Studio extension format), Eclipse Marketplace), Office Add-ins, smart-TV app stores, vehicle infotainment app stores, productivity-tool add-in galleries, and game-engine asset stores (Roblox catalog, Unity Asset Store) all instantiate the pattern. Review depth, publisher identity verification, and runtime constraints applied to plugin code vary widely across them. Plugin and extension hosting is a P&A vector by design: third-party code is admitted into the runtime as a first-class feature and runs with the host application's permissions.
- **Multi-language runtime hosting:** whether the platform hosts more than one language runtime in the same address space. Electron hosts JavaScript via V8 alongside Node.js native modules. Jupyter hosts Python, R, Julia, and additional languages through its kernel architecture. React Native hosts JavaScript in the Hermes engine alongside native code running through the host platform's Swift, Objective-C, Kotlin, or Java runtimes. Vehicle infotainment shells frequently host an embedded JavaScript engine alongside the native vehicle-API runtimes. Each hosted runtime brings its full language-level capability profile into the platform; the platform's ATG profile combines the capabilities of every hosted runtime and is not held down to those of the most restricted one.
- **Device API exposure:** which privileged APIs the platform exposes to running code. Browser-side: clipboard, geolocation, camera, microphone, notifications, file system access, push notifications, payment, and the broader Permissions API surface. Mobile-side: Bluetooth, NFC, SMS, contacts, biometrics, sensor APIs, telephony, and on certain platforms vehicle and industrial telemetry. Vehicle infotainment exposes vehicle telemetry, climate control, and increasingly cabin and seat actuators. IoT and control-system platforms expose the actuator surfaces that physically affect equipment in the world. The capability that converts a runtime foothold into physical-world reach lives here.
- **Persistent storage model:** what state survives across sessions and updates. Cookies, localStorage, sessionStorage, IndexedDB, and Cache Storage on the browser side. App sandbox file storage, key-value preference stores, secure enclaves, and embedded SQLite databases on mobile. File system access on desktop. Embedded device-side stores on IoT and vehicle platforms. Persistent storage is the channel through which a placement re-establishes itself after a session restart, accumulates exfiltrated data on a scheduled cadence, and rewrites values that the application trusts on the next load.

- **Update authority:** whether updates require user consent, are prompted but accepted by default, or are pushed silently. Mainstream consumer browsers auto-update in many configurations. App store updates often run silently in the background. OTA bundle delivery on mobile, vehicle, smart-TV, and IoT platforms is silent by default. The OTA model is the silent-update extreme: the runtime accepts a new bundle from the upstream service without any user-side decision. On a silent-update path nobody asks the user, so the user has no point at which to refuse.
- **Network model:** the platform's default network behavior and the controls that exist over outbound calls. Browsers enforce same-origin policy and CORS, and admit Content Security Policy and Permissions Policy when the application declares them. iOS enforces App Transport Security; Android exposes the network security config. Desktop applications, IoT firmware, and many embedded platforms have no analogous default constraints and admit outbound calls to arbitrary destinations. The network model determines what controls (if any) constrain a placement's ability to call home, exfiltrate data, fetch additional code, or contact a command-and-control endpoint.

Together the eight conditions cover where placement is admitted, what surrounds it once executing, what state survives between sessions, and what beyond-runtime reach the runtime exposes.

Several of the eight have meaningful defaults inside the browser; most other platform categories carry weaker defaults across the same conditions, and some carry no analog at all.

5.3: Device Scale and Platform Reach

A platform scores against a second axis: how many devices a single platform-level compromise reaches. A platform with severe conditions across a small device population produces a bounded total impact. A platform with milder conditions across billions of devices produces a large one.

Device scale runs from billions of devices at the top through hundreds of millions for the major application platforms and into the tens of millions and below for specialized surfaces. Browsers and mobile operating systems sit at the top, with Chromium-based engines, WebKit/Safari, iOS, Android, and the embedded browser engines that ship inside other categories together accounting for billions of devices in active use. Desktop application shells (Electron, Tauri, the JVM plugin hosts), the major game and interactive runtimes (Roblox, and the aggregate user populations across Unity and Unreal titles), the major productivity-tool extension hosts, and smart televisions in mainstream brands sit at hundreds of millions. Vehicle infotainment, IoT management dashboards in narrower deployments, control-system human-machine interfaces, AR/VR runtimes, narrower

productivity-tool surfaces such as Jupyter notebooks, and digital-signage networks sit in the tens of millions and below.

A single Android phone counts against the Android operating system, against the Chromium-based browser engine, and against any React Native or hybrid app installed on it that hosts its own JavaScript runtime. Device scale is a blast-radius indicator, not a unique-device count, and overlap across categories is expected. The comparative reading is the point: an OTA bundle delivery channel that touches a billion devices and an OTA bundle delivery channel that touches ten million devices represent different aggregate risk pictures even when the per-device exposure is identical. The Reference Matrix Platforms sheet captures the per-category reading in its Devices (Global) column.

Total impact is the conditions multiplied by the device-scale. Two platforms with identical profiles produce different total-impact pictures when their device populations differ by an order of magnitude, and two platforms with identical device populations produce different total-impact pictures when their profiles differ.

5.4: The Ten Platform Categories

Ten categories cover the platforms that host Client Runtimes in deployment today. Per-framework details live in the Reference Matrix Platforms sheet and the printable matrix sheets in the back matter.

Category	Iso	Plug	MLang	API	UA	Net	Scale
Browser	Y	Y	L	L	Y	Y	B
Mobile	Y	L	L	Y	Y	L	B
Desktop App Shell	L	Y	Y	Y	L	N	HM
Appliances	L	L	L	L	Y	N	HM
IoT	N	L	L	Y	Y	N	TM
Vehicle	L	L	L	Y	Y	N	TM
Control System	N	L	L	Y	L	N	TM
AR / VR / Spatial	L	L	L	Y	L	L	TM
Game / UGC	L	Y	L	L	Y	L	HM
Productivity / Data	L	Y	Y	L	L	L	HM

Y: condition present in normal platform configuration. L: present in limited or restricted form. N: not present in normal configuration. Iso: runtime isolation; Plug: plugin and extension hosting; MLang: multi-language runtime hosting; API: device API exposure; UA: silent update authority by default; Net: network controls comparable to same-origin,

CORS, CSP, and ATS. Scale: device scale band (B = billions; HM = hundreds of millions; TM = tens of millions and below).

Browser. The dominant Client Runtime host: the Chromium-based engines (Chrome, Edge, Brave, Opera, embedded WebView, embedded Electron renderer), the Firefox/Gecko family, and the WebKit/Safari family. The browser's profile combines strong runtime isolation (the renderer sandbox), first-class plugin hosting (extension stores), heavy auto-update with silent default, and the deepest existing defensive ecosystem of any platform category (CSP, SRI, the Permissions API, Trusted Types, COOP/COEP, the Origin model). Device scale sits at billions, and the user-visible per-device exposure runs from web pages through extension surfaces to the embedded-engine instances that ship inside other categories. The operationally significant ATG risk patterns are CDN-delivered third-party script compromise, browser-extension publisher takeover, and browser auto-update channel compromise reaching every browser instance on the channel. Mainstream browsers are uniformly CSR-capable; specialized stripped-down embedded browsers in some kiosk and appliance configurations approach precompiled-only behavior but rarely fully achieve it.

Mobile (Native, Hybrid, OTA Bundle Delivery). The iOS and Android families, plus the cross-platform frameworks (React Native, Flutter) and the hybrid WebView frameworks (Cordova/Capacitor) and the OTA-update vectors that ship with them (CodePush, Capgo, Appflow, Expo OTA). Mobile platforms combine strong app-level sandboxing with rich device-API exposure (camera, microphone, biometrics, location, sensors, payment, contacts, telephony) and silent OTA bundle delivery on the hybrid side. The OTA bundle channel is significant: it is silent-update at population scale, and it admits new JavaScript bundles into running applications without app-store review. Device scale sits at billions across Android and iOS. The operationally significant ATG risk patterns are OTA bundle compromise, WebView-hosted content compromise, and hybrid bundle replacement during an application session. Mainstream mobile deployments are CSR-capable through hybrid frameworks, WebViews, and OTA channels; pure-native applications without WebViews and without OTA approach precompiled-only behavior, but the combination is rare on modern phones.

Desktop App Shells (and Plugin Hosts). Electron, Tauri, JavaFX WebView, Qt WebEngine, and SWT Browser as application-shell runtimes; IntelliJ Platform, Eclipse RCP, VSIX, Blender, GIMP, Anki, Maya, Houdini, and FreeCAD as plugin hosts; Office Add-ins as the productivity-shell case. Desktop application shells host the full browser engine inside a non-browser shell, with the shell exposing OS-level APIs (file system, network, process control) that browsers do not. The profile combines first-class multi-language hosting (JavaScript via embedded engine plus native code via the shell's bridges), broad device-API

exposure (the host operating system), first-class plugin hosting, and weak defaults on the network model: Electron applications can call any host by default unless the application configures restrictions. Device scale sits at hundreds of millions, with Electron alone embedded in widely deployed IDEs, communication clients, password managers, and productivity software. The operationally significant ATG risk patterns are plugin-marketplace publisher compromise, Electron renderer-shell isolation failure where node integration is misconfigured, and supply-chain compromise of the embedded JavaScript engine or native modules. Universally CSR-capable through embedded browser engines and plugin runtimes.

Appliances (TV, Signage, Smart Home). Smart televisions (Tizen, webOS, Android TV, Roku, Fire OS), digital signage networks (BrightSign, Scala, NoviSign), and smart-home hubs (Google Home, Amazon Echo Show, the various SmartThings-class hubs). Many appliances run a full browser engine under the hood for their user-experience surface, often without the browser-class defensive ecosystem around it. Update authority is silent by default (firmware OTA at the network level, app-store auto-update at the application level), runtime isolation is uneven across vendors, and the device-API surface includes the appliance's hardware (camera, microphone, and lighting or HVAC controls in smart-home cases). Device scale sits at hundreds of millions for the major smart-TV families and lower for narrower categories such as digital signage. The operationally significant ATG risk patterns are connected-appliance silent update, app-store publisher compromise within the appliance ecosystem, and the inherited reach of the same Chromium-derived engine inside the appliance shell with weaker defenses around it. Older or low-end appliance firmware may be precompiled-only and out of scope; mainstream connected appliances are CSR-capable.

IoT (POS, Medical, Edge). Point-of-sale terminals, medical-device dashboards (patient-monitoring displays, infusion-pump operator interfaces, imaging-modality consoles), and edge computing nodes that expose management or operator interfaces. CSR-capable runtimes are commonly used for the management and configuration layer rather than for real-time control. Runtime isolation is often weak or absent (the management runtime executes with broad system privilege), update authority is silent OTA from the device vendor's management plane, and the device-API surface includes the actuator surfaces that physically affect equipment in the world. Device scale sits in the tens of millions and below across narrower deployments, but the per-device consequence runs into physical-world impact: payment data, medical-device behavior, edge-node control of downstream equipment. The operationally significant ATG risk patterns are management-dashboard compromise via the runtime layer, vendor OTA channel compromise reaching every device on the channel, and downstream actuator manipulation through a compromised

management interface. Some real-time control firmware within IoT devices is precompiled-only and out of ATG scope; the management dashboards and operator interfaces are typically CSR-capable.

Vehicle (Automotive Infotainment). In-vehicle infotainment systems (the Tesla infotainment UI, Ford SYNC, GM Infotainment, Mercedes MBUX, BMW iDrive, Volvo Polestar, the various Android Automotive deployments) and the OTA update channels that maintain them. Infotainment runtimes are CSR-capable through embedded browser engines and OTA bundle delivery; infotainment is kept away from the vehicle's safety-critical control bus by a gateway device sitting between the two networks, not by anything the runtime itself enforces. Device APIs include vehicle telemetry, climate control, and increasingly cabin and seat actuators, with vendor-specific gateway connections to driving-relevant subsystems. Device scale sits in the tens of millions and below, with growth as connected-vehicle penetration increases. The operationally significant ATG risk patterns are infotainment OTA compromise, supply-chain compromise of the infotainment runtime stack, and gateway-traversal attacks that reach beyond infotainment into safety-critical subsystems. The infotainment Client Runtime is CSR-capable; the safety-critical engine, brake, and steering ECUs are typically precompiled-only and out of ATG scope. ATG operates on the infotainment side; the gateway between infotainment and safety-critical systems is a structural concern in its own right, but it is not where ATG operates.

Control System (OT, SCADA, DCS). Operational-technology and industrial-control-system human-machine interfaces, or HMIs (Wonderware/AVEVA, Siemens WinCC, Rockwell FactoryTalk, GE iFIX, and the other vendor platforms), and the engineering workstations that maintain them. HMI runtimes are CSR-capable through embedded browser engines, scripting layers, and dashboard frameworks; runtime isolation between the HMI and the control-system network is operationally segmented but rarely enforced at the runtime layer. The device-API surface includes the actuator commands the HMI sends to the underlying PLC and RTU layer. Device scale is small in absolute terms, but the per-device consequence is the largest of any platform category: cascading control of physical infrastructure. The operationally significant ATG risk patterns are HMI compromise via the rendering or scripting layer, engineering-workstation compromise reaching the HMI deployment pipeline, and command injection through the HMI to the underlying control layer. The HMI layer is generally CSR-capable; the underlying PLC and RTU control layer is often precompiled-only. ATG operates at the HMI; downstream consequences traverse the HMI-to-control gateway.

AR, VR, and Spatial. AR Quick Look (USDZ on iOS), VR runtime engines (Meta Horizon, SteamVR, OpenXR-based runtimes), and spatial-computing platforms (Apple Vision Pro, Meta Quest, the various enterprise AR platforms). Spatial runtimes are CSR-capable

through asset auto-render (USDZ scenes loaded from a URL render with embedded behavior), runtime scene-graph manipulation, and asset-pack distribution. Device-API exposure includes cameras, depth sensors, eye and gesture tracking, and the spatial position of the user. The user-experience layer is distinctive: the immersive context elevates the user's confidence in rendered output, and gaze, gesture, and voice give the user fewer cues that something has been tampered with than a keyboard and mouse do. Device scale sits in the tens of millions and below, with growth on a longer time horizon than mobile or desktop. The operationally significant ATG risk patterns are USDZ AR Quick Look compromise (a tap on a link renders a malicious USDZ scene, and nothing asks the user to consent to executing it), VR or AR app-store publisher compromise, and asset-pack distribution channel compromise. Universally CSR-capable across mainstream deployments.

Game and Interactive Runtime (UGC). User-generated-content runtimes that load and execute scripts and assets supplied by other users: Roblox (Lua-based experience scripting), modded Minecraft servers, Unity AssetBundle delivery, Unreal Engine pak files, Garry's Mod and similar Source-based UGC, and the broader category of game engines that load and execute client-side content from a content server. UGC runtimes are CSR-capable by design: the runtime accepts user-supplied scripts and assets and executes them in the host application's address space. Runtime isolation between user-supplied scripts and the host engine varies across platforms, but supplied scripts run with the engine's API surface and reach. Update authority on the asset side is silent: a new asset version goes live the next time the runtime fetches it. Device scale sits at hundreds of millions across the major UGC runtimes. The operationally significant ATG risk patterns are UGC interactive runtime abuse (a script or asset distributed to a popular experience reaches every user who joins), platform-side asset-store publisher compromise, and engine-level supply-chain compromise that reaches every game built on the engine. Universally CSR-capable.

Productivity and Data Tools. Jupyter and similar notebook front-ends (JupyterLab, Google Colab, Databricks, VS Code Jupyter), Office.js add-ins (Excel, Word, Outlook, PowerPoint), browser extensions used in productivity workflows, and IDE plugin hosts (the desktop-shell plugin hosts re-appear here in their productivity application). Productivity runtimes are CSR-capable through extension surfaces, kernel-loaded notebook code, and add-in execution paths. Multi-language hosting is common: Jupyter hosts Python, R, Julia, and other kernels in the same notebook front-end; Office.js hosts JavaScript inside Excel, Word, Outlook, and PowerPoint; IDE plugin hosts admit Java, Kotlin, JavaScript, and native code into the same plugin runtime. Device-API exposure is moderate but the data access is high: notebooks routinely access sensitive data from corporate stores, AI services, cloud

APIs, and desktop file systems. Device scale sits at hundreds of millions for the major productivity surfaces (Office add-ins, browser extensions in productivity contexts) and lower for the narrower data-tool surfaces (Jupyter notebooks, scientific dashboards). The operationally significant ATG risk patterns are plugin-marketplace publisher compromise, notebook execution of malicious code from untrusted notebooks (the pickle execution path, and notebooks that run their stored code as soon as they are opened), and add-in supply-chain compromise that reaches every Office user with the affected add-in installed. Universally CSR-capable across mainstream productivity surfaces.

5.5: Cross-Platform Patterns

The underlying engines, channels, and trust models repeat across categories.

- **The embedded browser engine pattern:** a browser engine or browser-derived engine (Chromium, WebKit, Gecko) sits inside many non-browser shells: Electron in desktop applications, mobile WebViews, smart-TV runtimes, vehicle infotainment dashboards, IoT management interfaces, AR/VR display layers, and productivity-tool extension hosts. The engine carries its full language-level capability profile into every host. The browser-specific defensive ecosystem (CSP, SRI, Permissions API, Trusted Types) does not necessarily come with it; the host shell determines what defenses surround the engine, and most non-browser hosts apply weaker defaults than the browser does. The same Chromium engine inside Chrome and inside an Electron application is the same engine, but the surrounding defenses differ substantially. That is where defensive maturity falls off: the capability travels with the engine and the defenses do not.
- **OTA and silent-update as default:** OTA bundle delivery, silent firmware update, and live asset replacement are increasingly the default across platform categories: mobile hybrid frameworks ship CodePush and similar OTA channels by default, smart TVs auto-update firmware on a vendor schedule, vehicle infotainment systems update over the air, IoT management planes push firmware silently, and game runtimes hot-swap assets in active sessions. The silent-update model removes user consent from the security model. The convenience of silent update is operational; its security implication is that the population of devices any single push reaches is bounded only by the channel, not by user agreement.
- **Plugin and extension hosting as recurring trust models:** plugin marketplaces appear across desktop application shells, productivity surfaces, AR/VR platforms, smart-TV ecosystems, vehicle infotainment, and game runtimes. Each marketplace is a distinct trust model: review depth varies from active human review through automated scanning down to publisher self-attestation, meaning the publisher's own signed record of how the code was built and by whom; publisher identity verification varies; and

runtime constraints applied to plugin code vary. The structural pattern is the same across the categories: a centralized distribution channel admits third-party code into a host runtime, the code runs with the host's permissions, and the user trusts the marketplace by trusting the host. A compromise of any marketplace publisher account or the marketplace itself reaches every device that installs the affected extension.

- **Device-scale concentration vs physical-world consequence per device:** the browser and mobile categories together account for the overwhelming majority of Client Runtime devices in deployment. The vehicle, IoT, and control-system categories together account for the largest physical-world consequence per device. A billion-device population at moderate per-device consequence and a tens-of-millions population at the highest per-device consequence are different problems, and they call for different defensive priorities.
- **The defensive-maturity falloff outside the browser:** the browser-specific defensive ecosystem (CSP, SRI, the Permissions API, Trusted Types, COOP/COEP, the Origin model) constrains a meaningful subset of the language-level capabilities and the platform-level conditions inside the browser. Outside the browser, the analogs are largely absent. Mobile platforms have App Transport Security and the network security config; desktop application shells, appliances, IoT, vehicles, control systems, AR/VR, game runtimes, and most productivity surfaces have no comparable layer.

5.6: Defense Across Platforms

The browser's platform-level defenses are built around the browser's own process model. The renderer sandbox isolates the browser's renderer process; it has no analog in IoT firmware that runs management dashboards with broad system privilege, in a vehicle infotainment shell, or in a desktop application shell that exposes operating-system APIs to its renderer. The same-origin policy and CORS enforce boundaries between origins; there are no boundaries to enforce inside a Lua UGC runtime, an Electron application that calls any host by default, or an HMI runtime that talks to a control-system network.

Subresource Integrity pins the integrity of scripts the browser's fetch pipeline loads; OTA-delivered bundles in mobile, vehicle, and IoT systems have no equivalent integrity primitive at the runtime layer. The Permissions API gates sensitive APIs inside a permissions-grant model the host shell maintains; the broader device-API surfaces in mobile, vehicle, IoT, and control-system platforms have no comparable platform-level layer. Extension-store review constrains plugin admission inside browser ecosystems; the analogous marketplaces in desktop, productivity, AR/VR, smart-TV, vehicle, and game ecosystems run on weaker review depth.

The three patterns named in the language-level analysis apply at the platform level, with platform-specific implementation challenges added at each category.

- **SSoT for runtime resources:** each platform fetches and applies resources through platform-specific delivery channels: an OTA bundle channel on mobile and vehicle, a plugin marketplace on desktop and productivity, an asset CDN on game UGC, a firmware OTA on smart-TV and IoT, an extension store on browser. At the platform level the SSoT is a manifest service that observes the platform's resource-loading paths and reconciles fetched content against the reference. Implementation challenges are platform-specific: an OTA-delivered bundle's authenticity must be verified against the SSoT before runtime acceptance; a vehicle infotainment OTA channel needs its own source of truth for what a valid update is, held separately from the signed build record the OTA service supplies for itself; a plugin marketplace needs an SSoT reference for each published version that the host runtime checks at install and load time.
- **Content-swap detection:** at the platform level, content-swap detection has to integrate with each platform's distribution boundary. A plugin marketplace must enforce content-swap detection at the marketplace boundary so a compromised publisher account does not push divergent content to every installed instance. An IoT management dashboard needs a detector at the runtime boundary that catches an upstream-delivered update diverging from the deployed reference. A smart-TV firmware OTA channel needs a detector that reads the new firmware against the SSoT before runtime acceptance. The detection method is the cross-platform invariant; the integration into each platform's resource-loading and resource-applying paths is the per-platform work.
- **In-memory anti-manipulation:** at the platform level, the implementing code differs across categories. A vehicle infotainment runtime needs an anti-manipulation layer that operates inside the embedded browser engine, with the shell exposing the instrumentation it requires. A Roblox-style UGC runtime needs anti-manipulation that distinguishes the host engine's runtime state from the user-script runtime. An embedded browser engine in a non-browser shell needs a Permissions-API equivalent that the shell exposes for the shell's own non-browser APIs. The architectural pattern is one across the categories; the per-platform implementation work is what the pattern asks for.

The same three patterns answer Malice Without Malware at the platform level. Defenses purpose-built for client-side surfaces have to come from the runtime itself, not from upstream registry, build, or pre-delivery controls; at the platform level, that means applying the three architectural patterns to each platform category.

Chapter 6: The Provider Landscape

The runtime executes what providers send. Every Client Runtime in deployment runs content delivered by a population of supply-chain entities. Scripts come from CDNs. Analytics scripts come from tag managers, the hosted consoles that decide which of those scripts a page loads. Ads come from ad exchanges. Identity claims come from authentication providers. Model output comes from inference endpoints, the addresses an application calls to get an answer from a model. None of that content lived inside the application before it loaded. The runtime is built from what providers deliver, every time.

Modern applications depend on dozens to hundreds of providers across multiple categories. Each provider is a Placement and Access position. A compromise at the provider reaches every dependent application on every dependent runtime, every time those applications load.

Providers, languages, and platforms vary independently of one another: a single provider category usually delivers content into multiple language runtimes, the same provider usually feeds multiple platform categories, and language and platform together cannot predict which providers a given application depends on. A defender has to read all three together.

The provider dimension covers the full Client Runtime landscape. The familiar cases are web-targeting providers feeding browsers: CDNs serving JavaScript libraries, tag managers, ad networks, web fonts. The same provider categories deliver content into mobile applications, desktop shells that wrap a web runtime (Electron, Tauri), smart televisions and signage runtimes, vehicle infotainment systems, IoT and operational technology (industrial control) dashboards, AR/VR runtimes, productivity-tool extension hosts, and game and interactive runtimes that load user-generated content. A mobile attribution SDK, the code an app embeds to report which ad or campaign led to an install, is a provider. A vehicle infotainment map-tile service is a provider. A productivity-tool extension store is a provider. A foundation-model inference endpoint consumed by an Electron desktop client is a provider.

Six things shape a provider's risk, two of them about depth in different senses.

An adversary takes a provider over in one of five ways, each a different route to the same outcome. Which of the five is most likely varies from provider to provider.

The provider's shape and the ways an adversary gets in are two dimensions of risk. Reach is the third: how many Client Runtimes depend on this provider class, and how broad the population of dependent applications and devices is. Defendability runs alongside reach. SRI and CSP cover certain provider patterns inside browsers and have no functioning

analog at scale across most non-browser Client Runtimes, so a provider's defendability depends on which Client Runtime consumes its content.

Twenty provider categories cover the entities that deliver content into Client Runtimes at meaningful scale, ranging from code and package delivery and tag-and-analytics infrastructure through advertising and identity into AI services, plugin distribution, and the data-broker layer.

Eight patterns recur across the categories.

The code that implements a defense does not carry from one provider category to the next; the architectural pattern behind it does. The patterns that carry are a single source of truth (SSoT) for runtime resources, content-swap detection at the provider boundary, and provider-inventory discipline. Provider-inventory discipline carries an additional requirement: the inventory has to reach the dependencies of its dependencies, which is the lesson of the Log4j case, where exposure ran several dependency layers below what most organizations could see.

6.1: Why the Provider Matters

Providers are the dimension where the supply chain operates load by load. The runtime calls out, the provider responds, the runtime executes the response. Each call extends trust. Each response delivers content the runtime treats as authentic. Most of those calls go to organizations distinct from the application owner; some go to first-party infrastructure operated by the same organization. Either way, the application's own configuration named the destination, set the trust relationship, and admitted the content. Languages and platforms are also produced by organizations, and those organizations are themselves supply-chain entities at the level of the runtime. The operational supply chain inside any given application is the population of providers the application has configured to call out to and trust on every load.

An application that loads scripts from a CDN, more scripts from a tag manager, ads from an ad exchange, identity from an authentication provider, and inference from a model service has named five entities its runtime trusts on every load. Each is a Placement and Access position with a real address, a real owner, real personnel, and real means of being acquired or compromised. An adversary controlling any one of those providers determines the content every dependent runtime executes on the next call, and holds the power to segment those runtimes into a multitude of target groups: by region, by IP, by user-agent, by tenant, by hour of the day. The same machinery that delivers personalized content legitimately delivers selective compromise. Provider compromise is therefore a direct

operational task in adversary economics: identify the named entity, find the route in, and serve the desired content to the desired subset on the desired schedule.

Inside the runtime, the provider is a URL, a hash, a TLS certificate, a tag manager's container identifier, an inference endpoint, an extension manifest. The organizational structure behind the URL is invisible. The owner change last quarter is invisible. The maintainer's credential leak two months ago is invisible. The build-pipeline compromise yesterday is invisible. The runtime's trust extends to whatever the URL returns, on the implicit assumption that the URL maps to the same supply-chain reality it mapped to last load. Providers are where that assumption can drift, and an ATG operation is what runs in the gap between the supply-chain reality the URL mapped to last load and the one it maps to now.

6.2: What Makes a Provider an ATG Surface

A provider's risk has two parts. The first is the shape of the provider itself: what content it delivers, what defenses cover that content, how the runtime decides to trust it, how its updates flow, how widely it fans out, and how deep its own dependencies run. The second is how an adversary takes the provider over.

- **Injection vectors: which content types the provider delivers, and how each one carries content to the point where the runtime executes it.** Scripts (browsers, JavaScript runtimes inside Electron shells, mobile WebViews, productivity-tool extension hosts). Iframes (browsers and any embedded WebView). Media (images, video, audio that the rendering pipeline processes and where unsafe-deserialization paths exist). Styles and fonts (CSS that fetches additional resources, font foundries that ship hinting code). Manifest-driven auto-fetch (web app manifests, OTA update manifests, AI tool manifests). Declarative configuration (files the runtime reads at load time and acts on, which direct its behavior). Mobile over-the-air bundles: code updates an installed app pulls straight from a server, without going back through the app store (CodePush, Capgo, Expo OTA, vehicle infotainment OTA). Plugin manifests (browser extensions, IDE plugins, productivity-tool add-ins). Model weights (inference runtimes that load from a model hub). Prompt and tool definitions (AI agent runtimes that fetch templates and tool schemas at call time). Most providers carry more than one.
- **Defendability: whether the existing defenses built for client-side rendering cover the provider's content type.** SRI covers static script and style sub-resources fetched from a known URL with a known hash. CSP can constrain the set of URLs the runtime is permitted to fetch. Both are browser-specific defenses. Many provider patterns sit outside the coverage of either even inside browsers: tag managers that return different scripts to different requests, mobile attribution SDKs calling out from hybrid runtimes,

model endpoints returning fresh inference output, manifest-driven configuration auto-fetched at load. Across non-browser Client Runtimes (mobile WebViews outside the host-browser context, Electron and Tauri shells, smart-TV runtimes, vehicle infotainment systems, IoT dashboards, productivity-tool extension hosts, AR/VR runtimes, interactive runtimes that load user-generated content), SRI and CSP analogs are largely absent at scale. A provider's defendability depends on which Client Runtime consumes its content.

- **Trust model:** how the runtime decides to trust the provider's response. Pinned hash. Signed certificate (TLS, code signing, manifest signing). Domain reputation (the URL is on an allowlist or carries an organizational identity the runtime treats as authoritative). Contractual relationship (the provider is a third party the application owner has procured and tracks). No explicit trust check at all (the runtime fetches and executes the response without re-verification). Each model holds against the threats it was designed for, and not against an adversary operating inside the trust the model has already extended.
- **Update authority:** whether content updates from the provider require any explicit acceptance or run silently as part of normal operation. Web fetches happen on every page load; CDN-served libraries can change between page loads with no version pin. Tag managers change which scripts they serve from an administrative console, with no signal to the application. OTA bundles replace mobile application code on next launch. Plugin marketplaces push extension updates with no end-user prompt in many configurations. AI inference endpoints return fresh output on every call. Silent updates are the default for most providers.
- **Aggregation depth:** how widely the provider fans out across underlying sources. A CDN serving a library catalog hosts thousands of libraries from different upstream maintainers under one URL. A tag manager fires analytics, advertising, personalization, fraud-detection, and customer-engagement scripts through one container, a single hosted list of the scripts a page should load. A package registry holds millions of packages from millions of maintainer accounts. An AI-service hub aggregates models, adapters, and prompts from many publishers behind one endpoint. A compromise at the aggregator reaches every dependent application that loads any of its content, in one step.
- **Dependency chain depth:** how deep the provider's own dependencies run. The provider an application configured may itself fetch from a second provider, which depends on a third, which depends on a fourth. The Log4j case is the reference illustration: organizations were exposed to the Log4Shell vulnerability because the

affected Log4j library sat several dependency layers below the components they directly depended on, often in software they had not previously known they were running. Aggregation depth runs across one layer; dependency chain depth runs through the layers themselves.

There are five ways an adversary gets in.

- **Account takeover:** the provider's credentials or publishing access are compromised, and the adversary uses the legitimate publishing channel to deliver adversarial content. The credentials may be stolen, phished, leaked through a third-party breach, or harvested from a maintainer's personal infrastructure. The runtime sees a routine update from a trusted provider. The provider sees a routine publishing event from a credentialed account.
- **Network-path hijack:** the network path between provider and runtime is intercepted, and adversarial content is substituted in transit. BGP hijack, DNS poisoning, certificate-authority compromise, and on-path interception in network segments under adversary control all do this. TLS narrowed but did not eliminate the surface; certificate-authority and DNS-level compromise remain in scope, and many non-browser Client Runtimes ship with weaker default certificate pinning than the major browsers. The narrowing is smaller than it appears, because transport encryption protects the path and not its ends, and modern delivery arranges for a third party to hold one of the ends. A content delivery network terminates TLS at its edge: the edge decrypts the resource, holds it in plaintext, and re-encrypts it for the client. That is an interception position, occupied lawfully, established by contract, and invisible to the user and frequently to the application owner as well. Transport encryption does not eliminate the intermediary position. It relocates that position from any party able to reach the wire to the delivery provider the application owner has contracted with, inside the trust boundary rather than outside it. Whoever holds that position holds the plaintext, and the legal regimes described in the URL Overtrust analysis reach it there: national intercept regimes place their obligations on whoever holds the traffic in readable form, which for CDN-delivered content is the delivery provider.
- **Ownership change:** the provider is acquired or its ownership changes, and legitimate new owners use the legitimate publishing channel to deliver adversarial content. The Polyfill.io case (2024) is a real-world example: a long-trusted CDN domain was acquired by Funnull, a China-linked CDN operator (later sanctioned by the U.S. Treasury's OFAC in 2025 for facilitating large-scale scam and money-laundering infrastructure; see Silent Push, Triad Nexus), and the new ownership used the legitimate publishing channel to deliver selectively-targeted malicious content to a

population of dependent sites. Acquisition leaves the publishing channel intact; the adversary inherits trust along with ownership.

- **Insider compromise:** the provider's personnel are the threat actor. Insiders may be recruited adversaries placed deliberately, existing employees turned through coercion or financial inducement, or contractor personnel with legitimate access to publishing channels. Provider organizations vary widely in personnel vetting, in access controls on supply-chain-critical roles, and in monitoring of routine insider actions, and many lack threat-modeling that anticipates nation-state or organized-crime placement.
- **Cyber exploitation:** technical compromise of the provider's infrastructure, code, build pipeline, or dependencies through exploitation of vulnerabilities. The compromised provider continues to operate normally from its own perspective; the adversary's foothold sits inside the provider's systems and uses the provider's infrastructure to publish adversarial content. Cyber exploitation differs from account takeover (which uses legitimate credentials obtained illegitimately) and from ownership change (which uses legitimate ownership).

6.3: Provider Reach and Defendability

Provider reach is measured in dependent applications, not in user counts or transaction volumes. A CDN serving a hundred thousand sites reaches a hundred thousand dependent applications. A tag manager deployed across millions of properties (the websites and apps that embed it) reaches millions of deployed containers. An open-source package depended on by half the JavaScript ecosystem reaches every project in the transitive dependency graph that includes it. A mobile attribution SDK installed in tens of thousands of apps reaches the installed-app population and its active users. An AI-service endpoint consumed by thousands of products reaches the products that call it. User counts and transaction volumes follow from the dependent applications.

Attack routes and reach interact. A high-reach provider most often compromised through account takeover needs different defenses than a high-reach provider most often compromised through ownership change or cyber exploitation. The Reference Matrix Providers sheet scores each provider category by reach and by which attack routes apply, so a defender can read the per-category combination directly.

Defendability runs alongside reach. SRI and CSP are browser-specific. A provider that delivers content into a browser may be partially covered by SRI for its static script sub-resources; the same provider delivering the same content into a mobile WebView outside its host browser context, an Electron or Tauri shell, a smart-TV runtime, a vehicle infotainment system, an IoT dashboard, a productivity-tool extension host, or an AR/VR

runtime is generally uncovered. Reading defendability for a provider therefore requires knowing the Client Runtime platforms it feeds. What holds inside browsers does not hold across the rest of the deployed Client Runtime landscape.

6.4: The Twenty Provider Categories

Per-provider details live in the Reference Matrix Providers sheet and the printable matrix sheets in the back matter.

Category	Agg	DCD	Def	ATO	NPH	OC	IC	CE	R
Code and Package Delivery	Y	Y	L	Y	Y	Y	Y	Y	HM
Tagging and Analytics	Y	Y	N	Y	L	L	Y	Y	HM
Advertising	Y	Y	N	Y	L	L	Y	Y	B
Embedded Content	Y	L	L	Y	L	Y	L	L	B
Style, Font, and Icon Delivery	L	L	Y	Y	Y	Y	L	L	B
Trust Widgets	L	L	N	Y	L	Y	Y	Y	HM
Identity and Authentication	L	L	L	Y	Y	Y	Y	Y	HM
Payments and Commerce	L	L	L	Y	Y	L	Y	Y	HM
Maps and Reference Data	L	L	N	Y	Y	Y	L	Y	HM
Email and Messaging	L	L	N	Y	L	L	Y	Y	HM
Marketing and CRM	L	L	N	Y	L	L	Y	Y	HM
AI and ML Services	Y	Y	N	Y	Y	Y	Y	Y	HM
Customer Engagement	L	L	N	Y	L	Y	Y	Y	HM
Data Brokers and Profiling	L	L	N	Y	Y	Y	Y	Y	HM
Translation and Localization	L	L	L	Y	L	L	Y	L	M
Sanctions, Watchlist, and Compliance Data	L	L	N	Y	Y	L	Y	Y	M
Tracking and Attribution	L	L	N	Y	L	Y	Y	Y	HM
Cloud Storage and Asset Hosting	Y	L	L	Y	Y	Y	Y	Y	B
Plugin and Extension Distribution	Y	Y	L	Y	L	Y	Y	Y	HM
AI-Generated Content Services	Y	Y	N	Y	Y	Y	Y	Y	HM

Y: routinely applies. L: applies in limited or specific cases. N: rarely or not applicable. Agg: aggregation depth. DCD: dependency chain depth. Def: defendability inside the browser via SRI and CSP coverage. ATO: account takeover. NPH: network-path hijack. OC: ownership change. IC: insider compromise. CE: cyber exploitation. R: reach band (B = billions of dependent applications and devices; HM = hundreds of millions; TM = tens of millions; M = millions and below).

Code and Package Delivery. CDNs serving JavaScript libraries, package registries hosting open-source modules, and module-host services that distribute application dependencies. The category feeds browsers, mobile WebViews, JavaScript runtimes inside Electron and Tauri shells, and the embedded JavaScript engines that ship inside smart-TV apps, vehicle infotainment systems, and other Client Runtimes that load JavaScript at

runtime. Aggregation depth and dependency chain depth are both deep: a CDN serves thousands of libraries; each library has its own dependency graph. Ownership change is the concern (the Polyfill.io case); account takeover and cyber exploitation apply broadly across registry and CDN compromises.

Tagging and Analytics. Tag managers, analytics platforms, session-replay services, and the tag-distribution channels that fire scripts on page load. The category is primarily browser-and-mobile but reaches into productivity-tool surfaces and customer-engagement embeds wherever tags are deployed. The tag manager itself is an aggregator: dozens of vendor scripts can fire through one container, often with no review on the application side before they load. Account takeover, insider compromise, and cyber exploitation are the most common routes; ownership change occurs when a tag-publisher company is acquired and the resulting tag updates serve adversarial content.

Advertising. Ad networks, exchanges, and the demand-side and supply-side platforms that buy and sell advertising space automatically while a page is loading. The category reaches into browsers, mobile applications, smart-TV apps, in-app advertising surfaces, and connected-TV inventory. Ads are a long-running ATG concern (Magecart-class skimmer operations have used ad-tech infrastructure repeatedly), and the chain of trust between exchange, network, and ad is opaque to the consuming page. Account takeover, insider compromise, and cyber exploitation dominate; the ad-tech pipeline is itself an aggregator with many handoff points where adversarial content can enter.

Embedded Content. Social-media embeds, content embedded by pasting its URL (oEmbed and similar), video and audio embeds, and the iframe-based content channels that bring third-party-rendered surfaces inside the application page. The category is cross-platform: browsers, mobile WebViews, productivity-tool surfaces, and any Client Runtime that supports iframe-class embedding. The risk pattern is iframe content rendering with cookies, scripts, and access to embed-frame state that the host page cannot inspect. Account takeover at the embedded provider and ownership change of the embedding platform are the dominant routes.

Style, Font, and Icon Delivery. Web fonts, icon sets, and the design-system distribution surfaces that ship style assets. The category reaches into browsers, mobile WebViews, Electron and Tauri shells, smart-TV apps, productivity tools, and any Client Runtime that loads styles or fonts from a third-party host. Fonts can ship hinting code (executable instructions) and CSS can fetch additional resources at parse time, so the category is not purely passive. Network-path hijack and ownership change are the most operationally distinctive routes; account takeover applies broadly.

Trust Widgets. Chat widgets, comment systems, review-and-rating widgets, reCAPTCHA-class anti-abuse widgets, and the cross-platform reputational components that ship as embedded JavaScript and iframes. The category reaches across browsers, mobile applications, and any Client Runtime that admits widget-style integration. Trust widgets carry application-level trust by their nature (reCAPTCHA-class widgets are integrated as security primitives, not as decorative content), and the visibility-trust gap is wide. Account takeover, ownership change, insider compromise, and cyber exploitation all apply.

Identity and Authentication. Identity providers, SSO services, social-login providers, and the MFA service hosts that broker authentication into applications. The category is cross-platform and reaches into mobile shells, desktop applications, productivity tools, and increasingly into AI systems that are allowed to act on their own and that consume identity claims to call other services. Compromise of an identity provider produces direct authentication-bypass and impersonation effects that downstream applications cannot independently verify; the runtime accepts the claim because the protocol said the claim was valid. Account takeover at the operator level, network-path hijack on the auth flow, and insider compromise are the most operationally significant routes.

Payments and Commerce. Payment processors, buy-now-pay-later services, tokenization services, and the embedded checkout components that handle financial-transaction surfaces. The category reaches across browsers, mobile applications, in-store terminal interfaces, and any Client Runtime that integrates a payment surface. Magecart-class skimming operations have repeatedly compromised the payment-component layer; the surface attracts persistent, well-resourced adversary attention. Account takeover, network-path hijack, insider compromise, and cyber exploitation all apply.

Maps and Reference Data. Map-tile providers, geocoding services, weather data services, and the financial market-data feeds that ship reference data into runtime decision logic. The category reaches into browsers, mobile applications, vehicle infotainment systems, navigation surfaces in commercial fleet platforms, and the AR/VR runtimes that consume spatial reference data. Reference data is data-as-code in a particular sense: applications act on the returned values, and adversarial values produce adversarial behavior in the calling application. Account takeover, network-path hijack, and ownership change are the dominant routes.

Email and Messaging. Transactional email services, push-notification gateways, and SMS service providers that ship messaging primitives into application runtimes. The category is cross-platform across browsers, mobile applications, and the desktop and embedded surfaces that integrate notification delivery. The runtime exposure is at the SDK and template-rendering layer; messages can carry rendered HTML, scripts, and attachments

that the consuming application processes. Account takeover, insider compromise, and cyber exploitation apply most directly.

Marketing and CRM. Marketing-automation platforms, CRM SDKs, and the customer-data-platform integrations that deliver script-and-API surfaces to applications. The category reaches across browsers, mobile applications, and any Client Runtime that integrates a marketing-automation tag. The risk pattern is wide deployment combined with high update cadence: marketing-automation rules and tag content change frequently and silently. Account takeover and insider compromise are the dominant routes.

AI and ML Services. Foundation-model API providers, vector-database services, embedding-service hosts, model-hub publishers, and the channels that distribute adapters, the small add-on files (LoRA and similar) that adjust how a model behaves, into the code paths that run inference. The category reaches across browsers, mobile applications, desktop shells, productivity tools, and increasingly into local-AI deployments inside the device itself. The category is structurally different from earlier categories because the runtime cannot validate inference output against intent, and the response is data the application acts on. Account takeover at the model-hub level, ownership change of a publisher, network-path hijack on the inference call, and cyber exploitation of the API or the model file all apply, and insider compromise inside the model-hub operator is a sustained concern.

Customer Engagement. Live-chat services, support widgets, scheduling and document-signing tools, and the customer-facing operational surfaces that integrate as iframe and script components. The category is cross-platform and reaches into browsers, mobile applications, productivity tools, and the embedded support surfaces inside enterprise SaaS clients. Customer-engagement components frequently run with elevated trust because the application has procured them as core operational tools. Account takeover, ownership change, and insider compromise are the most applicable routes.

Data Brokers and Profiling Services. Data-broker APIs, profiling services, and the third-party-data integrations that feed personalization and risk-scoring decisions into the application. The category is API-facing rather than script-facing and reaches into browsers, mobile applications, and the back-end-driven surfaces that personalize displayed content. Adversarial data returned by a broker can drive adversarial application behavior at the rendering layer; the runtime sees a routine API response and acts on the values. Network-path hijack, account takeover, and insider compromise are the dominant routes.

Translation and Localization. Localization platforms, translation API services, and the localized-string and locale-bundle distribution channels that ship language-specific content into applications. The category reaches across browsers, mobile applications,

desktop shells, and any Client Runtime that loads localized content at runtime. Localization bundles can carry templated content that interprets at runtime, making the category a quieter Data-as-Code surface than scripts but no less consequential when compromised. Account takeover and insider compromise are the most common routes.

Sanctions, Watchlist, and Compliance Data. Sanctions-list providers, watchlist services, and the compliance-data feeds that drive regulatory and risk-control decisions inside applications. The category is API-facing and feeds browsers, mobile applications, and the back-end-driven compliance surfaces inside financial, healthcare, and government systems. Adversarial compliance data has direct downstream effects: a bad sanctions hit creates regulatory exposure; a missed sanctions hit creates regulatory exposure of a different kind. Network-path hijack, account takeover, and insider compromise are the dominant routes.

Tracking and Attribution. Mobile attribution SDKs, affiliate tracking services, and the cross-app and cross-device attribution channels that ship measurement code into mobile applications. The category is primarily mobile but reaches into web-and-mobile hybrid surfaces, and the SDK installation footprint is wide. Attribution SDKs run with broad access to device identifiers, network state, and event data; compromise produces direct privacy and exfiltration consequences. Account takeover, insider compromise, and ownership change are the dominant routes.

Cloud Storage and Asset Hosting. Object-storage services, image-hosting services, video-hosting services, and the asset-distribution surfaces that ship media into runtime pipelines. The category is cross-platform and reaches into browsers, mobile applications, AR/VR runtimes that consume 3D asset packs, and any Client Runtime that loads media or assets from a third-party host. Asset-hosting compromise is operationally distinctive because media can carry executable behavior (PDF JavaScript, USDZ behavior nodes, parseable formats with deserialization paths). Account takeover, network-path hijack, ownership change, and cyber exploitation all apply.

Plugin and Extension Distribution. Browser extension stores, IDE plugin marketplaces, productivity-tool extension stores, and the asset-distribution channels that ship plugin content into host runtimes. The category reaches across browsers, mobile shells with extension surfaces, desktop applications with plugin marketplaces (the Microsoft Store, the JetBrains Plugin Repository, Visual Studio Marketplace, and equivalents), and the productivity-tool extension stores that distribute add-ins. Plugin and extension content runs with elevated trust because the host application has explicitly admitted it. Account takeover, ownership change, insider compromise, and cyber exploitation all apply, and the marketplace itself is an aggregator with its own attack surface.

AI-Generated Content Services. Image-generation services, voice-synthesis providers, content-generation tooling, and the generated-content distribution surfaces that ship synthesized media into applications. The category is cross-platform and reaches into browsers, mobile applications, productivity tools, and any Client Runtime that integrates a content-generation API. The structural concern combines AI service compromise with media compromise: synthesized output is indistinguishable from authentic content at the rendering layer, and adversarial generation produces tailored deception output that the runtime treats as authentic. Account takeover, insider compromise, ownership change, and cyber exploitation all apply across the providers in this category.

6.5: Cross-Provider Patterns

The same aggregation patterns, defendability gaps, attack routes, and dependency dynamics repeat across categories. Eight patterns describe what the categories have in common.

- **Aggregation depth as a multiplier:** many of the categories above are aggregators: a CDN serving thousands of libraries, a tag manager firing dozens of vendor scripts through one container, a package registry holding millions of packages, an AI-service hub fronting models from many publishers, an asset-hosting service distributing media across thousands of dependent applications. Each of these categories concentrates risk the same way: one operator's failure becomes a failure in every application that depends on it.
- **The defendability gap that widens outside the browser:** SRI and CSP cover static-script and constrained-URL provider patterns well inside browsers. The gap widens for providers whose business model is delivering different content per request (tag managers, ad exchanges, model endpoints), for manifest-driven providers (web app manifests, OTA update manifests, AI tool manifests), and for AI-service providers whose responses are freshly generated model output rather than a known file with known contents. The gap widens further across non-browser Client Runtimes, where SRI and CSP analogs are largely absent at scale; the same provider serving the same content into mobile WebViews outside their host-browser context, Electron and Tauri shells, vehicle infotainment runtimes, IoT dashboards, productivity-tool extension hosts, AR/VR runtimes, or interactive runtimes that load user-generated content is generally uncovered.
- **Insider threat as a dominant compromise pattern:** no framework governing these providers requires the controls that would reach it. Vetting for publishing and build roles, access control on supply-chain-critical infrastructure, monitoring of routine insider actions that could mask Adversarial Placement and Access (APA), and threat

modeling at the personnel layer are each available, and none is mandated. Whether a given provider has implemented any of them cannot be determined from outside, which is the operative point: a dependent application cannot verify the insider posture of a provider it has never inspected, and holds no mechanism to ask. The provider ecosystem is a high-leverage insider target precisely because so few providers have controls calibrated for the threat, and the leverage compounds where the targeted role sits in a publishing channel that reaches thousands or millions of dependent applications.

- **Cyber exploitation and the misattribution layer:** many provider compromises are achieved through cyber exploitation of the provider's own infrastructure, code, build pipeline, or dependencies. When the resulting ATG event is attributed to the compromised provider rather than to the underlying adversary, the defensive response lands on the wrong layer. The actual adversary is obscured behind the visible provider name and may continue to operate against the same provider, or against analogous providers, from positions the misattribution did not surface.
- **Dependency chains and transitive risk:** providers depend on providers, which depend on providers. Provider inventory is necessary but not sufficient; the inventory must reach into the dependencies of the dependencies, transitively, and most provider inventories today do not.
- **Which attack routes apply at which provider scale:** account takeover dominates at small-to-medium provider scale, where individual maintainer credentials and per-team publishing access are the primary publishing surface. Ownership change is the increasingly observed route at large provider scale, where the entity itself is acquirable and the existing publishing channel has more value than any single credential or piece of infrastructure (the Polyfill.io case is the illustration). Insider compromise and cyber exploitation apply across both ends of the scale. Defensive priorities differ accordingly: small-to-medium providers warrant credential-and-publishing-channel focus, large providers warrant ownership monitoring and corporate-supply-chain awareness.
- **AI services as an emerging gap:** the AI and ML services category is growing rapidly in deployment, in the diversity of providers, and in the depth of integration into runtime decision logic. Established defendability mechanisms have less coverage here than for traditional script delivery: inference responses are not pinnable hashes, model files are not script sub-resources, prompt and tool definitions are not constrained URLs. Local AI deployments compound the gap by moving the provider boundary inside the device, where the same defendability gap applies and detection is harder than at the network layer.

- **The data-endpoint blind spot:** many provider categories deliver data rather than code: data brokers, profiling services, sanctions and watchlist feeds, reference data, market data, inference output, localization bundles, identity claims. The Two-Stage Attack architecture exploits exactly these endpoints: the runtime fetches data from a trusted provider, the runtime acts on the values, and adversarial values produce adversarial application behavior at the rendering layer. Information manipulation, the dominant ATG attack mode, is most often delivered through data-endpoint providers rather than code-endpoint providers; the data endpoint is the channel and the runtime acting on the data is the mechanism.

6.6: Defense Across Providers

SRI hash policy works for the static script sub-resources delivered from the Code and Package Delivery category but does not work for the same-URL-different-content delivery pattern that defines tag managers, ad exchanges, and inference endpoints. CSP allowlists work for the constrained-URL pattern but not for runtime-mutable providers whose business model depends on serving different content per request. Certificate pinning narrows network-path hijack but does not narrow account takeover, ownership change, insider compromise, or cyber exploitation. Pinning is weak by default outside the major browsers. Tag-container governance reviews work for the tag-manager category but not for the ad-tech pipeline that runs through it. Extension-marketplace review processes work for plugin distribution but not for the open-source registries that feed every JavaScript runtime. Vendor compliance audits operate on the procurement timeline, not on the publishing-event timeline of an ATG operation.

Three patterns carry across.

- **SSoT for runtime resources:** the architectural pattern named at the language level applies again at the provider boundary. A reference for what each provider should be delivering, against which actual deliveries are continuously compared, makes content-swap detection a routine rather than an investigation. The implementation differs by provider category (a hash list for static scripts, a behavior baseline for tag containers, a manifest baseline for OTA bundles, a response-shape and value-range baseline for data endpoints, a baseline for how a model's output normally varies), but the primitive is the same: store the truth somewhere that is not the provider, and compare every load against it.
- **Content-swap detection at the provider boundary:** when the runtime accepts content from a provider, the application has the option to compare what arrived against what was expected. The compare can be implemented at the rendering surface, in the network layer, or at a service that tees provider responses for inspection. The pattern is

reusable across provider categories: a script firing inside a tag container, another script loading from a CDN, a model returning an inference, a manifest delivering an OTA update, a font loading from a foundry, an extension updating from a marketplace. The runtime sees the same content the adversary delivered; a defender comparing against an expected baseline sees the deviation.

- **Provider inventory, extended transitively:** inventory of every provider an application depends on is the defensive prerequisite that most organizations do not currently meet. Knowing the providers is necessary but not sufficient. The Log4j case is the operative argument for going further: organizations were exposed to Log4Shell because the affected library sat several dependency layers below the components they directly depended on, often inside software they had not previously known they were running. Provider inventory must therefore extend into the dependencies of the dependencies, transitively. Most provider inventories today do not. The operational difficulty is significant, and the cost of not meeting it is the exposure surface Log4j made visible.

Malice Without Malware lands at the provider boundary. If malicious is no longer defined by the file's authorship or the channel's reputation but by what the runtime is being made to do, then the provider boundary is one of the places that judgment must be made. Provider inventory, ownership monitoring across the inventory, content-swap detection at the boundary, and SSoT for what each provider should be delivering are the architectural patterns that put that judgment into practice at the provider level.

The gap is structural. The code that implements a defense does not carry from one provider category to the next, and the architectural patterns that do carry are not deployed at scale across the Client Runtime landscape. Browser-side coverage is the exception, and it stops at the browser.

Chapter 7: Content Delivery Networks

A CDN is not just optimization infrastructure. It is the entity the runtime trusts to deliver code, content, and configuration into the rendering surface. The runtime resolves a URL, the CDN responds, the runtime executes the response. Whoever controls the CDN controls what the runtime executes. The runtime has no independent way to verify that response against what the application owner intended. CDNs sit at the highest aggregation depth of any single provider category in the deployed landscape, and that depth is what makes a CDN compromise consequential.

CDN-delivered content reaches Client Runtimes across the full platform landscape. Browsers are the most visible consumers because the web has the largest installed base of runtimes that render content on the client. CDNs also deliver JavaScript libraries, fonts, media, manifests, configuration, and increasingly model files into mobile applications through WebViews and native HTTP clients, into desktop shells that wrap a web runtime (Electron, Tauri), into developer IDEs and code editors, into AI chatbot interfaces, into smart-television and digital-signage runtimes, into vehicle infotainment systems, into IoT and operational technology (industrial control) dashboards, into AR/VR runtimes, into productivity-tool extension hosts, and into game and user-generated-content interactive runtimes. The browser is where CDN compromise is most visible; it is not the boundary.

The real-world ATG event is a CDN case: the Funnull/Polyfill.io operation of 2024, in which a long-trusted CDN hosting polyfills, the small compatibility scripts a site loads so older browsers can run newer code, was acquired by Funnull, a China-linked CDN operator, and used to deliver selective malicious content to its dependent population. CDN compromise spans every provider route (account takeover, network-path hijack, ownership change, insider compromise, cyber exploitation), and the ownership-change route is most visible in the CDN class. The longer-arc adversarial lifecycle from acquisition to re-emergence has been most fully observed at the CDN level.

7.1: Why CDNs Sit Where They Sit

Client Runtimes fetch content from remote origin servers, the servers the application owner names as the source of that content, and cache it locally to save bandwidth. The origin server controls how long a cached copy may be reused, through response headers, and lifetimes of zero are common for content the application owner expects to update without forcing client redeployment; that setting requires the runtime to re-fetch on every load. Serving those repeated fetches quickly to users in every region means keeping copies of the content near them, which is what a CDN does. The entity that operates the CDN inherits the trust the application owner extends to the application's own infrastructure,

because the CDN's URL is configured into the application as a delivery endpoint and the runtime treats responses from that endpoint as authentic.

Four conditions make that inheritance consequential.

- **Origin-authority delegation: the CDN is granted authority to answer in place of the origin server, in the application owner's name.** Domain configuration, contractual setup, and certificate issuance position the CDN as the entity the runtime resolves to when it calls the application's configured URL. The application owner does not deliver the content; the CDN does. The runtime cannot tell the difference and is not designed to.
- **Implicit trust granting:** the runtime treats CDN-delivered content as authoritative, with no verification mechanism in most cases. Subresource Integrity (SRI) exists in browsers for static script and style assets fetched from a known URL with a known hash, but most CDN content (dynamic responses, fonts, media, manifests, configuration, increasingly model files) sits outside SRI coverage even inside browsers. Outside browsers, SRI analogs are largely absent at scale across the non-browser Client Runtime landscape.
- **Runtime delivery position:** the CDN sits between the origin server and the rendering surface, in the network path the runtime traverses on every fetch. Whatever the CDN returns is what the runtime executes. There is no intermediate verification step that compares the response to what the application owner intended, and no economic alternative to that delivery position for content that has to reach the rendering surface at scale.
- **Aggregation depth:** a single CDN serves many Client Runtimes across many platforms simultaneously. Compromise at the CDN propagates to the full dependent population in one step. Aggregation depth is a multiplier; CDNs operate that multiplier at the highest end.

Each condition exists at lower intensity in other provider categories. CDNs are where all four reach maximum intensity at once. Scale, trust, and position together are what make the class consequential, not the technology itself.

All four conditions operate cross-runtime, not only inside the browser. Origin-authority delegation works the same way wherever a runtime resolves a URL the application owner configured. Implicit trust granting becomes more pronounced outside the browser, where SRI and CSP analogs are largely absent. Runtime delivery position is a property of the network path; the consuming runtime does not change it. Aggregation depth is a property

of the CDN's dependent population; it scales the same way regardless of which platforms that population spans. CDN compromise is therefore not browser-bound.

7.2: CDN Categories and the Trust Hierarchy

The CDN class is four trust models, not one. Application owners often collapse the relationship into a single idea, "we use a CDN", but the four CDN categories differ at the level of who is on the other end of the URL, what is contractually owed, and which routes in most apply.

Commercial CDNs. Cloudflare, Akamai, Fastly, AWS CloudFront, Google Cloud CDN, and Microsoft Azure Front Door operate on contractual relationships with the dependent applications, anchored by signed enterprise service-level agreements, identifiable corporate ownership, and named operational personnel. All five routes apply: account takeover, network-path hijack, ownership change through subsidiary acquisition, insider compromise, and cyber exploitation. Contractual structure constrains some routes more than others; it does not close any of them.

Open-source and community CDN services. cdnjs, jsDelivr, and unpkg deliver content without contractual relationship to the dependent applications that consume it; trust is inherited from the open-source library, the maintainer, or the operator's reputation, and ownership models vary. The routes that most apply are account takeover, insider compromise, ownership change (the Funnull/Polyfill.io pathway), and cyber exploitation. The lack of contractual structure does not predict less reliable operation; cdnjs, jsDelivr, and unpkg have served the ecosystem for more than a decade. It does change the recourse path after compromise.

Embedded script and asset hosts. Google Hosted Libraries, Google Fonts, Bunny Fonts, and Adobe Fonts operate as ecosystem services run by major platform companies. Trust is implicit in the platform brand: the dependent application owner trusts Google, Adobe, or Bunny by extension when consuming hosted libraries, fonts, or icons. Contractual structure varies, but the operative trust mechanism is brand reputation, not contract. The routes apply much as they do for commercial CDNs; the consequence is broader because a single compromise reaches every property that embeds the host's URL, regardless of whether the embedding owner has a direct relationship with the host.

Specialized CDN-like infrastructure. Image and media CDNs, video streaming networks, edge compute platforms, networks that deliver mobile over-the-air bundles, code updates an installed app pulls straight from a server without going back through the app store (CodePush, Capgo, Expo, vehicle infotainment OTA services), AI model and adapter distribution networks (Hugging Face, model and prompt registries for inference services), and AR/VR asset distribution platforms each operate variations on the CDN pattern with

different content types. The trust models split along content-type lines: media CDNs lean closer to commercial-CDN structure; over-the-air services can replace application code on the next launch, which CDNs serving script assets cannot; AI model distribution ships model weights and adapter files, content types that traditional CDNs never carried. The reach is into Client Runtime platforms beyond the browser: mobile applications consuming OTA bundles, AR/VR runtimes loading scenes, smart-TV systems pulling apps and media, vehicle infotainment systems updating maps and software, AI runtime hosts pulling model files.

Commercial CDNs have contractual structure and named ownership, which constrains some routes without closing others. An acquirer can buy a subsidiary that holds a delivery position. A certificate authority can issue for a domain it should not. An operator's publishing credentials can be stolen. Contractual formality decides who answers afterward; it does not decide whether a route is open. Open-source community CDNs have less formal trust structure; they have served the ecosystem for more than a decade and have been compromised when ownership of a long-trusted domain changed hands. Embedded script and asset hosts carry major-platform brand trust; brand trust does not narrow the routes in the way contractual structure does. Specialized CDN-like infrastructure delivers into Client Runtime platforms with less developed defensive primitives than browsers; the consuming runtime matters as much as the delivery operator.

The trust hierarchy is not a clean ranking. The CDN dependency an application carries is a different security problem in each of the four categories.

7.3: The CDN Acquisition Attack Model

Acquire, establish trust, weaponize, activate: each step is the precondition for the next. Acquisition without subsequent establishment of trust risks early detection; weaponization without prior trust-building exposes the operation; activation requires weaponization already in place. The trust that activation exploits is the trust the establishment phase built. The pattern is not a description of how a CDN attack might unfold; it is the description of how a CDN attack has to unfold to remain undetected long enough to matter.

- **Acquire:** the adversary takes ownership or operational control of the CDN. Commercial acquisition purchases the entity outright, often through layered ownership that obscures the buyer's identity. Influence operates through board placement, controlling-stake investment, or partnership arrangements that give the adversary effective decision authority without title transfer. Coercion targets the operator personally through financial pressure, regulatory exposure, family threats, or extortion. Cyber exploitation compromises the CDN's infrastructure or administrative credentials and operates the position covertly without an ownership event.

- **Establish trust:** the CDN operates legitimately under the new ownership. The same content serves the dependent applications. Performance, reliability, and content fidelity all hold. Existing dependents detect no change; new dependents onboard normally because the CDN looks indistinguishable from a legitimate operator. The phase often runs months to years; the longer it runs, the larger the dependent population grows and the deeper the trust extends, both of which the adversary harvests at activation.
- **Weaponize:** adversarial capability enters the delivery pipeline. The implementation is often selective: only requests from particular geographic regions, only requests from particular user-agents, only requests during particular hours, only specific content types, only specific dependent applications. Selectivity delays detection because the malicious content does not appear in most normal observation samples. Standard CDN telemetry, security testing, and content-integrity checks miss content that activates conditionally.
- **Activate:** activation delivers the malicious content. Triggers can be specific events (a known target visiting a specific URL), target conditions (an application loading on a specific platform or in a specific geographic region), or simply a scheduled time. Activation is the first step that produces detectable adversarial behavior; everything before it has been routine CDN operation. The detection window opens at activation; by that point the dependent population may include applications, properties, and devices that onboarded entirely under adversarial ownership.

Funnull/Polyfill.io of 2024 walks the model end to end. A long-trusted polyfill-hosting CDN was acquired outright from its original operator. The new ownership operated the CDN under the existing domain and against existing dependent expectations for an extended period. Selective adversarial capability entered the delivery pipeline, targeting particular dependent properties and particular conditions. Activation redirected visitors and delivered malicious code to a subset of dependent applications, surfacing through researcher detection rather than CDN-side observation. By detection, the dependent population had reportedly grown to more than one hundred thousand sites.

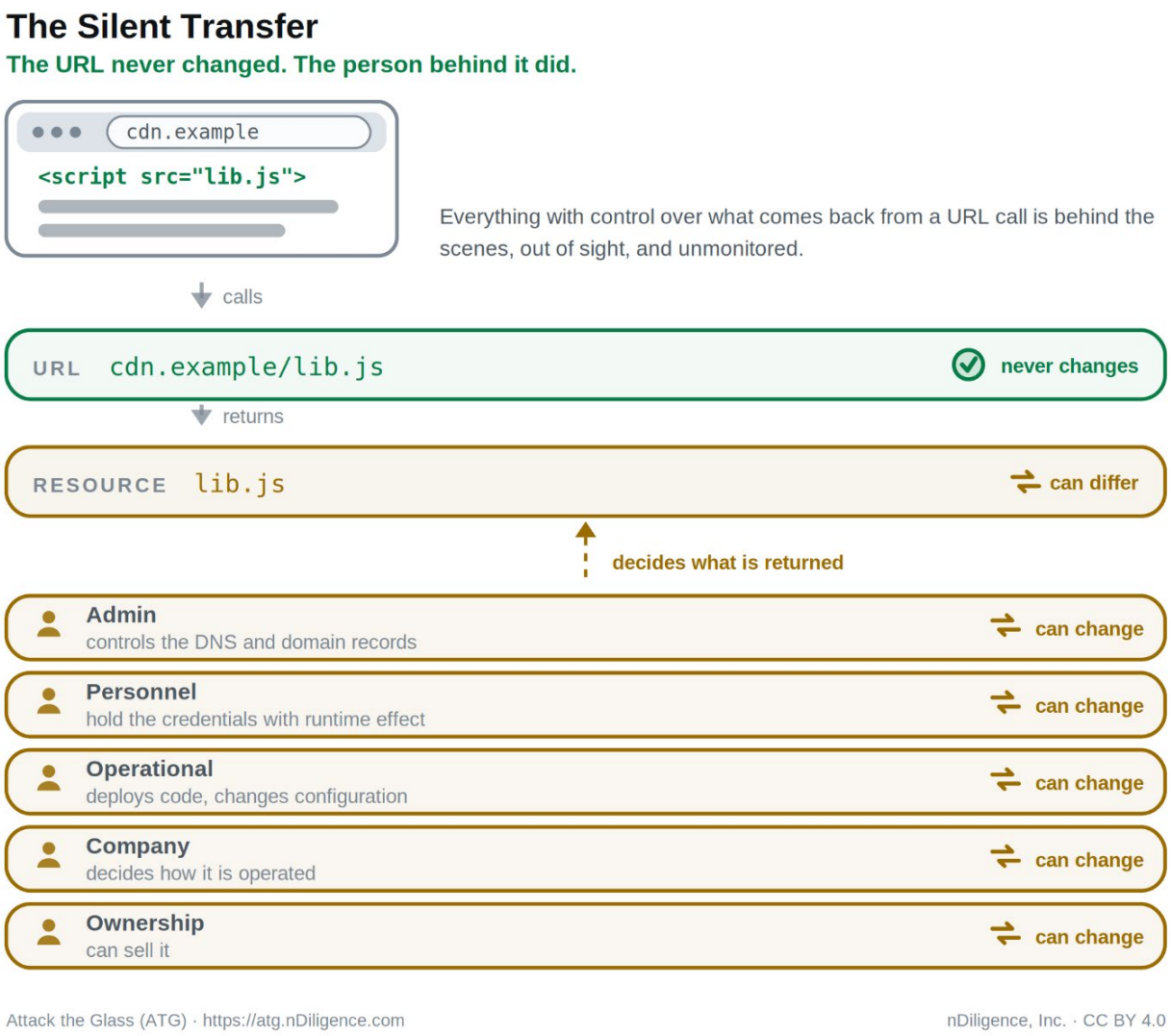

***Figure 24**: The URL never changed. The party behind it did, and nothing in the runtime registers the handover.*

The four-step pattern has the shape the Two-Stage Attack has at the resource level: the CDN's delivery URL operates legitimately for the period the adversary needs to build position, then delivers compromised content to the runtime that has already extended trust. Aggregation depth scales what activation reaches. A compromised analytics script reaches the sites and apps that embed that script. A compromised CDN reaches every dependent application at once.

7.4: CDN as Risk Multiplier

The reach a CDN compromise produces varies by CDN across orders of magnitude. Aggregation depth is uniformly high across the CDN class, but dependent populations vary by category. Reading any specific compromise requires reading which profile applies.

- **Major commercial CDNs:** each serves hundreds of thousands to millions of dependent sites. Cloudflare and Akamai together front large fractions of the public web; Fastly, AWS CloudFront, Google Cloud CDN, and Microsoft Azure Front Door each serve significant portions of the high-traffic enterprise application population.
- **Open-source and community CDN services:** cdnjs, jsDelivr, and unpkg each deliver to dependent populations measured in the millions of sites. Together they serve a meaningful share of the JavaScript ecosystem at runtime.
- **Embedded script and asset hosts:** Google Hosted Libraries, Google Fonts, Bunny Fonts, and Adobe Fonts reach billions of page loads daily, embedded across a population of properties that includes most of the consumer-facing web.
- **Specialized CDN-like infrastructure:** delivers into deployment populations measured in devices, not sites. Mobile OTA bundle services reach the installed bases of dependent mobile applications; AI model distribution networks reach the applications that load model files in order to run them; AR/VR asset distribution reaches the runtimes and devices that render scenes from those assets; smart-TV, vehicle, and IoT delivery networks reach their respective device populations.

Every figure is a count of what a single compromise reaches in one step. Cross-platform reach compounds the per-CDN counts. The same CDN often serves into multiple Client Runtime platforms simultaneously: a CDN that delivers JavaScript libraries to web properties also delivers them to mobile applications consuming them through WebViews, to Electron and Tauri shells that embed them as dependencies, to smart-TV runtimes loading them as app framework code, and to AI chatbot interfaces and developer IDEs that pull them at runtime. A single compromise reaches every dependent application that consumes the compromised content, across every Client Runtime platform that consumes it, in one step. The risk multiplier is not browser-bound; the multiplier compounds wherever the CDN's content is consumed.

A sophisticated adversary with CDN-level position chooses how to activate. Targeted activation runs against a narrow population (a specific dependent application, a specific geographic region, a specific user-agent or device type, a specific time window) and preserves stealth at the cost of impact. Broad activation runs against the full dependent population at once and maximizes impact at the cost of detection time. The choice

depends on the operation's goal: targeted activation suits influence-and-control operations against specific entities; broad activation suits campaigns where reach itself is the objective.

The risk multiplier and the operational discretion together make the CDN class consequential. Reach is the technical property; discretion is the operational choice. CDN-class compromises can be precise enough to evade detection through the dormancy phase and broad enough to determine outcomes for an entire dependent population. Other supply-chain compromises trade stealth for scale. CDN compromise does not.

7.5: Lying in Wait

Mature adversarial CDN operations sit inside a longer lifecycle: pre-positioning, dormancy, activation, persistence tradecraft, re-emergence. The lifecycle pattern is familiar from the broader supply-chain adversary literature. What is ATG-distinctive is concentrated in the dormancy phase, *Lying in Wait*.

Lying in Wait, dormancy as detection-defeat: this is the most important phase of the CDN lifecycle. A dormant adversarial capability sits pre-positioned within a genuinely legitimate, trusted dependency. The adversary acquires the position, then operates the CDN legitimately for months or years, serving exactly the content the dependent applications expect. The dwell phase produces no detectable adversarial signature; standard CDN telemetry, security testing, content-integrity verification, and behavioral monitoring all return normal results because the CDN is in fact operating normally, which is why nothing fires (during the dwell there is no malicious file to find). The capability lies in wait, and what conceals it is not a hidden piece of code but the working dependency around it. The only detectable event is activation, and by then the dependent population has often grown materially under the new ownership. Defensive postures built around active-attack signatures miss the dwell window entirely. The threat is present but inert. There is no event to detect, often not even the strike itself. This is why CDN-class compromises are difficult to prevent and are frequently discovered weeks or months after activation begins.

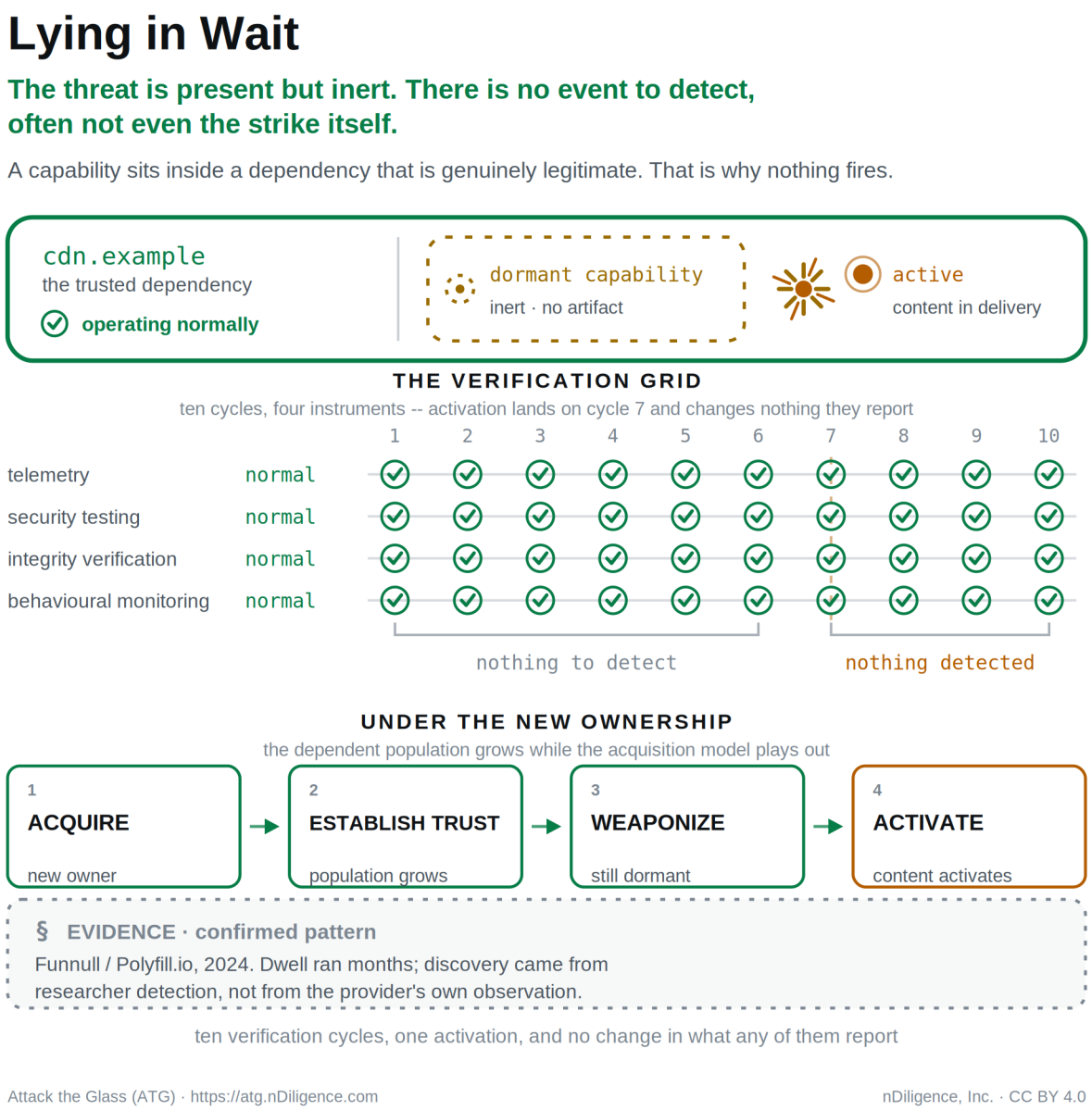

***Figure 25**: Dormancy is indistinguishable from a benign service by construction. Activation is the only detectable event.*

The host of a Lying-in-Wait capability is not faking legitimacy. It is legitimate: a real business, with real revenue and a real market, adversarial-controlled at the ownership layer and weaponized only in the rare instance. That genuine legitimacy is why nothing fires during the dwell. Three related terms are worth keeping apart: a shell exists only to obscure ownership; a front is a fake business or facade; a cover is a genuine business that is weaponized only rarely. A Lying-in-Wait capability is hosted by a cover. Separately, the persistence phase routes CDN infrastructure through hosting providers and jurisdictions that do not cooperate with takedown requests, law enforcement inquiries, or content-moderation demands. The mechanism is where the infrastructure is placed: the chosen

hosting providers sit in jurisdictions that decline to honor cross-border enforcement actions, or that hold legal positions making compelled cooperation slow or unreliable. This routing is familiar from the broader supply-chain adversary literature. The ATG-relevant consequence is direct: detection of the malicious CDN does not necessarily lead to its removal, and compromised content can continue to deliver to the dependent population for an extended period after detection, sometimes indefinitely, while takedown procedures wind through unwilling intermediaries.

Re-emergence closes the lifecycle. After detection or takedown, the adversary cycles the asset through new hosting infrastructure, often under a new domain or rebranded operator identity, and resumes operation. The pattern is the CDN-level expression of Adversarial Asset Cycling, the broader ownership-opacity pattern. The dependent population the new asset acquires may overlap meaningfully with the population the original asset served, particularly when the cycling re-uses code patterns, naming conventions, or recognizable operator behavior.

Lying in Wait is what hides the operation. Non-cooperative hosting is what keeps the operation running after detection. Re-emergence is what brings the operation back under a new identity. The four-step model captures one event; the lifecycle captures the operator. Defenders who detect events without recognizing the operator will keep finding the same operator behind new events.

7.6: Typosquatted and Impersonating CDN Domains

The typosquatting and impersonation surface differs from the acquisition attack model. The acquisition attack uses the legitimate CDN: the same domain, the same publishing channel, the same trust relationship that has worked for the dependent population for years. Typosquatting and impersonation both substitute a different entity, and they arrive by different routes. Typosquatting registers the misspellings a developer might type or paste: cdnjs.cloudflare.com becomes cdnjs.cloudfare.com with a letter dropped, cdn.jsdelivr.net becomes cdn.jsdeliver.net with a letter inserted. Impersonation constructs a domain no one reaches by error, reusing the brand's own words (jsdelivr-cdn.example, googlefonts.example) or replacing a dot with a hyphen so the legitimate name still reads correctly at the front of the address (cdnjs.cloudflare-com.example). The substituted entity has no relationship to the legitimate CDN; the attack succeeds when the developer or the build pipeline references the wrong domain and the runtime fetches from it.

Defense is largely organizational. Developer awareness, code-review discipline, dependency declaration audits, and pre-deployment scanning of CDN domain references catch most typosquatting before the misreference reaches production. CSP allowlisting constrains which CDN domains the runtime is permitted to fetch from in browsers; this

defense is browser-specific and covers static script and style sub-resources where SRI is not in use. Outside browsers, the equivalent defense is largely absent: most non-browser Client Runtimes lack CSP analogs, and the typosquatted domain reaches the runtime if the application owner has referenced it.

The pattern is well-documented in the broader phishing and supply-chain literature. What matters for ATG is not the pattern itself but that typosquatted domains enter the Client Runtime trust graph alongside legitimate CDN domains. The CDN domain trust graph carries two distinct attack surfaces: legitimate CDNs the application owner has chosen to depend on, and impersonating entities that have entered the graph through error or deception. The runtime treats both the same way at fetch time.

Chapter 8: Third-Party Script Loading: The Direct Trust Problem

A third-party script tag hands a decision to someone else and never takes it back: which code runs at that address is the vendor's call, not the application owner's. The application owner places the tag; the vendor places the code at the URL the tag points to; the runtime fetches whatever sits at the URL on every load and executes it with the same permissions the user granted to the host application. The vendor controls what sits at that address and can change it between one fetch and the next. By writing that one line, the application owner has put the vendor inside the trust boundary: the vendor's code runs with the host application's full privileges and can change what the runtime does and what the user sees.

In browsers the mechanism is script tags loading vendor JavaScript URLs. That is the most visible case, not the boundary. Mobile WebViews load vendor scripts and SDKs into the host application's runtime, with the host's permissions. Hybrid mobile frameworks pull vendor JavaScript modules at runtime through their own loading mechanisms. Desktop shells that wrap a web runtime (Electron, Tauri) load third-party JavaScript and Node modules into the desktop runtime, with the desktop application's privileges. Productivity-tool extension hosts load extension code that inherits the productivity tool's data access. Vehicle infotainment runtimes, IoT and operational technology (industrial control) dashboards, AR/VR runtimes, and user-generated-content interactive runtimes each carry their own runtime-script-loading mechanisms. The trust dynamics are the same across all of them: the vendor delivers the code, the host runtime executes it, the host's permissions extend.

Two consequences follow from this delegation, and the broader supply-chain literature does not address either. The first is that most organizations cannot enumerate the third-party scripts their applications load at runtime. No amount of operational discipline closes it. The second is that loaded scripts inherit the host application's permissions wholesale: same DOM and storage, same network endpoints, same user-granted device APIs. The runtime draws no privilege boundary between first-party and third-party code at the moment of execution.

8.1: Script Loading as Runtime Delegation

Every loaded script follows the same path: the runtime resolves the URL, fetches the response, and executes it as code. The directive varies across Client Runtimes (script tag, WebView load API, native module loader, extension manifest); the fetch-and-execute structure does not.

Runtime-loaded scripts are not statically-linked libraries. A statically-linked library is fixed at build time. Its behavior travels with the file the application owner ships; if the library is

malicious, that file carries the malicious code, and the application owner can open it and look. A runtime-loaded third-party script does whatever the code sitting at that URL does on the day it is fetched, and the application owner has no way to inspect that code before it runs.

A typical e-commerce site, news site, or SaaS web application loads dozens to hundreds of distinct third-party script sources at runtime. A typical mobile application carrying hybrid frameworks and analytics SDKs loads tens of script sources. A productivity-tool environment loads a script population that grows with each extension the user installs. Each source is a Placement and Access (P&A) position the vendor holds at runtime. Every one of those sources carries both problems: the application owner cannot list them all, and the runtime places no limits on any of them once they execute.

8.2: The Third-Party Script Inventory Problem

In production, no one can produce a complete list of the scripts a runtime loads. The problem is widely recognized; better operations have not closed it. Four structural reasons hold the gap open.

Transitive script loading. Scripts load other scripts. A tag manager, the hosted console a marketing team uses to decide which scripts a page pulls in, loads analytics scripts; analytics scripts load advertising scripts; advertising scripts load attribution scripts. The one script the application owner wrote into the page is a small fraction of what ends up running.

Dynamic script behavior. The script loaded today can load different content tomorrow without any change the application owner sees. The URL is constant; the code at the URL is not.

Tag manager dynamics. Tag managers add and remove scripts dynamically based on configuration changes outside the application's release process. A marketing team's change to a tag-manager rule alters the runtime script population without touching the application code.

Vendor SDK transitive dependencies. A vendor SDK loaded as a single script can pull in dozens of additional scripts as its own dependencies. The application owner sees one inclusion; the runtime executes many.

All four reasons follow the loading pattern onto every platform. A mobile WebView running a tag manager and several analytics SDKs faces the same transitive, dynamic, and dependency-driven population it cannot enumerate. An Electron shell with multiple npm-loaded modules presents the same. A productivity-tool environment with several installed extensions presents the same. The platforms differ; the inventory problem does not.

Better operations close pieces of the gap, never the whole of it. Better tooling, better vendor governance, and better tag-manager hygiene cut into the population without enumerating it. The runtime executes a population of scripts the application owner cannot enumerate, even with full inspection authority over the source code. The problem is structural.

8.3: High-Trust Script Categories

High-trust script categories operate at elevated privilege by design. Their function requires it.

- **Analytics scripts (Google Analytics, Mixpanel, Segment, mobile analytics SDKs):** read navigation, interactions, and often form contents.
- **Advertising scripts:** inject content and modify display.
- **Payment scripts:** handle financial information directly.
- **Identity scripts:** manage authentication, session establishment, and credentials.
- **Trust and engagement widgets (chat, comments, captcha, document signing):** hold privileged DOM and runtime access.
- **Mobile attribution and tracking SDKs:** reach device APIs.
- **Productivity-tool extensions:** reach user documents and data.

Elevated trust is granted by necessity. An analytics script that cannot read interactions has no analytics function; a payment script that cannot handle financial information has no payment function; an identity script that cannot manage credentials has no identity function. Whoever compromises one of these scripts gets that same elevated access, for the same reason the script was given it. What makes a high-trust script useful is what its compromise gives the adversary.

8.4: Runtime Anarchy

Every control that exists here acts at the point content arrives: CSP governs what may load and execute, SRI checks the integrity of what is fetched, and CORS governs whether running code may read what a cross-origin response returned. Once a script clears that gate and executes, nothing governs what it may do. Within a single execution context every script is co-equal, with full rights to read and modify shared state, the DOM, globals, and other scripts' data. There is no per-script sandbox, no privilege hierarchy, and no authority imposing limits; the only real interior boundaries are coarse and document-level (cross-origin iframes, sandboxed frames, workers), not per-script. That is Runtime Anarchy: CORS

and CSP enforce rules on who is allowed through the door but once they are in the room, there are no rules. A script that reaches this context reaches every privilege the host application holds: the same DOM and UI access; the same cookies, localStorage, IndexedDB, and equivalent storage; the same network endpoints; the same camera, microphone, geolocation, and payment APIs where granted or invoked; the same sensor and actuator APIs in vehicle, AR/VR, IoT, and OT contexts; and the same document and data access in productivity-tool extension contexts. The third-party receives the host's permissions wholesale, by design.

Loaded scripts are not isolated. A subprocess runs in its own address space; a sandboxed component runs under a constrained capability set; an out-of-process service runs at a network boundary. A third-party script runs inside the host's runtime, with the host's permissions, in the same privilege space first-party code occupies. Third-party script compromise equals full application compromise from the user's perspective.

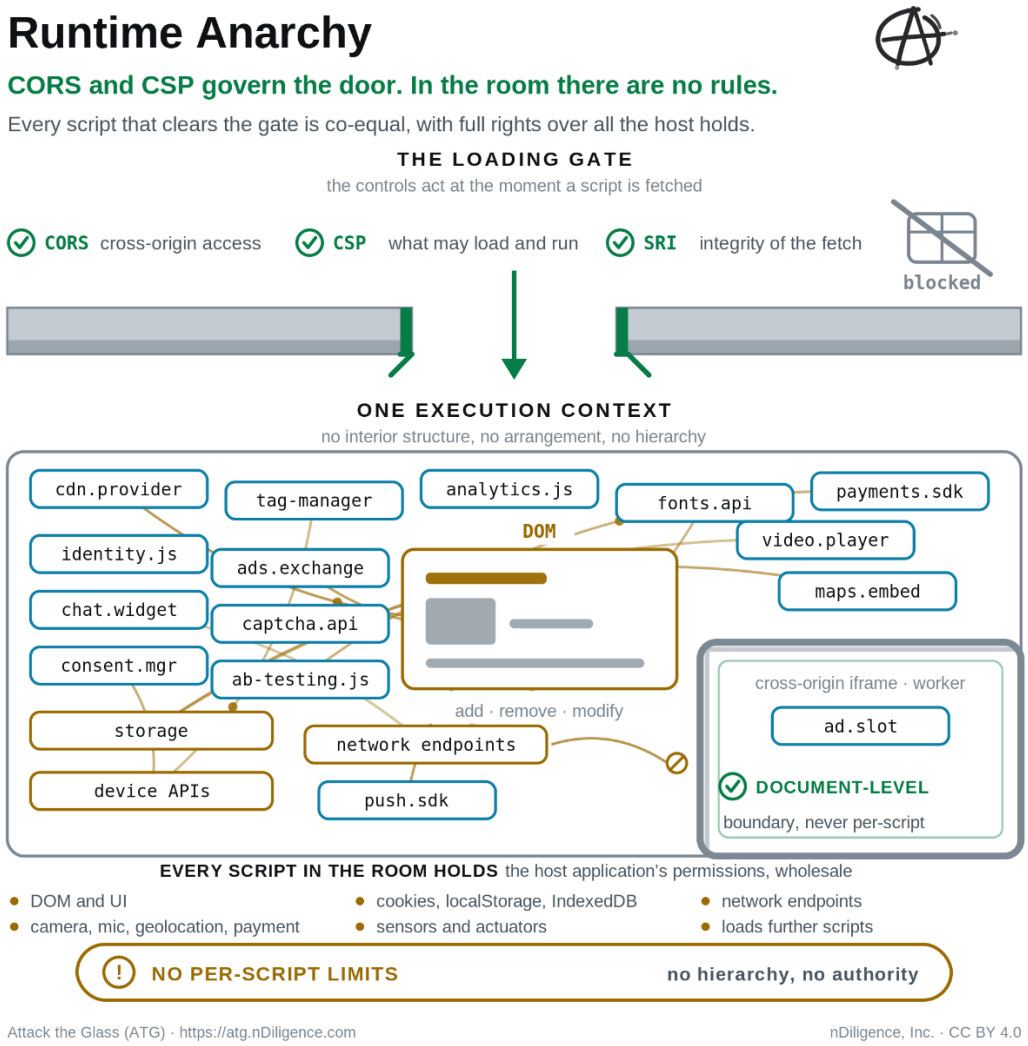

***Figure 26**: Controls govern who gets through the door. Inside, scripts share one runtime with no limits between them.*

Real compromises show what that inheritance buys an adversary. A compromised analytics script can exfiltrate payment data. A compromised payment script can capture identity credentials. A compromised mobile attribution SDK can access device APIs the host application was granted. A compromised productivity-tool extension can reach the user's documents and data. The runtime does not know the script is acting beyond its declared purpose. The runtime does not have a model of declared purpose.

The same inheritance applies to everything else the runtime fetches and trusts. A compromised CDN-delivered library inherits the application's trust. A compromised inference response, the answer that comes back when the application calls a model, inherits the application's authority to act on it. A compromised plugin inherits the host application's privileges. A compromised data endpoint inherits the application's trust. A compromised over-the-air bundle, code an installed app pulls straight from a server without going back through the app store, inherits the host application's permissions. As long as a runtime accepts content because of where it came from, and then runs that content with the host's permissions, whoever compromises the source gets the host's permissions too.

Supply Chain Injection is how content enters. The Runtime Injection Advantage is why an adversary would rather act inside the runtime than anywhere earlier in the chain. Runtime Anarchy is what the runtime gives the adversary once content has entered: co-equal execution with the host's full permissions. Combined, they are the full ATG attack architecture.

8.5: Scale of Exposure Across Application Categories

Exposure varies sharply by application category. E-commerce typically carries the highest third-party script populations, often dozens to over a hundred sources per page. News and media sites are heavy, driven by advertising, analytics, and engagement widgets. SaaS and productivity surfaces are moderate. Banking surfaces are lower on public pages and higher inside authenticated experiences. Government public-facing surfaces span thousands of sites, comparable in scope to large enterprise, with per-page populations generally lower than commercial peers.

Authentication does not change the runtime architecture. Tens of thousands of authenticated portals sit beyond the public-facing front: government and agency portals, first-responder web portals, the web-based tools governments procure from their vendors and the Defense Industrial Base, secure banking portals, and stock-trading platforms. All are Client Runtimes that load third-party scripts in the same way public surfaces do. Login walls, network isolation, and procurement controls reduce the audience, not the architecture. The inventory and inheritance dynamics are the same.

Specialized Client Runtimes change the consequence calculus. Vehicle infotainment, IoT and OT dashboards, and AR/VR runtimes carry smaller per-application script populations than browsers or mobile, but the consequence per compromised script is disproportionate. A compromised script reaching a vehicle infotainment runtime, an industrial control dashboard, or an AR/VR sensor stream can produce physical-world effects browser-based exposure does not.

Counting the scripts changes how large the problem looks; it does not change how the runtime treats them. Every loaded script is a placement and access position. Every loaded script inherits the host's permissions.

Chapter 9: Open-Source and the Path to the Client Runtime

Modern applications running inside Client Runtimes are mostly other people's code. A typical bundled application, whether it ships through a web origin, a mobile-app store, a desktop installer, a vehicle infotainment image, a smart-TV package, or a productivity-tool plugin marketplace, carries more open-source dependency code than application-authored code. The application owner writes the code that wires the pieces together; the rest comes from npm, PyPI, Maven, NuGet, crates.io, RubyGems, the package ecosystems for embedded runtimes, and the transitive dependency graphs each of those carries. The runtime that executes the assembled application cannot tell the application owner's code apart from the dependency code, and grants both the same trust at the rendering surface.

Traditional supply-chain security treats the open-source ecosystem as an upstream-of-build concern: vet the source, scan the package, sign the published package, monitor the registry. These controls are valuable, but they cluster at the registry and the build. By the time dependency code reaches a Client Runtime and starts executing on the rendering surface, almost none of them are still in force. The Client Runtime is where the user encounters the system, and where ATG operates.

Two properties make the open-source supply chain ATG-distinctive. The first is that the threshold for an adversary to acquire Placement and Access (P&A) inside an open-source project is structurally lower than the threshold for any other comparable supply-chain access route: the *Open-Source Open Door*. The adversary does not bypass trust the way a backdoor does; they are handed it, walking in the open door as an invited, trusted contributor or maintainer. Nothing is broken into; the door was open by design. The second is that once code from many sources is combined and loaded into the runtime, the record of which source supplied which part is gone. The runtime executes the bundle as one unit and cannot trace any part of it back to the project that wrote it. That is Origin Laundering. One script can build runtime code out of data supplied by another source, so the code that finally executes has no single author. As in money laundering, adversarial input passes through legitimate sources until no one can say where it started, and it moves along inside the clean flow. Each property defeats a different existing control. Together they decide what an open-source compromise is worth to an adversary once the code reaches the runtime.

9.1: Why Open-Source Matters for ATG

Open-source dependency code dominates what executes inside Client Runtimes. Public dependency analyses find that open-source components make up the large majority of modern codebases (Sonatype, State of the Software Supply Chain; Black Duck, Open Source Security and Risk Analysis). Both measure composition at the codebase level rather than inside a rendered bundle, and no public dataset measures the bundle directly. A

bundle inherits that composition through the build, with the remainder divided between application-authored code and the build-tool output that links them. Mobile applications built on hybrid frameworks, desktop applications built on shells that wrap a web runtime (Electron, Tauri), vehicle infotainment systems running embedded JavaScript engines, IoT and operational technology (industrial control) dashboards built on web stacks, and productivity-tool extensions all carry comparable proportions. The host platform varies; the dependency dominance does not.

Adversaries reach that dependency code through entry vectors the broader supply-chain literature has documented in depth: maintainer credential compromise, new-maintainer takeover, long-game contributor infiltration, intentional sabotage by the original developer, CI/CD pipeline injection, dependency confusion, and typosquatting at ecosystem scale. Specific cases include XZ Utils, ua-parser-js, event-stream, node-ipc, and the Magecart-related open-source compromises.

What matters for ATG is what the compromised code can do once it is executing on the user's screen. Because most of the executing code is dependency code, an upstream compromise lands in a runtime that cannot tell the compromised dependency from code the application owner wrote, and gives it the same trust on the rendering surface. The registry-level and aggregation-level patterns operate upstream. The share of running code that comes from dependencies is what turns a compromise at the entry point into a consequence on the Glass.

9.2: The Open-Source Open Door

Open source is the dominant, legitimate, wide-open route by which code reaches the Glass, and acquiring Placement and Access (P&A) inside an open-source project carries a lower threshold than any other comparable supply-chain route. Most of what runs on the Glass is code no one on the target's payroll wrote. Employment requires hiring processes, background checks, identity verification, contractual relationships, and a continuing employment record. Vendor positioning requires a commercial relationship with a buyer or a recognized place in a procurement chain. Commercial acquisition requires capital and produces corporate-transaction visibility. Open-source projects routinely look for maintainers and admins from the contributor base, and they do so without any equivalent vetting infrastructure.

Three mechanics account for the low threshold:

- **Public recruitment:** active "looking for maintainers" calls are common across the major package ecosystems. Projects with declining maintainer engagement, single-

maintainer projects, and projects whose original developer has burned out actively seek volunteers.

- **Contribution as credential:** sustained helpful contributions over months earn reviewer or commit privileges. The contribution history is the credential. No employment relationship, no background check, and no formal identity verification is involved at any step.
- **Single-decision transfer:** for some projects, becoming a co-maintainer or sole maintainer requires only that the existing maintainer agrees. The transfer can happen in a single email exchange.

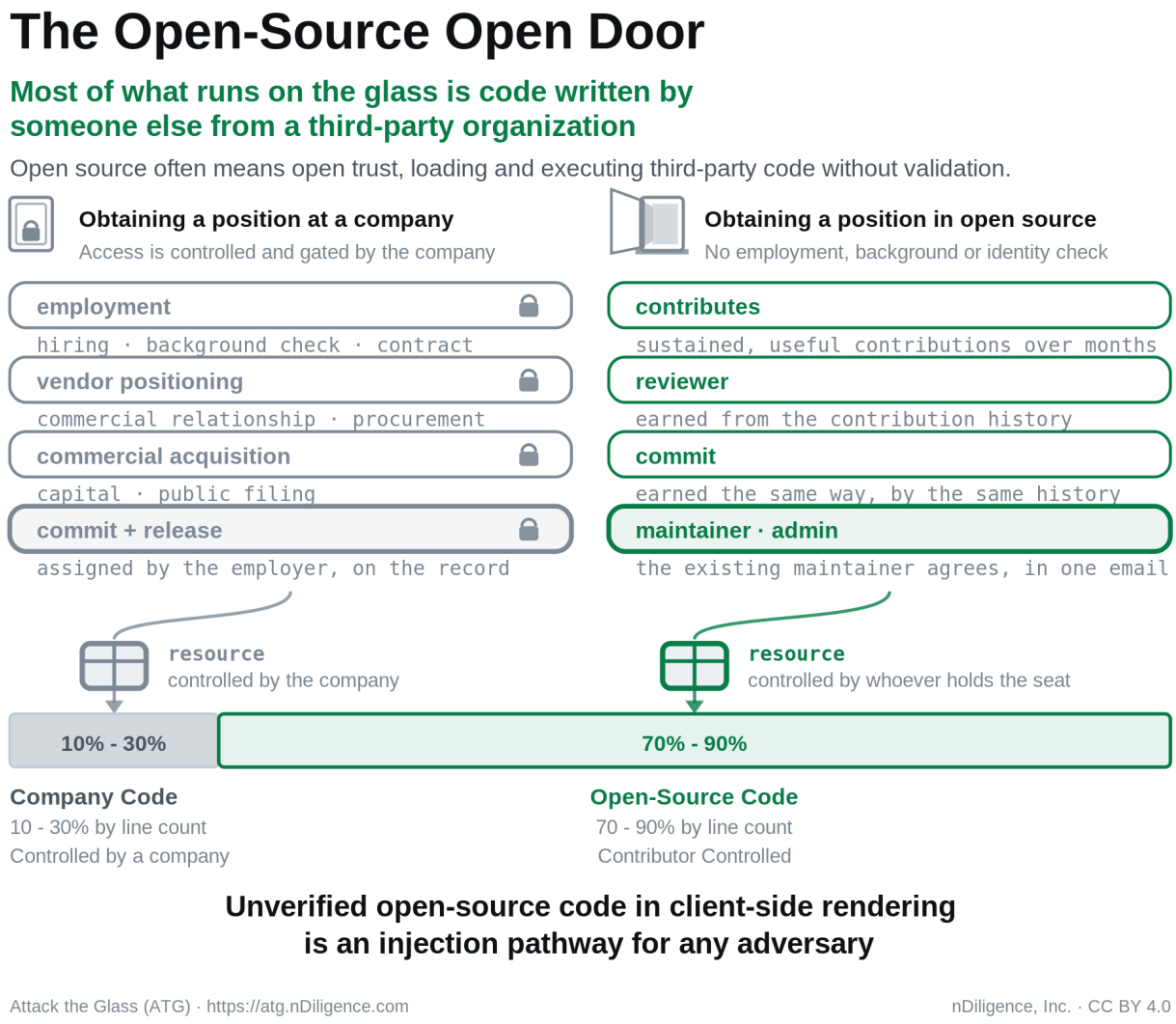

***Figure 27**: The lowest-barrier route to a rendering surface is contribution to something everyone already depends on.*

XZ Utils (2024) demonstrates the pathway end to end. A single contributor, over years of legitimate-looking commits, earned reviewer access, then commit access, then effective

control over a critical infrastructure project. No employment relationship, contractual structure, or commercial transaction stood between the contributor and project control.

Maintainer and admin positions hold authority over more than the commit log:

- **Project governance:** the maintainer's voice is the dominant one in roadmap, design, and release decisions.
- **Publishing and release authority:** the maintainer signs and publishes the next release. To the registry, that signature is the proof that the released package came from the project.
- **Community trust positioning:** the maintainer's name and reputation vouch for the project, and downstream consumers, employers, conference programs, and adjacent projects rely on that.
- **Communication and documentation channels:** the maintainer controls what the project tells its users: changelog framing, security advisory wording, deprecation timing, end-of-life notices.
- **Issue management:** the maintainer decides which bug reports are recognized, which are reframed, and which are quietly dismissed.
- **Downstream relationship building:** the maintainer position is a credential that opens doors to other projects, advisory roles, and offers of employment from companies that depend on the project.

Open-source projects depend on community trust mechanisms (reputation, contribution history, code review) for access decisions. The mechanisms are designed to detect bad code in individual contributions; they are not designed to detect Adversarial Placement and Access (APA) by patient contributors with long-game intent. An adversary who wants insider access takes the cheapest route to it, and open source is the cheapest: the threshold for becoming the insider is lower there than on any other supply-chain access pathway.

9.3: The Path from Package to Rendering Surface

The path from open-source package to executing rendering-surface code is several layers deep, and the layers carry uneven controls. A maintainer publishes a package to a registry. The registry's controls cover signing, account integrity, and the response to maintainer compromise. An application's build process resolves the package, fetches the published files, and bundles them with the application's own code and the files of every other dependency it resolved. The build's controls cover dependency scanning, lockfile pinning,

software-bill-of-materials generation, and integrity checks on the files it fetched. The bundle then ships through a distribution channel: a CDN for web origins, an app-store binary for mobile, an installer for desktop, an over-the-air channel that pushes code to vehicles, IoT devices, and smart TVs after they are already in service, without a store review in between, a marketplace listing for plugins and extensions. Each channel carries its own controls: code signing, store review, transport integrity. The host Client Runtime loads the bundle and executes its contents at the rendering surface. At that layer, the controls thin to almost nothing.

A package published to npm reaches an Electron desktop application through the same path it reaches a React web origin. A Python package published to PyPI reaches a Jupyter notebook front-end through the same path as it reaches a server-side service. A Java library published to Maven Central reaches an Android APK through the same path it reaches a desktop JAR. A Rust crate published to crates.io reaches a Tauri desktop shell or a vehicle infotainment image through the same path. The host platform determines which Client Runtime finally executes the bundle. The path is the same, and so is the way the controls thin out along it.

Two properties of the path are ATG-specific. The runtime cannot distinguish dependency code from application code at execution time. Any compromise that survives the build is granted application-level trust at the rendering surface, with the same access to runtime, storage, network, and device APIs the application owner's own code holds. Dependency code is the most common case of trusted-but-foreign code the runtime executes.

The runtime's other blind spot is provenance: it holds no record of where the code it is executing came from. It cannot answer which dependencies were used to build the bundle, what their published versions were, whether the code in the bundle matches what those projects published, or whether something was added to the bundle after the registry-published sources were resolved. Whatever list of contributing packages the build had, that list stayed at the build. The bundle itself is opaque to the runtime that loads it. The build knew where each part came from; none of that knowledge travels with the bundle to the rendering surface.

9.4: Origin Laundering

Dependency analysis examines one thing; the runtime executes another. Dependency tooling resolves the manifest (package.json, requirements.txt, Cargo.toml, Podfile, build.gradle) and reasons over what the manifest names. The bundle that ships is a different thing entirely, assembled by the build tool from the resolved dependencies plus everything the build added on top: transitive dependencies the declared packages pulled in, code generated by plugins or transforms, code the bundler copied in directly for

performance, polyfills the build tool added, and package versions the resolution algorithm picked at dependency depths the manifest never records. The Client Runtime sees the bundle. It does not see the manifest.

A defender who validates the manifest validates the wrong thing. The manifest names the dependencies the application asked for; the bundle carries the versions that were actually resolved, plus everything the build tool added on top of them. Software-bill-of-materials generation that operates against the manifest produces an inventory of what the application asked for, not an inventory of what the bundle contains. Adversarial code introduced at any of the build-tool layers (plugin transforms, polyfill insertion, code the bundler copied in directly) lives in the bundle without appearing in the manifest. Compliance that rests on a software bill of materials assumes the bundle matches the manifest, and the build tool guarantees no such thing.

A defender who validates the bundle validates the static delivery, not the runtime behavior. Bundle hashes, code signatures, and Subresource Integrity tags answer one question: did the bundle that arrived match the bundle that was published. They do not answer what the bundle does once the runtime executes it. The Manifest Illusion and Runtime Anarchy operate inside the runtime, after the bundle has been validated and its contents have begun to execute. Bundle validation is necessary work, and Subresource Integrity does the job it was built for. Neither is sufficient at the runtime trust boundary.

The problem is invariant across bundle formats. A JavaScript bundle delivered to a browser, an iOS app binary delivered through the App Store, an Android APK delivered through Google Play, an Electron asar archive, an Office add-in package, an OTA update bundle delivered to a vehicle infotainment system, a smart-TV application package, a Tauri desktop binary: each is a unit of execution whose contents were composed at build time, and each is paired with a manifest the runtime never sees. The packaging differs. The split between what the manifest declares and what the bundle executes does not. That split is Origin Laundering: the bundle runs as one unit but carries no record of where its parts came from.

9.5: Plugin and Extension Marketplaces

Plugin and extension marketplaces carry a compromise to executing code on the user's screen more directly than any other open-source route. Developer-facing package registries deliver code into application bundles, where Origin Laundering applies but a build step at least exists and some controls run there. Plugin and extension marketplaces deliver code directly to end-user Client Runtimes. The user installs an extension or plugin and the code the marketplace published runs in the user's host application, against the user's data, on the user's device. No intervening build step exists for any control to operate at. Marketplace categories reach across the deployed platform landscape:

- **Browser extension stores:** Chrome Web Store, Firefox Add-ons, Edge Add-ons, Safari Extensions Gallery.
- **IDE marketplaces:** VS Code Marketplace, JetBrains Marketplace, Eclipse Marketplace.
- **Creative-tool plugin stores:** Blender, GIMP, Maya, Houdini, FreeCAD.
- **Productivity add-in marketplaces:** Microsoft AppSource (Office Add-ins built on Office.js), Slack App Directory, Atlassian Marketplace, Google Workspace Marketplace.
- **In-app extension and asset marketplaces in productivity, design, and creative tools:** Figma plugins, Adobe Creative Cloud add-ons, Procreate brush packs, Lightroom presets, and analogous in-app marketplaces that publish executable or runtime-loaded behavior into the host tool.
- **Game-platform user-generated-content stores:** Roblox UGC experiences, Steam Workshop (mods, maps, scripts), Minecraft Marketplace, Fortnite UGC mode, and similar surfaces where the user-distributed asset includes executable behavior delivered directly to end-user runtimes.
- **Smart-TV application stores:** Roku Channel Store, Samsung Tizen Apps, LG webOS apps, Apple TV App Store, Amazon Fire TV apps, and the partner channels on console application stores that distribute applications running inside Client Runtimes.

Plugins literally run as CSR. A browser extension executes JavaScript in the browser's Client Runtime alongside the host page. A VS Code extension executes JavaScript in the editor's runtime. An Office add-in executes JavaScript in the productivity tool's runtime. A Blender plugin executes Python in Blender's scripting runtime. A Roblox experience or a Steam Workshop mod executes runtime-interpretable behavior inside the platform's interactive runtime. The host application offered the plugin or extension a privileged seat at execution, and whatever the marketplace published takes that seat.

Privileges the seat carries are operational, not theoretical. Browser extensions can read every page the user visits and inject content into them. VS Code extensions can access workspace files, execute arbitrary code in the developer's environment, and read whatever credentials live in the editor's context. Office add-ins can interact with documents, including the document the user has open. IDE extensions reach the developer's source tree, build configuration, and any tokens cached for source-control or deployment authentication. Game-platform UGC assets reach the runtime's input, network, and persistence APIs. Runtime Anarchy applies here at marketplace scale. Every installed plugin inherits the host application's permissions, and a publisher compromise reaches every install with those permissions intact.

Plugin-Marketplace Publisher Compromise stands as a confirmed event category, alongside the established CDN and third-party-script categories. The marketplace surface is no longer a peripheral case folded into "browser extensions." It is its own route from open-source compromise to executing rendering-surface code, and a high-leverage one because it skips the build step entirely.

9.6: No Verification at the Client Runtime Boundary

Open-source supply-chain defenses are valuable. Signed packages keep published packages traceable to their publishers. Registry security closes the credential-takeover route into the most-used registries. Software bills of materials produce a manifest-level inventory of what an application asked for. Dependency-scanning tools surface known vulnerabilities in declared packages. Vulnerability databases (CVE, GHSA, OSV, ecosystem-specific advisories) carry the public record of known-bad versions. These controls reduce the attack surface meaningfully.

Every one of those controls operates upstream of the build. Those controls end where the bundle leaves the build. The Client Runtime Boundary, where the bundle arrives and execution begins, is where the upstream regime stops. At that boundary the runtime accepts the bundle, accepts whatever the bundle loads at runtime through the Manifest Illusion, and grants the loaded code the host application's permissions under Runtime Anarchy. None of the upstream controls reach inside that sequence.

Defenses needed at the Client Runtime Boundary do not exist as default capabilities of any major host runtime. A single-source-of-truth inventory of the runtime resources actually loaded would let the runtime compare what it just executed against what it expected to execute. Content-swap detection at runtime would flag a fetched file that did not match what the application owner intended. In-memory anti-manipulation would defend the executing code from runtime tampering by code that has already been admitted. None of these is a default capability in browsers, mobile-app runtimes, desktop shells, vehicle infotainment systems, IoT or OT dashboards, AR/VR runtimes, productivity-tool extension hosts, or game and user-generated-content interactive runtimes. The defensive primitives that exist (Subresource Integrity in the browser, code-signing checks at app-store load, sandboxing where it applies) are partial and platform-bound.

Malice Without Malware applies in its open-source-specific form. The same gap appears across the language, provider, and AI dimensions, each from a different angle. What they share is the Integrity Blindspot at the load-to-execute boundary. The flaw is architectural and endemic to the pattern, so it appears on its own in every runtime built this way. Open-source supply-chain defenses are necessary and worth deploying. They stop at the Client Runtime Boundary, where the bundle becomes executing code.

Chapter 10: AI in the Client Runtime

AI is not an adjacent technology to ATG. It makes the same architectural pattern worse. The model is a runtime resource. The inference response, the answer that comes back when an application calls a model, is a data endpoint. Both reach Client Runtimes that already carry the Just-in-Time App Blueprint, Runtime Anarchy, and the Client Runtime Boundary: the moment fetched code begins to execute. AI does not introduce a new architecture for ATG. It compounds the one already in place, with properties that make detection harder and consequences larger than the non-AI cases.

AI in the Client Runtime is not a browser concern. Browsers run on-device inference through WebGPU, ONNX Runtime Web, and transformers.js. Mobile applications run models through CoreML, TensorFlow Lite, and MLC. Desktop shells that wrap a web runtime (Electron, Tauri) bundle local inference; productivity tools host model-driven features inside their plugin runtimes; vehicle infotainment systems and IoT dashboards increasingly carry on-device inference; AR, VR, and spatial runtimes are following the same trajectory. Every one of these host platforms also calls inference APIs: the addresses an application calls to get an answer from a model. Wherever a Client Runtime already runs, AI is moving inside it.

Two paths carry AI into the runtime, and they instantiate different ATG mechanics. Local model execution loads a model file into the Client Runtime the same way the runtime loads any other fetched resource, and it carries the same trust gaps; compromise of the model is compromise of the runtime that loads it. Inference API consumption calls a remote inference endpoint and acts on the response; the response is the second stage of the Two-Stage Attack, with the model in the loop and per-target personalization built into the output.

10.1: Why AI Matters for ATG

AI puts a new kind of component, the model, into the architecture already in place. Models load like scripts; inference responses arrive like data; the runtime acts on both. The two paths into the Client Runtime (local model execution and inference API consumption) instantiate ATG mechanics already named. What changes is what is loaded and what comes back: the response is shaped for one target, the model's behavior is statistical and emergent rather than written down anywhere, and the consequences build on the static-content cases.

The ML supply chain risks are well documented: model-file format issues including pickle execution; model repository trust across HuggingFace, Civitai, ModelScope, and Replicate;

namespace hijacking and typosquatting in model distribution; and training-data poisoning research.

When AI risks meet the Client Runtime, the picture changes. A poisoned model sitting in a public registry is one thing; the same model loaded into a vehicle's infotainment runtime, a productivity tool's plugin host, or a mobile banking application is another, because the host application now acts on whatever that model produces. From there, the path through the runtime is the ATG path; what is new along it comes from the loaded thing being a model rather than a script.

10.2: The Path from Model to Client Runtime

The two paths share little beyond the destination. A local model is a runtime resource: a file the host fetches and executes through a model framework. An inference response is a data endpoint: bytes the host receives from a remote service and acts on. What is loaded, why the runtime trusts it, how it fails, and what would defend it all differ between the two.

Local model execution loads a model file into the Client Runtime the way it loads any other fetched resource. Whether the runtime is a browser running ONNX Runtime Web, a mobile application running CoreML or TensorFlow Lite, an Electron or Tauri shell running bundled inference, a vehicle infotainment system running an embedded engine, or a productivity-tool extension running an in-app model, the thing being loaded is the same kind of thing and the trust gaps are the same. The Manifest Illusion applies: the runtime fetches and executes without inspecting what behavior the model will produce. The model file behaves like any other runtime resource the host trusts. Compromise of the model is compromise of the runtime that loads it.

Inference API consumption calls a remote endpoint from the Client Runtime and acts on the response. This is Supply Chain Injection at the data endpoint; the response is the second stage of the Two-Stage Attack, with the model in the loop and per-target context shaping what comes back. Whether the calling runtime is a browser, a mobile app, an Electron shell, a vehicle infotainment system, or a productivity tool, the structure is the same: trusted endpoint, trusted response, runtime acting on whatever the response contains. Compromise of the inference path reaches every Client Runtime that calls it, regardless of which host platform that runtime runs on.

Prompt injection is the dominant technique on this path: content arriving in the model's input gets read as an instruction rather than as data, so whoever controls the input controls the answer. Direct prompt injection places adversarial content in a user-supplied input the model reads and acts on; the second stage delivers behavior the application owner did not intend. Indirect prompt injection places adversarial content in retrieval sources, tool outputs, or memory the model will later ingest; the second stage activates at

consumption rather than at submission, decoupling the adversary's act of placement from the moment of effect. Either variant turns the inference path into a chain of data endpoints. The endpoint and the model can both be uncompromised. What is compromised is what the model reads, and the runtime carries no signal that the model's behavior has been redirected.

The two paths need different defenses. Local-model loading is dependency loading at the model layer; a single-source-of-truth inventory of runtime resources covers the model file as one more resource. The inference-API path is harder. A clean endpoint can return a compromised response, and prompt injection produces compromised responses without ever touching the endpoint. No major host runtime carries the defenses either path needs at the AI boundary today.

10.3: The Inference Response as Data Endpoint

The inference response is a data endpoint. The visible application code is benign; the operative content arrives in the response. The Two-Stage Attack describes the same structure for static content. AI inference adds three properties the static case did not have. The pattern holds across every host platform a Client Runtime runs on.

- **Per-target personalization:** inference produces output tailored per-request, per-context, per-target. The same code path returns different content to different users. Detection, audit, and comparison across a population of users all weaken when every user is shown something different. This is where AI amplifies the problem. A data endpoint that returns different content to every user is what the easiest, most rewarding, and most dangerous ATG attack mode requires: information manipulation.
- **Action endpoints:** where a model is allowed to act on its own, the response carries tool calls, function invocations, and instructions the host runs. The data endpoint becomes an action endpoint, and the Two-Stage Attack's second stage reaches beyond what is rendered into what the host does.
- **No validation against intent:** the runtime knows the response arrived. It does not know whether the response is what the application owner would have wanted the model to produce. There is no model of intent the runtime evaluates against, and no signal in the response itself that intent has been violated. Adjacent engineering work has produced partial mechanisms: structured-output enforcement against JSON schemas, deterministic policy frameworks such as NeMo Guardrails, constrained-decoding libraries, and tool-call signature validation. Each binds a narrow property (schema conformance, topic adherence, output shape) without addressing what the application

owner would have wanted the model to produce in the operative case. The gap holds: each of these measures narrows it without closing it.

The three properties compound. A response personalized to the target, encoding actions in its content, with no runtime mechanism to validate it against intent: none of these existed in the static-content case.

10.4: The Model as Compositional Surface

A model in production is rarely just the base model. Base weights load first. LoRA adapters, small extra weight files that adjust a base model without retraining it, specialize behavior. System prompts shape the model's defaults. User prompts arrive at runtime. Tool definitions let the model act on its own by calling other software. Retrieval-augmented context supplies additional knowledge during inference. Each layer is a separate source, often delivered separately, each with its own trust assumptions. The Just-in-Time App Blueprint returns at the model layer.

- **Each of those layers is a P&A position:** the base weights, the LoRA adapter, the system prompt, the tool definition, and the retrieval source are each a place an adversary can take up position and change what the model produces at runtime. The model's behavior is not produced from one source; placement at any layer reaches the response.
- **The layers combine in ways no one can predict from the parts:** a small change in any one of them can produce a large change in what the composed model does. Detection through code analysis does not transfer; the behavior is statistical and emergent, and the defensive question is what the composed model produces under varied inputs rather than what the loaded model file "is."
- **The pickle problem is one instance:** the well-documented pickle-execution issue at the model file level is one case of a problem that recurs at every layer that goes into an inference.
- **Foreign jurisdictional risk is a compositional concern:** a model trained or hosted under a government that can compel cooperation with its intelligence services becomes one of these layers, and its behavior cannot be presumed neutral. The jurisdictional thread reaches the runtime through whatever layer carries the affected source, and Ownership Opacity at any layer means the defender cannot tell who holds placement there.

Inside a composed model, several resources interact at inference time. Each is a P&A position whose trust assumptions the runtime inherits. The Just-in-Time App Blueprint at

the application layer was about how the bundle is assembled. At the model layer, it is about how a single inference is assembled: from weights, adapters, prompts, tool definitions, and retrieved text, each arriving separately.

10.5: Local AI and the Detection Blind Spot

When the model runs on the device, the adversarial intelligence work happens inside the runtime with no outbound traffic for any external system to inspect. The model loads once, inference runs locally, and manipulation happens inside the runtime. Network monitoring, the dominant defensive tool for behavioral signal, sees none of it.

- **No network fingerprint for inference:** detection architectures built around network monitoring miss local-AI-driven operations. The traffic that produced model behavior happened at load time, often once, and the inference that uses the model produces no traffic at all. The blind spot holds across host platforms; it is not a property of the browser, the mobile app, the desktop shell, the vehicle infotainment system, or the productivity tool. Wherever inference runs locally, the network-monitoring layer is silent.
- **Model behavior is harder to characterize than code behavior:** static analysis of code can enumerate what the code might do. Static analysis of a model's weights cannot enumerate what the model will produce. The behavior is statistical and emergent, and the defensive question shifts from inspecting the file to watching what the model produces under varied inputs. The instruments that exist for code do not exist for models.

The detection blind spot is architectural, not an oversight in any specific runtime. Network-era instruments do not reach into it.

10.6: Tailored Deception

Tailored Deception is the offensive capability that exploits the local-AI blind spot: generative AI in the runtime produces substituted content per target, on-device, at machine speed, with no shared population to detect against. AI's content-generation reach now spans text, image, audio, video, voice, and code, and it can produce content for a single target using whatever context the device makes available about that target. Three properties combine to turn this from a marketing feature into an ATG-class capability.

Per-target tailoring at machine speed eliminates the production bottleneck that previously gated precision deception. Content shaped to one user's data, history, social graph, behavior, and communication style once took a human analyst working one target at a time; a model now produces it in the time it takes to answer a prompt. On-device execution

moves the tailoring inside the runtime, beyond the reach of network observation. The deception material is generated where it will be consumed.

The third property reaches into the device itself: the AI executing inside the Client Runtime inherits the runtime's access to local resources. Files, documents, browser history, contacts, calendar entries, geolocation, sensor data, and application state are reachable through the same APIs the host application uses. The context the model tailors against does not have to be handed to it. The model can gather it, on demand, from wherever the runtime is permitted to read. Runtime Anarchy reaches the AI layer here: the AI executes with the host's permissions, and per-target tailoring draws on the host's view of the target. Tailored at machine speed, generated on-device, fed by direct access to the device: each property compounds the others, and the combination is what produces a different problem than precision deception or local AI alone.

- **Deepfakes on device:** on-device voice synthesis can clone a contact's voice for a real-time call placed through the device. On-device video synthesis can impersonate a participant in a video call rendered through the device's runtime layer. On-device image generation can fabricate documents, evidence, or communications that appear inside the user's familiar runtime context. The Hong Kong $25 million video-conference fraud in early 2024 shows the eight-figure stakes of deepfake deception; that case used attacker-generated deepfakes presented on a call rather than on-device synthesis inside the victim's runtime, so it evidences the consequence rather than the on-device mechanism.
- **Tailored persuasion content:** phishing messages, social-engineering scripts, fake notifications, fabricated context, manipulated documents, and plausible inbox messages can be produced per-target through any host runtime the user interacts with: a browser, a mobile messaging app, a desktop email client, a productivity tool.
- **Adaptive deception where the model is allowed to act on its own:** an AI agent generates content for each interaction that adapts to the user's responses in real time. Every interaction produces unique content. Static signature detection has no purchase because there is no static signature.

Detecting deception that is generated on the user's device and shaped for that one user is a different problem from detecting static deception or server-generated AI content. The defensive techniques that have evolved against email spam, fraudulent web pages, and templated phishing depend on a population of deceptive instances that share signal: text overlap, image hashes, infrastructure correlations, behavioral patterns. Locally generated, per-target content shares no such signal across users. Each instance is a population of one. Existing approaches do not transfer.

Older deception cases did not reach into the user's trust in their own device. Tailored Deception does. When the user's own device produces convincing deceptive content through legitimate runtime APIs, the user's trust in their own device becomes part of the attack surface. *This is* trust-of-runtime erosion. The user has been taught that the device is the trustworthy edge. The voice on the call is the contact's voice, generated by the trusted device. The face on the video is the colleague's face, rendered through the trusted client. The message is the message the trusted application showed. The user has no working heuristic for distrusting any of it, and the attack surface includes the user's confidence in their own runtime.

Information manipulation is the easiest to execute, the most rewarding to run, and the most dangerous ATG attack mode. Tailored Deception is the AI-amplified instance, delivered at population scale through the user's device.

Each instance is unique to the target, assembled inside the device, running in memory. On-device generation leaves no signal for network monitoring or population-scale signature detection: there is no shared population to compare against and no common signature to match across users. The model file and the in-memory state remain forensically recoverable on the device itself. This is, in effect, a unique deception per user, manufactured on demand, with the user's own device as the manufacturing line.

10.7: Deceiving the Model

Tailored Deception treats AI as the adversary's instrument. The model is also a target, and the Deceive effect applies to it directly. A model does not evaluate whether its input is true. It produces output consistent with what it was given, which makes deception of a model a matter of controlling what reaches it rather than of defeating any judgment it makes.

Deception of a model happens at two different timescales. At training time, corpus poisoning and disinformation channels feed the material a model learns from, so that the model states falsehoods as fact to everyone who asks. This is already happening. At runtime, the inference response can be substituted, retrieval and context can be poisoned, and instructions can be carried in content an agent reads on the Glass. ATG's specific contribution is the second: the Glass is where a model's inputs and outputs are assembled, and an agent operating in a Client Runtime consumes whatever the runtime hands it, with the same absence of verification that applies to every other resource.

The defensive position is worse than for a human consumer. A person can, in principle, notice that a figure looks wrong. An automated consumer has no such capacity, executes at machine speed, and produces output that downstream systems and people then treat as the product of analysis rather than as the echo of a substituted input.

10.8: No Defense at the Client Runtime AI Boundary

Existing ML supply chain defenses operate at the registry and build layers. Model signing, repository security, model scanning, and records of where training data came from each reduce the attack surface meaningfully and remain necessary defensive work. None reach the Client Runtime AI boundary.

At the Client Runtime AI boundary, the host runtime accepts the model (or accepts the inference response), runs inference, and acts on the output. Each of those steps is downstream of the defenses just named. A model can arrive here signed, scanned, and documented back to its training data; everything after this boundary is the runtime's problem, and the runtime carries no built-in instrument for any of it.

- **A single-source-of-truth inventory of the model resources actually loaded:** the runtime would have to know which model files, LoRA adapters, system prompts, tool definitions, and retrieval sources are loaded into the inference path at any moment, and verify them against an authoritative inventory. The same single-source-of-truth pattern used at the language and provider layers extends here.
- **Content-validation against intent for inference responses:** the runtime would have to test the inference response against the application's declared intent for the call before acting on it. Today the runtime knows the response arrived; it does not know whether the response is the response the application owner would have wanted the model to produce. Schema enforcement, policy-layer guardrails, and constrained decoding cover narrow slices of this requirement and do not reach intent validation.
- **In-memory anti-manipulation for model behavior:** the runtime would have to monitor model behavior under varied inputs, in memory, against an expected behavioral envelope. Static analysis of the loaded weights cannot do this; behavioral monitoring at the runtime layer can.
- **Signature-independent detection of Tailored Deception:** the runtime would have to detect locally produced, per-target deception material without depending on shared signatures across users. Because each user receives material no other user receives, there is no shared signal to match against; the requirement is an instrument that does not need one.

The gap is the same across host platforms. Browser, mobile app, desktop shell, vehicle infotainment system, IoT and operational technology (industrial control) dashboard, productivity tool: none of them ship with these capabilities at the AI boundary today. The architectural requirement is uniform across the landscape, even where implementation differs.

This is the AI-specific expression of Malice Without Malware. The same architectural shape repeats from the language, provider, and open-source defense gaps: existing controls operate where they have always operated, the runtime boundary remains unprotected, and the gap is structural rather than operational. What is distinctive at the AI boundary is the fourth required defense: signature-independent detection of locally generated, per-target deception. The other layers do not need it.

Part III - The Attack Mechanism

An ATG operation requires a position in the supply chain: a point that content, code, configuration, or data passes through on its way to a device. The adversary has to hold that point without the applications and users depending on it noticing.

Placement and Access (P&A) is the condition any ATG operation requires: an actor holds a spot from which what a device receives can be changed. An adversary reaches that spot by one of seven acquisition pathways, can occupy it anywhere across the APA Vector Landscape, turns holding it into effect along the P&A → Influence → Control chain, and can take it through any of the surfaces in the cross-surface attack model.

Ownership Opacity is the condition that a defender usually cannot find out who owns or controls a piece of digital infrastructure, because nothing public records it. Adversaries work inside that condition through Adversarial Asset Cycling, and four missing ownership-tracking capabilities keep it in place.

Chapter 11: Placement and Access (P&A)

Every ATG operation begins with a position. An adversary who has not acquired a node from which content, code, configuration, or data reaches a Client Runtime cannot run the operation; an adversary who has acquired one can run it at the moment of choosing. That position is Placement and Access (P&A).

The position is platform-agnostic. A browser tab, a mobile WebView, a mobile app shell built on a cross-platform framework such as React Native or Flutter, a desktop application built on a web runtime such as Electron or Tauri, a developer IDE, an AI chatbot interface, a smart-television runtime, a vehicle infotainment system, an IoT or industrial control dashboard, an AR/VR scene, a productivity-tool plugin host, an interactive UGC runtime: in every one of them an adversary can take the same kinds of position, and in every one the switch from a dormant position to a live operation works the same way.

11.1: P&A Defined and the Causal Chain

Placement and Access (P&A) is the structural condition of occupying a position from which content, code, data, or configuration delivered to a target can be controlled, modified, or substituted. The position does not depend on any particular exploit or technique: whoever holds it can act on what gets delivered, at a time of their choosing, whether or not they ever do. An exploit, when one comes, is simply the position being used; without the position there is nothing to use.

The position takes any of five forms:

- **Technical:** control of an infrastructure component the supply chain depends on, such as a CDN node, a DNS server, a build system, or a runtime endpoint.
- **Corporate:** control of an entity that owns or operates such a component, including the parent company, the holding entity, or the acquired subsidiary.
- **Human:** insider access by a maintainer, administrator, or operator holding legitimate credentials.
- **Physical:** location-based access to hardware, cabling, or facilities.
- **Coercive:** compelled cooperation from an actor who would otherwise resist.

Sophisticated operations will layer and combine multiple P&A positions. A position turns into an effect in three stages, each one a step up from the one before:

- **Placement and Access:** the actor occupies the position and can operate it. Holding the position is not the same as using it: it can sit dormant for any length of time before anything a defender could observe happens.
- **Influence:** what builds up as the actor uses the position repeatedly and across more of what passes through it. The systems fed from that position, the content they show, and the decisions people make on the strength of that content all start to move the way the actor wants.
- **Control:** Influence held over a whole population long enough that the actor's choices decide how the systems behave for the dependent set, every application and user relying on that infrastructure, rather than merely nudging that behavior.

How much Control an actor can reach depends on how good the position is and how long the actor keeps using it. A weak position exercised briefly produces transient Influence at most. A strong position exercised consistently produces Control over the dependent population.

11.2: Levels of Control

Control rises with the quality of the position that produces it. Five levels run from a one-off effect on a single request up to control of infrastructure that many systems sit on:

- **Transient influence over a single request:** the position influences one interaction. A network-path adversary who alters one response and then loses position has acted at this level.
- **Persistent influence over a defined user population:** the position consistently shapes outcomes for a population that depends on the affected resource. A compromised analytics script delivered through a CDN reaches every page of every property using that CDN, every load.
- **Selective control over content delivered to targeted individuals:** the position lets the actor change what one chosen person receives, provided the actor knows enough about that person to pick them out and reach them. An operation that ships different content based on geographic IP, device fingerprint, or authenticated user identifier acts at this level.
- **Full control over the information environment of a dependent system:** the position shapes everything the dependent system sees and processes. An adversary holding a vehicle's over-the-air update channel, the path by which new code is pushed straight to the car without passing through a dealer or an app store, decides what the head unit displays, requests, and accepts.

- **Systemic control over the infrastructure layer on which many dependent systems rely:** the position affects an entire infrastructure category and every system that depends on it. An adversary at a top-tier app store, a major mobile carrier's update channel, or a TLS root CA reaches the population of every dependent application.

Reaching each level takes a position of matching quality, and how far the adversary actually pushes it is their own choice. A CDN-level position enables application-scoped D5 effects; a device-level position enables cross-application D5 effects; an ISP-level position enables network-wide D5 effects; a platform-level position enables ecosystem-wide D5 effects. The position determines what is reachable; the operator decides whether to reach for it.

11.3: P&A as Injection Point in Supply Chains

Every P&A position established within a supply chain is an injection point. The injection point comes with the position itself rather than with any exploit; it lasts as long as the position lasts, and it is open to whoever holds the position, whatever technique they choose.

Seeing the position this way changes what a defender should watch for. A defender watching for exploit signatures is monitoring an Adversarial Influence Operation (AIO), the activation phase, rather than Adversarial Placement and Access (APA), the underlying capability. By the time a signature appears, the position has already been held long enough to deliver. The position is the threat; the activation is one expression of it.

11.4: The Influence and Control Monitoring Imperative

Defenders must continuously know who influences or controls each piece of infrastructure delivering content, code, or data to the rendering layer. Who owns it, who governs it, who staffs it, which legal jurisdictions it answers to, and who runs it day to day all count, because any one of them can be used to change what gets delivered. Whatever a defender does not watch is where the position can be taken or handed on unseen.

This is what Malice Without Malware means in practice for the supply chain: an adversary can change what a device receives without ever shipping code a scanner would call malicious. Finding malicious code is necessary but not enough. The broader job is knowing who influences and controls the supply chain.

11.5: APA and AIO

Adversarial Placement and Access (APA) names the case where the actor holding a P&A position is hostile to the dependent set. Adversarial Influence Operation (AIO) names the activity by which an APA holder uses the position against that set. Holding a position and

acting through it are separate things: an adversary can do the first for years without ever doing the second.

Many P&A positions exist legitimately. CDN operators, package maintainers, infrastructure providers, registry administrators, and credential authorities all hold P&A by design; the position is what makes their service possible. The position is the same in both cases; what differs is the holder.

An AIO selects targets, delivers content, code, or behavior through the controlled node, and manages duration, reach, and detectability. Running an AIO is what an APA position is held for; until one is run, the position is capability sitting idle.

Distinguishing APA from AIO matters for policy, detection, and response. The defender may observe an AIO without knowing about the APA that produced it. The defender may identify an APA before any AIO has been launched. The two situations are not interchangeable; the defenses that fit one do not fit the other.

11.6: The Activation Event

The activation event is the moment dormant APA produces a detectable effect. Until then the position gives a defender nothing to find: the traffic, the logs, and the behavior all look the same as a legitimate operator running the same infrastructure. Activation is typically the earliest moment at which a defender can observe the operation.

Policy and architecture that focus only on the activation event miss the APA window entirely. By the time activation is observable, the position has already been held long enough to gather intelligence, prepare payloads, and shape the infrastructure for the operation to come. Rules that make an organization track who owns and controls its infrastructure, before anything has been launched through it, act on the APA phase instead.

11.7: The Scope Hierarchy

P&A operates at five scopes:

- **Component:** a single script, package, or library.
- **Application:** an entire application's delivery infrastructure.
- **Platform:** a CDN, cloud provider, app store, OTA update channel, plugin marketplace, or distribution channel serving many applications across one or more host Client Runtime platforms.
- **Device:** a position on the target device itself: laptop, phone, vehicle head unit, smart-TV, IoT controller, or an embedded control panel.

- **Infrastructure:** a position in the network path between server and client.

Each level widens what an adversary can reach, and a position at one level helps win positions at the others. An adversary with P&A at one level uses it to achieve P&A at additional levels: a supply-chain position that profiles devices enables targeted device-level P&A; a device-level position that observes credentials enables platform-level P&A. The Polyfill.io operation ran this pattern in confirmed form, using platform-level CDN access to profile visiting devices and selectively redirect a subset to an exploit site. Each position creates intelligence that enables the next; a sustained operation accumulates positions across multiple levels rather than holding one.

11.8: The APA Vector Landscape

A vector is a position an adversary can occupy. A pathway is a route by which the adversary gets to it. The two are independent of each other: any position can be reached by any route, and most real campaigns use several of each. The APA Vector Landscape spans nineteen categories, organized by infrastructure layer:

- **Foundation and platform layer:** hardware, firmware, OS and kernels, hypervisors and virtualization, container runtimes.
- **Network and transit layer:** the largest backbone carriers, mobile carriers and SIM/eSIM providers, the internet exchange points where networks hand traffic to each other, BGP and routing, VPN and proxy services, Wi-Fi infrastructure, web application firewalls, DDoS mitigation, edge compute.
- **DNS, naming, and trust layer:** registrars, authoritative DNS, recursive resolvers, TLS certificate authorities, certificate transparency logs, cryptographic libraries, hardware security modules and key management services, secrets managers.
- **Cloud and hosting layer:** IaaS, PaaS, and SaaS providers; managed Kubernetes; serverless platforms; database-as-a-service; managed storage and object stores.
- **Code, build, and software distribution layer:** package registries, source repositories, CI/CD pipelines, container image registries, code-signing infrastructure, software-update servers.
- **Identity and access layer:** identity providers, SSO and federation services (SAML, OIDC), OAuth providers, MFA services, directory services, privileged access management.

- **Web application front-end supply chain (the IOTG layer):** CDN-hosted libraries, third-party scripts, tag managers (the tools that load other companies' scripts into a page), ad and analytics SDKs, font and asset hosts, embedded widgets.
- **Embedded third-party services and widgets:** chat and support widgets, video players, social-media embeds, comment platforms, payment widgets.
- **Payments and commerce:** payment processors, gateway providers, tokenization services, fraud-prevention vendors, e-commerce platforms.
- **Marketing, advertising, and tracking ecosystem:** ad networks, demand-side and supply-side platforms, audience platforms, RTB exchanges, attribution and analytics platforms.
- **Data and reference services:** weather, location, mapping, currency, market-data, and reference APIs; data brokers.
- **AI and ML supply chain:** model registries, model providers, inference APIs (the addresses an application calls to get an answer from a model), training-data sources, MLOps pipelines, vector databases.
- **Communications infrastructure:** SMS and email gateways, push-notification services, voice and video providers, messaging APIs, real-time communication platforms.
- **Mobile-specific supply chain:** app stores, in-app purchase platforms, mobile SDK vendors, mobile-specific analytics, OTA hot-update services.
- **Endpoint security and management:** endpoint detection and response vendors, mobile device management platforms, antivirus, endpoint configuration tools, vulnerability scanners.
- **Operations, observability, and infrastructure tooling:** application performance monitoring, logging, monitoring, error reporting, feature-flag services, configuration management.
- **Standards, governance, and registry layer:** standards bodies, certificate authorities (root and intermediate), domain registries, technical-spec governance.
- **Adjacent service ecosystems with privileged access:** enterprise SaaS with administrative rights, including CRM, ITSM, document management, and productivity suites.

- **IoT and operational technology supply chain:** device firmware vendors, OTA update channels, telemetry and control platforms, industrial control systems, smart-home platforms.

The supply chain really does run this wide. Every one of these categories holds positions that become injection points once an adversary occupies them. Few defenders can see into all nineteen, and adversaries pick the ones nobody is watching.

11.9: P&A Acquisition Pathways

Adversaries acquire P&A through seven pathways. None of them is new to supply-chain security; what changes with ATG is what each one buys the adversary at the rendering layer:

- **Technical infrastructure control:** CDN, DNS, hosting, ISP, TLS CA, Wi-Fi network compromise.
- **Software supply-chain control:** package takeover, third-party script hosting, data endpoint compromise, CI/CD pipeline, code repository, container or image registry, AI model supply chain, browser extension acquisition.
- **Corporate and organizational control:** acquisition of entity, board or governance influence, investment leverage, vendor and contractor relationships, joint venture access, standards body influence. Because buying a company is a lawful transaction, this route can be taken with a signature rather than an exploit; the Funnull/Polyfill.io acquisition is the clearest case.
- **Human factors:** insider placement, insider recruitment, developer or maintainer targeting, social engineering of access processes, third-party personnel with infrastructure access. The Open-Source Open Door is the version of this that costs an adversary the least: an unpaid maintainer, with no security team behind them, holding publishing rights over code that runs everywhere.
- **Physical access:** data center, network infrastructure, hardware supply chain interception, endpoint device access.
- **Coercion, extortion, and hostile action:** extortion of operators, coerced insider action, jurisdictional coercion, criminal acquisition through fronts, physical intimidation, blackmail. Corporate acquisition and coercion combine readily: control of what an operator's infrastructure delivers passes with the asset, whether the transfer was bought or forced.

- **Communication and distribution channels:** app stores, update and patch distribution, email and messaging infrastructure, content delivery and caching infrastructure.

Pathways combine. Most documented operations use two or more, and the choice depends on the defenses the adversary faces.

11.10: Operational Characteristics

An APA position behaves in three ways worth naming separately. Dormancy, persistence, and escalation are well-documented adversary tradecraft generally; each takes a particular form here:

- **Dormancy:** until the adversary acts, the position produces nothing a defender could flag. This is what defeats checking a file before it is delivered: at the moment of the check nothing is wrong, so the check passes, and the position is what defenders have to watch instead.
- **Persistence and resilience:** APA positions are designed to survive personnel changes, infrastructure migrations, security audits, and vendor reviews. Redundant APA across multiple vectors is the operational reality, not the exception.
- **Escalation:** each P&A position is both a capability in itself and the intelligence that makes the next one reachable. That is how APA spreads across the five levels of scope.

Sustained operations carry all three by design.

11.11: The Cross-Surface Attack Model

Every digital asset has more than a digital surface. It has a physical presence (a server in a building, a cable in a conduit, a device on a desk), a legal owner (a person, a corporation, or a trust), a financial structure (investors, creditors, acquirers), and a human operator (a developer, an administrator, or an executive).

An adversary can obtain P&A in the digital supply chain by attacking any of these surfaces, not only the digital one. Control taken through one of these surfaces is worth exactly as much as control taken through any other, because the end state is the same. Defending one surface does nothing for the rest, and an attack that lands on one carries into them.

Lateral movement across surfaces is the operational reality. Financial leverage leads to corporate control, then to technical access, then to digital P&A. Physical access leads to credential theft, then to infrastructure control, then to supply-chain position. Social engineering leads to insider recruitment, then to code repository access, then to deployed adversarial capability. The digital supply chain is reachable from all of them.

The same point can be stated as a rule about ownership. However abstract or widely distributed a digital supply-chain component looks, it runs on equipment that belongs to a company, and that company is run by people who live somewhere and answer to the law of that place. The adversary who can exert control over the person, the entity, or the jurisdiction can obtain control over the digital asset without executing any technical exploit against the digital asset itself.

P&A names what makes ATG operations possible. The position lives across five surfaces, not one. A defense that reaches only the digital surface reaches only a fraction of the threat.

Chapter 12: Ownership Opacity: The Visibility Problem

Every P&A position is held by someone, and the someone matters. To defend against an adversary sitting in a position, a defender has to notice that the position is held by an adversary, and that means knowing who holds it. The Influence and Control Monitoring Imperative is the operational form of that requirement. Without ownership visibility, the imperative cannot be operated.

Ownership visibility is what the current ecosystem does not provide. Digital infrastructure ownership is harder to determine from public information than physical-asset ownership, for reasons that have nothing to do with active adversary deception and everything to do with how the digital supply chain is organized. The conditions exist whether or not any adversary is exploiting them. They set the limit on what a defender can learn from public sources, however much effort goes into looking.

Adversaries work in the room that limit leaves them. Adversarial Asset Cycling is the operational pattern by which they hold position quietly, run through it, and replace under new infrastructure when attribution begins to close. The same cycle runs on every kind of asset the digital supply chain is built from: domains, packages, browser extensions, publisher accounts, update channels.

12.1: Why Ownership Matters for ATG

Every P&A position has owners at several layers at once, and an adversary needs only one of those layers to be unreadable in order to stay hidden. The Influence and Control Monitoring Imperative requires continuous understanding of who influences or controls each piece of infrastructure delivering content, code, and data to a Client Runtime. That understanding is operationally meaningful only if ownership at each layer can be determined. Where ownership cannot be determined, the imperative cannot be operated against the layer in question, and the position can be acquired or transferred without observation.

The cross-surface attack model comes down to one question asked at each layer: who owns this. Following a single request through the infrastructure shows where the layers are. A URL resolves to a host; the host is named by DNS; the DNS is controlled by a domain; the domain is owned by a party. Behind that technical chain sits a chain of owners: a financial owner who can sell it, a corporate owner who can make decisions about how it is operated, an operational owner who can deploy code or change configuration through it, and the personnel who hold the credentials that produce runtime effect. Each link can turn over without any change visible at the link above it. Each layer is a distinct ownership question.

Control acquired at any one of them is fungible with control acquired at any other; opacity at any one is sufficient to conceal the holding.

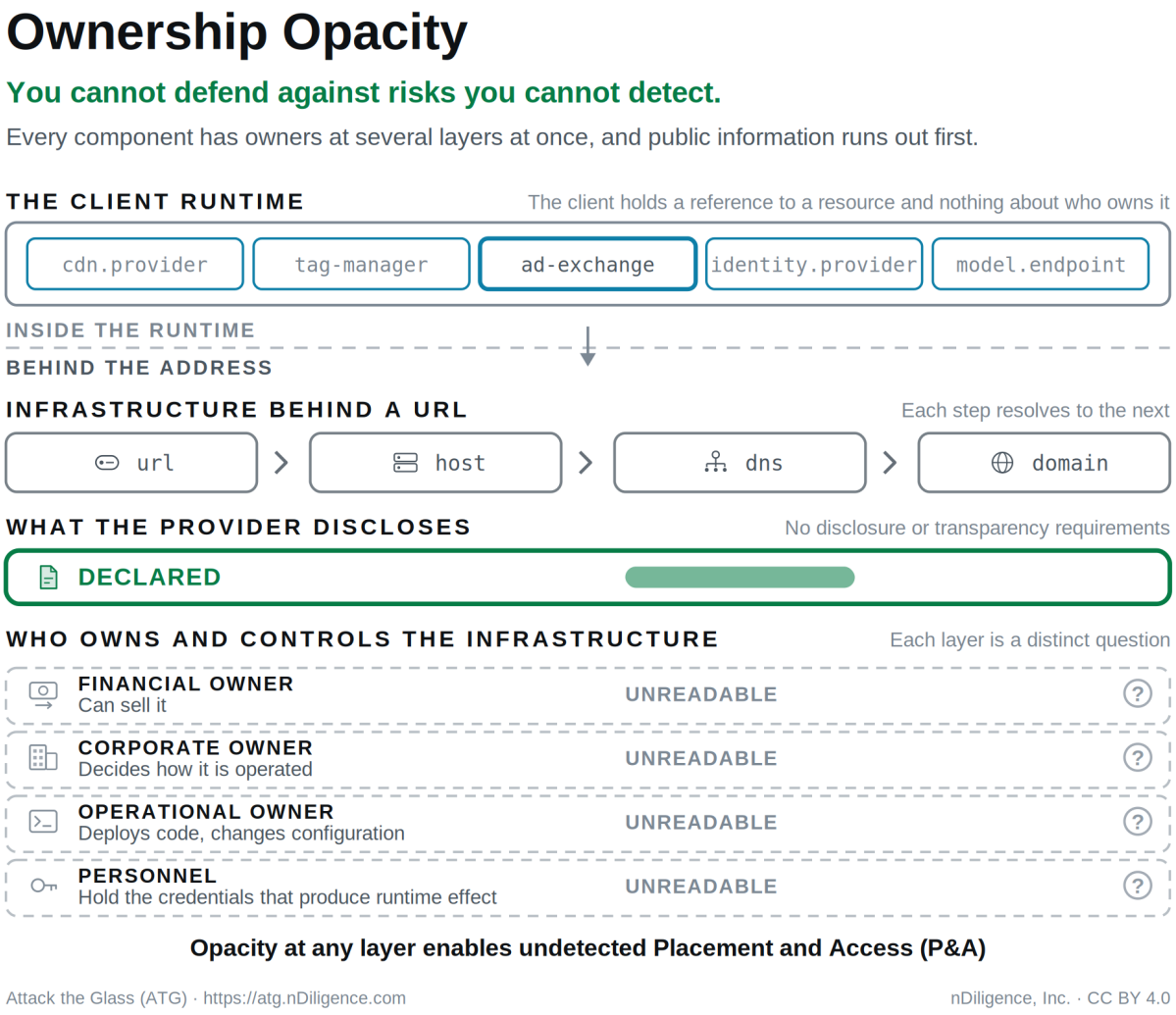

***Figure 28**: Four things a defender would need to know about who controls a provider. None of them resolvable.*

12.2: The Public-Information Floor

Familiar mechanisms of ownership opacity apply with full force to digital infrastructure. WHOIS privacy hides domain registration. Shell companies and nominee ownership obscure corporate beneficiaries. Rules that require a company to name the humans who ultimately own it apply only in the countries that have such rules, and only to the kinds of company those rules cover; an owner outside both is recorded nowhere. A chain of holding companies can be routed through countries chosen because they publish little about who owns what.

What makes the ownership problem distinctive in the ATG context is what is absent rather than what is present. For physical assets, public records let anyone check who holds title

to a building, who is registered as controlling a company, and where large sums of money moved, which puts a practical limit on how far ownership can be hidden. For the infrastructure that delivers code to a screen, no records of that kind exist.

- **Nothing works like a land registry.** Real property has public ownership records, so anyone can look up who holds title to a building and when it last changed hands. Nobody can look up who holds a software namespace, who runs a package repository, or who operates a given CDN, because no public record of any of it is kept. Ownership is whatever the operator chooses to say, and nothing obliges the operator to say anything.
- **Nothing works like securities disclosure.** A listed company must publish who its major shareholders are and who controls it, so an outsider can check the answer against a filing. Those rules stop at the boundary of the listed company. Most infrastructure that matters for ATG is run by private companies, by subsidiaries bought into larger holding structures, or by individual maintainers, and for all of them the filing that would answer the ownership question is never made.
- **Nothing works like anti-money-laundering reporting.** A bank has to report transfers above a set threshold, so a large movement of money leaves a trail an investigator can follow afterwards. Selling a CDN, handing a package to a new maintainer, changing who operates a script host, or transferring an extension or plugin publisher account sets off no report to anyone. The change of hands can complete without a single outside party learning that it happened.
- **Legal ownership and operational control are not the same thing.** Even where the legal owner can be identified, the people holding commit access, publishing credentials, authority over runtime configuration, or platform-administrator rights are frequently not the owners at all. An operator can replace those people without telling anyone outside, and whoever holds the credentials next can change what reaches a user's screen straight away.

The ways ownership gets hidden are the same in every industry. What is missing only here is anything on the other side of the ledger: no registry, no filing, no report an outsider could check the claim against. A defender working from public information has far less to work with for a CDN or a package than for a building or a bank account, and that shortfall follows from how the ecosystem is organized rather than from anyone's oversight.

The most important supply chain humanity has ever built has no way of telling anyone when the people controlling it change. The *Silent Transfer* is that change of hands with no technical signature: control moves through non-technical channels, a company sale, a credential compromise, a maintainer or namespace transfer, a DNS repoint, none of which

alter the URL, the hash, or the certificate. Everything a scanner measures stays green while the party deciding what the asset serves becomes someone else. The URL never changed. The person behind it did.

12.3: The Adversarial Asset Cycling Pattern

Adversarial Asset Cycling is a four-stage operational pattern. An adversary acquires an asset, runs a campaign through it, abandons it before sustained attribution can resolve, and re-emerges with new infrastructure that may be related to the abandoned operation but is not formally linked to it. The pattern works because the asset arrives with the trust and the installed dependents already attached, which is the part an adversary cannot manufacture. Building a new CDN, a new package, or a new extension from nothing takes years and rarely makes sense as an attack route. Buying one that already runs in millions of Client Runtimes takes days.

Acquisitions follow established commercial and operational routes. A small company that operates a CDN domain or a script-hosting service can be purchased as a going concern; the domain, the URLs, and the existing <script> tags pointing at it transfer with the company. A package maintainer position can be obtained by credential compromise of the maintainer, by social-engineering a namespace transfer, or by purchasing the maintainer's broader business. A browser extension publisher account can be bought from the original developer, who often welcomes the offer for a low-traffic but installed-base extension. Mobile-app developer accounts, OTA channel operators, plugin marketplace publisher accounts, and AI model namespace registrations each have commercial-acquisition or credential-compromise routes. None require constructing new trust; all inherit it.

Polyfill.io is the confirmed instance. The cdn.polyfill.io domain changed hands in 2024; existing <script> tags pointing at it continued fetching from the same URL afterward. The infrastructure changed ownership; the trust did not.

Each stage exploits a specific opacity. The acquisition is invisible because no public registry records ownership transfers of script-host domains, package maintainer positions, or publisher accounts, so there is nothing an outsider can look the change up in, the way a land record can be looked up. The operational period is hard to attribute because operational control diverges from legal ownership and the personnel running the asset are not publicly identified. The abandonment closes the trail through any of several routes: the operator can sell the asset to another shell, scrub the malicious version and resume benign operation, dissolve the operating entity, or simply walk away from the credentials. The re-emergence sends an investigator back to the beginning, because the new infrastructure is registered to a different operator and nothing on paper connects it to the abandoned one.

Detection of cycling requires ownership-tracking capabilities the current ecosystem does not provide. Threat-intelligence frameworks track operational tradecraft and infrastructure indicators; they track ownership only as far as public information reaches, which stops well short of the point where cycling happens. The ownership-tracking gap is the structural reason cycling is a viable adversary pattern.

12.4: The Ownership-Tracking Capability Gap

Four ownership-tracking capabilities would close the visibility gap. None is provided at the scale ATG defense requires. Together they would make the Influence and Control Monitoring Imperative operable.

- **Ownership-state monitoring:** continuous monitoring of who currently owns and operates each piece of ATG-relevant infrastructure: CDN operators, package maintainers, script hosts, AI model repositories, extension publishers, plugin marketplace publishers, mobile-app developer accounts, OTA channel operators. Today's threat-intelligence tools do not treat ownership as a signal in its own right. Where ownership shows up at all, it shows up as background detail on one incident, not as something recorded about the asset and kept current.
- **Ownership-change detection:** identification of ownership transitions (commercial acquisition, maintainer transfer, namespace transfer, infrastructure-operator handoff, publisher-account transfer) and propagation of those transitions to the dependent applications. No tool today tells an application that something it depends on has changed hands: a dependency carries a name and a version, and nothing at all about who now controls it.
- **Operational-control inference:** inference of operational control from observable signals (commit patterns, release cadence, infrastructure footprint, behavioral changes) when legal ownership cannot be determined. Today's tools keep these signals apart from any question about ownership, so nobody ends up with one answer to who is actually in control of the asset.
- **Cross-asset ownership correlation:** correlation of ownership signals across multiple infrastructure assets to identify common control, asset cycling, and adversary infrastructure clusters. Current tooling treats each asset in isolation; cycling that operates across assets cannot be detected without correlating ownership signals across them.

Across asset types, the gap is the same. The infrastructure that delivers to a Client Runtime spans browser-side providers (CDNs, script hosts, ad networks, analytics), mobile-side providers (app stores, OTA channels, in-app extension hosts), desktop-side providers

(distributors of desktop applications built on a web runtime such as Electron, plugin marketplaces), vehicle and IoT providers (OEM update channels, infotainment app stores), and AI providers (model repositories and inference endpoints, the addresses an application calls to get an answer from a model). The four capabilities apply uniformly. The defender that lacks them in one asset class lacks them everywhere the asset class touches a Client Runtime.

With the four capabilities in place, the visibility problem closes. Without them, the Influence and Control Monitoring Imperative names a requirement no defender can meet.

Part IV - Evidence and Existing Defenses

How much of the digital supply chain a defender can actually see, the historical record of confirmed operations, the defenses already deployed, and the cybersecurity frameworks the discipline works inside each bear on ATG, and each covers only part of it.

The confirmed historical record covers nine categories of ATG-class event documented in the public record. Taken together, they establish that ATG is operational, accelerating, and multi-vector.

The claims in this record carry different levels of support. The table below separates what is confirmed in the wild from what rests on research demonstration, structural inference, or a stated working presumption, so each assertion can be weighed on its own basis.

Claim or capability	**Support level**	**Basis**
Targeted content substitution through an acquired, trusted supply-chain node	Confirmed in the wild	Polyfill.io / Funnull (2024): 110,000+ sites; operator OFAC-sanctioned in 2025
Client-side script injection that reads and exfiltrates what the page renders	Confirmed in the wild	Magecart family (payment-card skimming)
Substitution of a third-party script by an actor holding the network path, delivered to selected requesters	Confirmed in the wild (adjacent: on-path placement, not supply-chain placement)	Great Cannon (2015): analytics and advertising scripts, served over unencrypted HTTP, answered with substituted JavaScript for a selected fraction of requesters; executed in bystander browsers outside China; attributed to the Chinese state by the Citizen Lab
Open-source maintainer takeover reaching downstream builds	Confirmed	event-stream and ua-parser-js (delivered); XZ Utils, 2024 (near-miss for reach)
Build-pipeline compromise with long operational dormancy	Confirmed (adjacent:	SolarWinds / SUNBURST (2020)

	backend, not the rendering layer)	
AI deepfake deception at eight-figure stakes	Confirmed capability (social engineering, not supply-chain injection)	Arup, Hong Kong, US$25 million (2024)
Demand and scale for cloning trusted news brands	Confirmed (adjacent: uses separate infrastructure)	Doppelganger, Russia-attributed, since 2022
Two-Stage Attack: a hash-valid script carries the attack in the runtime data it fetches, defeating static verification	Demonstrated in part; structural	Evasive payloads observed (the Polyfill payload activated by device and time); no single public end-to-end rendering-layer case
Endemic Flaw: browser defenses (CSP, SRI, Trusted Types) have no equivalents outside the browser	Structural inference	Absence of equivalent runtime defenses across mobile, desktop, OT, vehicle, AR/VR, and productivity runtimes
ATG operational at scale today, including well-resourced and nation-state use	Working presumption (stated as such)	Inference from architectural conditions, not an established finding
On-device, per-target tailored deception at population scale (weapons of mass deception)	Structural inference and working presumption	Extrapolation from confirmed deepfake capability and the CSR architecture

Four defenses are already deployed against this: Content Security Policy, Subresource Integrity, Zero Trust Architecture, and Server-Side Rendering. Each helps, and each stops short for reasons built into how it works. They converge on the Endemic Flaw: the weakness sits in the architecture itself, so it turns up in every runtime separately rather than spreading from one to the next.

ATG can be placed inside the frameworks the discipline already uses: CVE/NVD, MITRE ATT&CK, NIST SP 800-161, and the regulations that apply sector by sector. None of them holds it comfortably, and five gaps in the standards follow.

Chapter 13: The Confirmed Historical Record: Summary Reference

ATG is in operation. The public record contains confirmed events whose mechanisms instantiate the conditions the framework chapters develop, and over the past five years the record has grown: more cases observed, greater sophistication, wider reach across surfaces.

Three structural conclusions hold across the record: ATG is operational, accelerating, and multi-vector. The cases that ground each conclusion reach across the deployed Client Runtime population beyond the browser and are documented in public reporting.

13.1: What the Confirmed Record Establishes Analytically

The confirmed record establishes that ATG is operational, accelerating, and multi-vector. It is therefore not a forecast, not a steady state, and not a narrowly bounded technical issue.

- **Operational:** ATG-class operations have been observed in production at significant scale, with measurable consequences. The architecture's exploitability is not a theoretical inference from structural conditions; it is a documented observation across multiple cases. The threshold for operational is not "an attempt occurred" but "a campaign ran and produced effect." That threshold has been crossed.
- **Accelerating:** frequency, sophistication, and scope of confirmed cases have grown over the past five years. The trajectory is consistent with two structural drivers: the AI Amplification Factor lowers per-target cost, and the commercialization gray zone puts attack capability within reach of buyers who could not build it. The acceleration is not a function of better detection alone; new operational patterns appear in the record that earlier cases did not exhibit.
- **Multi-vector:** cases distribute across the surface landscape, the provider taxonomy, and the language and platform dimensions. They reach across Client Runtime host platforms: browser, mobile, desktop, productivity-tool host, and the interactive runtimes of games and user-generated content. No single defensive measure addresses the full set; cases that compromise a CDN sit beside cases that take over a package maintainer's account, cases that target a deployed model, and cases that abuse a user-generated content channel.

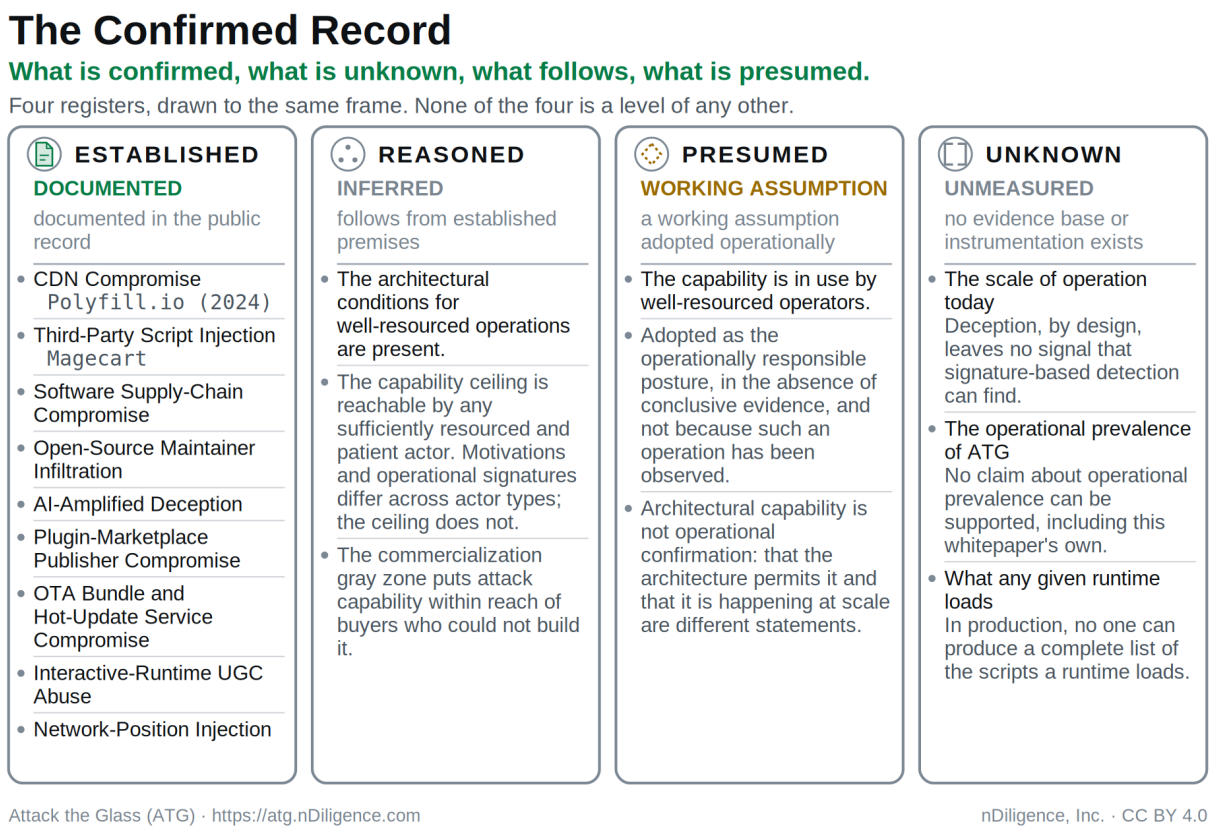

***Figure 29**: Four registers of certainty: established, reasoned, presumed, unknown. Every claim is filed against one.*

The cases collectively justify the cross-runtime architectural patterns. They are what one architectural pattern looks like when adversaries operate it.

13.2: ATG-Class Event Categories and Their Structural Significance

Nine categories of ATG-class events appear in the confirmed public record. Each pairs a structural mechanism anchored in the framework with cases documented in public reporting.

- **CDN Compromise:** Polyfill.io (2024) is the confirmed case. The CDN acquisition attack model operates at scale, and commercial acquisition is an observed pathway to Adversarial Placement and Access (APA), not a theoretical one.
- **Third-Party Script Injection:** the Magecart family is the most-documented confirmed expression, with the same shape recurring: a script injected into the page copies payment-card data and sends it to the adversary. Runtime Anarchy turns into real compromise here: the injected script reads what the host renders, and never has to breach the host's backend.

- **Software Supply-Chain Compromise:** SolarWinds-class operations are the confirmed expression, combining corporate and human-factors acquisition pathways with operational dormancy to escape detection for months. SolarWinds is a build-pipeline/backend compromise rather than a rendering-layer ATG event; it is included here as an adjacent boundary case that demonstrates build-pipeline compromise and operational dormancy, and it reaches Client Runtimes only indirectly, through the affected vendor's management consoles and tools.
- **Open-Source Maintainer Infiltration:** XZ Utils (near-miss), event-stream, and ua-parser-js are confirmed expressions, with long-game contributor infiltration shown to be feasible within community trust structures. The Open-Source Open Door is operationally exploited, and where the compromised dependency is one that ships in client bundles, it reaches every Client Runtime that bundles it. XZ Utils itself is a server-side systems library (liblzma/sshd) and is cited here for the maintainer-takeover pathway, not for reach to the client rendering surface.
- **AI-Amplified Deception:** the Hong Kong $25 million deepfake video-conference fraud against Arup (early 2024) is the most-cited AI-deception case of the period; note that Arup confirmed none of its systems were compromised, so the case demonstrates the deepfake capability an ATG operation would weaponize rather than the ATG delivery mechanism itself. Voice-cloning fraud is a corroborating operational pattern in the same period. Tailored Deception arrives at the user through whatever Client Runtime the user happens to be using: a browser-based meeting client, a mobile messaging app, a desktop conferencing client, a vehicle infotainment voice channel.
- **Plugin-Marketplace Publisher Compromise:** compromised browser extensions, IDE plugins, and creative-tool extensions are recurring entries in the public record. The path from compromise to execution is the most direct: nothing is rebuilt in between, and the plugin renders its own content on the client, alongside or inside its host application (browser, IDE, creative tool, productivity tool, mobile app extension).
- **OTA Bundle and Hot-Update Service Compromise:** mobile over-the-air bundle delivery channels, which push code updates from a server straight into an already-installed app without going back through the app store (CodePush, Capgo, Appflow, and similar services), are a recognized route by which code reaches an installed application without passing app-store review. NIST's Mobile Threat Catalogue records the category at APP-20: an application may evade vetting by downloading and executing code after installation, and code that was absent at submission cannot be caught by review. No public compromise of any named over-the-air service is on the record, so this category is carried as Reasoned rather than Established. Because the update is

silent, one compromised bundle reaches the entire installed base, and no store review sits in the way.

- **Interactive-Runtime UGC Abuse:** Roblox, Unity AssetBundle, and analogous UGC interactive runtimes carry the exposure, and the grading differs by platform. Roblox documents the problem itself, recording that third-party assets from its Creator Store are a common source of malicious scripts called backdoors, which is Established from the platform operator. The Unity claim is narrower than it has been stated: Unity's manual records that AssetBundles cannot contain executable code, and that altered serialized data may instead exploit the application or the runtime. Published research has demonstrated execution from a crafted asset using only default engine components, which is Reasoned from demonstration rather than Established from an in-the-wild case. The trust model for user-generated content is structurally weak against ATG-style payload injection: the host runtime is a Client Runtime in its own right, and content delivered into it reaches the rendering surface directly.
- **Network-Position Injection:** the Great Cannon (2015) is the confirmed example, and it is the only category here whose placement is not a supply-chain position. An actor holding the network path between the requester and the host substitutes the response instead of compromising the source. An on-path system can inject a response but cannot stop the real one, so it races; an in-path system suppresses the original request and answers in its place. The Great Cannon was in-path, which makes it a man-in-the-middle in the strict sense rather than an injector. The intercepted resources were served over unencrypted HTTP, and the researchers who documented the system named HTTPS as the remedy, so the operation as run in 2015 does not repeat unchanged against an encrypted delivery path. And the position is not exclusive to states: it is reachable by BGP hijack, by DNS hijack, by certificate-authority compromise, and by any operator of a transit link or middlebox, which is why network-path hijack already appears in the provider-landscape analysis as a provider compromise route. The other categories describe how an adversary comes to control a component. Controlling the path is sufficient without controlling any component at all.

Together the nine categories reach the full deployed Client Runtime population. Each has a distinct structural mechanism in the framework.

Chapter 14: Existing Defenses: Partial Measures and Structural Limits

Existing defenses do real work. Content Security Policy, Subresource Integrity, Zero Trust, and Server-Side Rendering each address a real component of the threat surface. Measured against ATG, each one shows a boundary where its protection stops. The boundary is not a matter of operator skill or implementation maturity. It is architectural.

Outside the browser, far fewer of these defenses exist. Mobile WebViews and hybrid runtimes, desktop application shells built on web technology (Electron, Tauri), vehicle infotainment, IoT and operational-technology dashboards, AR/VR scene graphs, productivity-tool extension hosts, and the path by which an application calls an inference endpoint (the address an application calls to get an answer from a model) all lack an equivalent of the browser primitives by default. Where browser-engine components are embedded inside non-browser hosts, the browser-class defenses can apply to the embedded surface; they do not extend to the surrounding runtime.

The architecture defeats integrity checks made against a file before it is delivered; the defense that complements them is watching how code behaves at the rendering layer while it runs. That defense exists today in concept and in narrow practice but is not yet a default capability of any Client Runtime in the deployed landscape.

Runtime behavioral monitoring is an open engineering problem. Inspecting objects the running code has changed, DOM nodes, output from tools a model is allowed to run on its own, and inference responses as fast as the runtime executes them costs real CPU, memory, and energy, against the same economic incentives that produced the Compute Tax. Partial expressions exist today in narrow forms: in-browser script-behavior telemetry shipped by client-side attack surface management vendors, runtime sandboxing layers that observe a subset of API calls, and policy layers that sit in front of a model and check its responses against fixed rules before an application uses them. None is yet a runtime-default capability with rendering-layer breadth across the Client Runtime landscape. The requirement is architectural: it has been named, the engineering path to building it into runtimes by default is open, and the fact that no runtime ships it today does not make the requirement go away.

14.1: The Structural Limits of Existing Defenses

Each condition is a property of the architectural pattern itself. No amount of operational discipline closes them.

- **Structural blindness in existing frameworks:** NIST CSF, ISO 27001, MITRE ATT&CK as a defensive vocabulary, and the broader compliance landscape were built against a threat model in which the application was what got reviewed, signed, and shipped. ATG

operates after that boundary, on the rendering surface the frameworks do not address. The frameworks are not wrong about the territory they cover. They say nothing about the territory the runtime occupies, and the program documents, audit checklists, and detection vocabularies built from them carry that same gap into daily operations.

- **The runtime code injection problem:** verification ends before execution. Every meaningful integrity check finishes before delivery, and the runtime executes afterward. Supply Chain Injection names the consequence: malice is delivered into the runtime through the same supply chain the runtime was designed to consume, and the delivery clears every pre-delivery control along the way. Existing defenses operate inside the pre-delivery window. The injection ATG enables happens after that window closes. The structural property holds across every CSR-capable Client Runtime in the deployed landscape (browser, mobile, desktop, vehicle, IoT, AR/VR, productivity-tool host, UGC interactive runtime), even where the pre-delivery defenses available to each host differ.
- **The scale problem:** defenders cannot enumerate what their applications load at runtime. The requirement to know your suppliers' suppliers at the provider boundary, the structural reasons no one can list the scripts a runtime loads, and the Manifest Illusion each name a different facet of the same observation. Existing defenses presume an inventory that operationally does not exist. The supply chain is too large for any defender to act on it item by item.

The frameworks that govern security programs do not see the runtime boundary. The verification regimes that govern delivery end at that boundary. The inventories that would let defenders see past that boundary cannot be assembled at the scale the runtime supply chain reaches. Each of the four major existing defenses runs into one or more of these three problems. The defenses do what they were designed to do. ATG operates outside what they were designed to defend.

14.2: Content Security Policy: Value and Limits

Content Security Policy raises attack cost meaningfully in the browser context. CSP declares an allowlist of URLs the browser runtime may fetch, restricts inline-script execution, and provides structured reporting on policy violations. Three architectural limits sit beyond what the policy constrains.

- **Allowlist coverage:** CSP constrains URL allowlists, not what those allowed URLs deliver. If a CDN on the allowlist is later bought or compromised, following the four-step CDN acquisition attack model, it delivers adversarial content and the policy still permits it. The runtime fetches from an allowed address; the policy checks the address, not what comes back. CSP cannot evaluate what the trusted source actually serves.

One narrow exception is worth stating: Content Security Policy Level 3 permits a hash source expression to match an external script where the script element also carries matching integrity metadata. That is genuine content pinning. It reaches script and style elements only, it requires the integrity metadata alongside it, and it is checked at load like everything else.

- **Runtime Anarchy:** once allowed code executes inside the page, CSP does not constrain what that code does within the runtime. Privileged-API access, DOM manipulation, storage access, and runtime resource loading all happen with the host application's full permissions regardless of CSP. Once a script is allowed in, it inherits the page's privileges, and the policy has nothing further to say about how it uses them.
- **Coverage outside the browser:** the policy mechanism is a browser-engine feature. Mobile-application runtimes, hybrid desktop shells, AR/VR scene graphs, smart-TV apps, vehicle infotainment, IoT and OT dashboards, and productivity-tool extension hosts carry no equivalent capability at default. Where browser-engine components are embedded inside non-browser hosts (mobile WebViews, Electron-rendered application surfaces), CSP can apply to the embedded surface; the surrounding runtime carries none of the policy's protection.

CSP is necessary and worth deploying within the browser context. It is not sufficient against ATG, and it does not extend beyond the browser to the rest of the CSR-capable Client Runtime landscape.

14.3: Subresource Integrity: Value and Limits

Subresource Integrity provides hash-based verification that a fetched resource matches an expected hash. For a static script or stylesheet fetched from a known URL with a stable content hash, the mechanism is meaningful protection: if whoever serves the file swaps its contents, or something substitutes it in transit, the bytes change, the check fails at fetch time, and the runtime never runs it. SRI was designed for exactly that case: a fixed file that does not change between fetches. Three architectural limits define where the coverage ends.

- **Load-time scope:** SRI checks the resource as it is fetched. What that resource does once it runs is not covered. The declarative form of the mechanism covers script and link elements. A resource fetched by already-executing code can carry integrity metadata as well, because the Fetch standard gives every request an integrity field and fetch() accepts one, but that verification is opt-in per call, is required by nothing, and is audited by nothing. Supply Chain Injection and the Two-Stage Attack are designed to defeat file-level integrity verification by separating the file that was checked from the

behavior that eventually executes. The static script that passes the hash check fetches data, configuration, or further executable content at runtime, and the malice rides in there. SRI does what it does correctly; the attack model relocates the malice to where SRI does not look.

- **Hash-stability assumption:** SRI requires content addresses to be known and stable in advance. Scripts injected by a tag manager, content generated on the fly, AI inference responses, and CDN content built per request all defeat the assumption. Where the resource a runtime loads is generated, personalized, or otherwise permitted to vary by request, no advance hash exists to verify against. The mechanism cannot apply to content that is not stable in the first place.
- **Coverage in non-browser Client Runtimes:** SRI is a browser-engine feature. Mobile-application runtimes, hybrid desktop shells, AR/VR scene graphs, vehicle infotainment, IoT and OT dashboards, and productivity-tool extension hosts carry no equivalent default capability. Where browser-engine components are embedded inside non-browser hosts, SRI can apply to the embedded surface; outside the embedded surface, the mechanism does not exist.

SRI is correct for what it covers. The coverage is narrower than the ATG attack surface, and what remains outside it is the Detection Blindspot. Static integrity verification of the file as it is fetched is what the Two-Stage Attack defeats. Monitoring how code behaves at the rendering layer while it runs is not.

Subresource Integrity pins file content at fetch, and it needs a hash known in advance, which content generated per request never has. Certificate pinning fixes the identity of an endpoint, so an adversary holding that endpoint satisfies the pin and delivers through it. URL pinning and allowlisting fix which address may be contacted, approving the sender rather than the package. Lockfile pinning fixes what enters the build and says nothing about what running code fetches afterward. Only the first inspects content, and none of the four can hash a response that does not exist until the request is made.

14.4: Zero Trust: Why ATG Operates Beneath It

Zero Trust Architecture is a meaningful security posture against an adversary attempting to gain access to the system. NIST SP 800-207 is guidance rather than a standard, and it sets out seven tenets. Four of them bear on ATG: continuous verification, least privilege at runtime, explicit denial of implicit trust extension, and continuous re-evaluation under changing context. Applied to identity, network, and workload boundaries, the framework raises substantial defensive cost across the access perimeter. ATG operates beneath the Zero Trust boundary because ATG happens after the trust decision has been made.

- **Session-scope persistence:** Zero Trust decisions establish trust at the session boundary. Once a user is authenticated and a session is granted, the session itself carries persistent trust over the duration of its operation. The check happened when the session was issued; after that, whatever the session loads is what runs.
- **Inheritance of trust to fetched content:** ATG attacks the content the trusted session loads at runtime. The session is trusted; the supply-chain content delivered into the session is trusted by inheritance. Zero Trust does not re-check each fetched resource against the decision that granted the session, and its policy machinery has no way to judge whether a fetched file is intact.
- **Runtime principles structurally absent:** continuous verification, least privilege at runtime, explicit denial of implicit trust extension, and continuous re-evaluation under changing context are not part of how client-side rendering works by default. The absence holds across every CSR-capable Client Runtime: browser, mobile, desktop, vehicle, IoT, AR/VR, productivity-tool host.

Zero Trust is the right framework at the boundary. ATG is below the boundary, on every CSR-capable Client Runtime on which the session runs.

14.5: Server-Side Rendering as Partial Mitigation

Server-Side Rendering materially reduces ATG exposure. SSR moves the runtime composition out of the client environment: the application is delivered as HTML, and the client does not assemble it at runtime in the way a CSR application does. Several CSR properties are reduced or absent in pure SSR: the App Blueprint assembly becomes a server-side problem, the Manifest Illusion is largely dispelled, and Code That Writes Code at the client is largely absent. Against ATG, SSR is a partial mitigation, and two limits explain why it is only partial.

- **Hydration brings exposure back:** modern "SSR" applications often hydrate on the client (the client attaches JavaScript behavior to the server-rendered HTML and takes over from there), load additional scripts at runtime, and integrate third-party content. The hydration model brings ATG exposure back to the extent the client takes over composition after the initial render. Pure SSR with no client-side scripting is the strongest mitigation; few real applications meet that bar.
- **Browser-context only:** SSR is a browser-context mitigation. Most non-browser CSR-capable Client Runtimes (mobile applications, desktop shells, vehicle infotainment, IoT dashboards, productivity-tool extension hosts) have no SSR analog. The runtime composition in these hosts is structurally on the client because the host is the client.

SSR does not extend as a mitigation across the rest of the Client Runtime host landscape.

SSR materially reduces ATG-class exposure for browser-hosted applications that can bear the user-experience trade-off. For high-assurance browser contexts where modern UX is not the primary requirement, SSR is the strongest defense available today. SSR does not address the non-browser Client Runtime host platforms.

14.6: Defenses Outside the Browser, and the Endemic Flaw

Outside the browser, far less of this defensive machinery exists. CSP, SRI, the Permissions API, Trusted Types, and the surrounding browser-engine primitives are capabilities every modern browser provides, and each has to be switched on by the application that wants it. The other nine CSR-capable Client Runtime host categories carry analogs that range from partial to absent.

That asymmetry points to something deeper: ATG is endemic to client-side rendering. Because the vulnerability arises from the architecture itself, composition at render time and execution on the Glass, it is present, independently, in every runtime that uses the pattern: browser, mobile webview, desktop app, smart TV, kiosk, embedded panel. It is not transmitted between runtimes; each inherits it from the shared design. The Endemic Flaw is that inherited condition: like any endemic condition it cannot be eradicated by treating individual cases, and a per-runtime patch does not reach the pattern that produces it. It is not an infection that spreads, but a pre-existing condition every Client Runtime is born with.

CSR runs in mobile applications, desktop shells, vehicle infotainment, IoT and OT dashboards, AR/VR scene graphs, smart-TV applications, productivity-tool extension hosts, and UGC interactive runtimes. Each is a non-browser CSR-capable Client Runtime. The browser-class defenses do not generally extend to these hosts, even where browser-engine components are embedded inside them. Those hosts are unprotected in proportion to how little of that machinery reaches them, and the amount varies by host rather than being uniformly absent. Where a host embeds a browser engine, that engine's primitives are available to the embedded surface whether or not the host configures them: Electron's own security documentation recommends a Content Security Policy and documents two ways to deliver one. What no host outside the browser provides is a default, and availability is not deployment.

The gap takes a different shape in each platform category.

- **Mobile native and hybrid runtimes:** over-the-air bundle delivery, where code updates are pulled straight from a server after install without going back through the app store,

has no equivalent of a file-integrity check at runtime. Hybrid and cross-platform runtimes (React Native, Capacitor) have no equivalent of CSP for the JavaScript bridge that lets web code call native device functions. Where the host embeds a WebView, browser-engine defenses can apply inside the WebView surface; they do not extend to the surrounding native host.

- **Desktop app shells and plugin hosts:** Web-technology desktop shells (Electron, Tauri) and the plugin marketplaces of development environments (IntelliJ Platform, VS Code VSIX extensions) carry no consistent analog of CSP or Trusted Types. A loaded plugin runs with the host application's permissions.
- **Appliance, IoT, vehicle, and control-system runtimes:** smart-TV client-side rendering, IoT dashboards, vehicle infotainment, and the operator screens of industrial control and SCADA systems carry minimal to no equivalent defensive ecosystem at the rendering layer. Well-run industrial control environments compensate with perimeter measures (URL pinning, mutual TLS, network segmentation, hardened runtime engines) that materially narrow the script environment relative to commercial deployments. Those perimeter measures do not reach the supply-chain source of the rendering-layer components, which is where ATG sits. Where these runtimes embed browser engines, the embedded surface can carry browser-class defenses; the surrounding runtime cannot.
- **AR/VR, game and UGC, productivity and data-tool runtimes:** the AR viewers that open a 3D file straight from a web page (USDZ AR Quick Look), the Roblox and Unity AssetBundle content paths, Jupyter notebook trust models, and the add-ins that run inside Office documents carry no equivalent of browser-class defenses.
- **Inference-API consumption:** no standard exists for checking an AI inference response against what the application asked it to produce. The data-endpoint trust gap is unaddressed across every CSR-capable Client Runtime that consumes inference output.

The Endemic Flaw is the structural convergence point across the language, platform, provider, open-source, and AI layers: the same architectural exposure appears in each, and per-runtime measures do not reach the pattern that produces it. Each layer carries the same architectural requirement: a single source of truth for the resources the runtime actually loads, detection of content swapped at runtime, and protection against tampering with the application's state in memory while it runs.

The architectural patterns translate across runtimes. The implementing code does not.

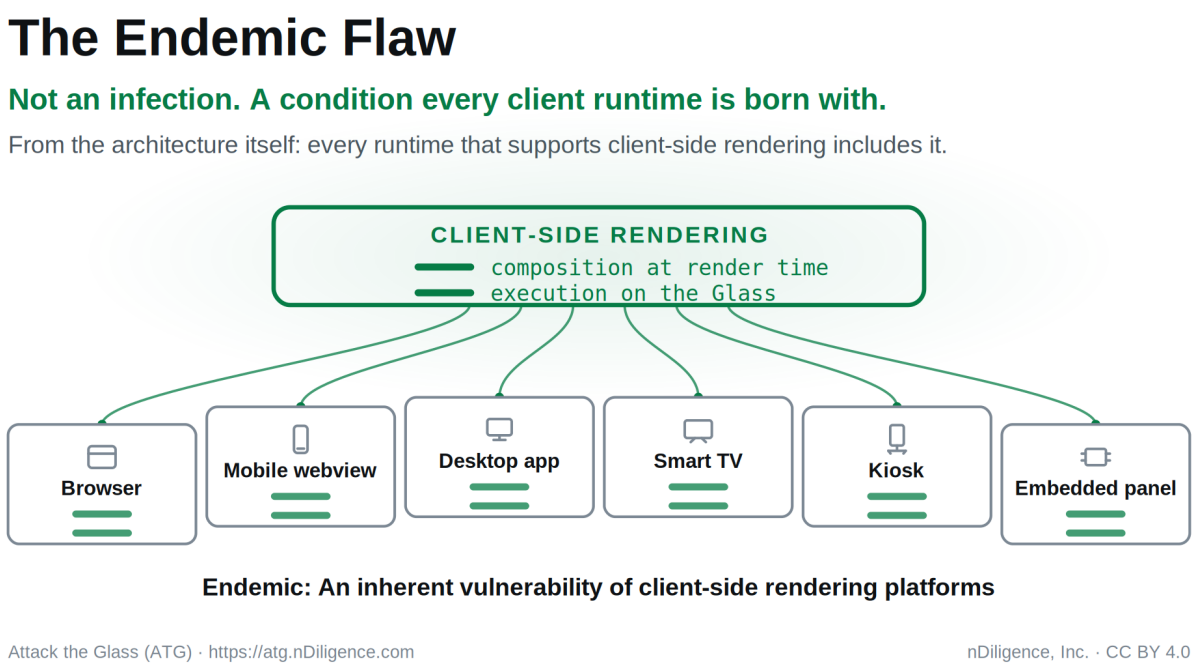

***Figure 30**: Not an infection that spreads. A condition every client runtime is born with.*

Inside the browser, the defenses do real work. Outside the browser, the equivalent maturity does not exist as a default capability. The complement that would close the gap, runtime behavioral monitoring at the rendering layer, is not yet a default capability of any Client Runtime in the deployed landscape. Operational maturity reduces how many positions an adversary can occupy and shortens the time between activation and discovery. What it cannot do is establish, at the moment of render, that a permitted source returned what the application expected, because no Client Runtime in the deployed landscape offers an operator that check. The limit sits in what a runtime provides, not in how well an operator runs it.

Chapter 15: Relationship to Existing Frameworks

Cybersecurity work is organized through frameworks. CVE/NVD catalogs vulnerabilities. MITRE ATT&CK catalogs adversary techniques. NIST SP 800-161 governs supply chain risk management. NERC CIP, TSA directives, DORA, and the SBOM regime under EO 14028 govern compliance for the regulated sectors. Practitioners work inside those frameworks. If ATG cannot be located inside them, it sits to one side of the work practitioners actually do.

CVE/NVD does not register ATG. MITRE ATT&CK registers it partially. NIST SP 800-161 covers supplier relationships but not the runtime trust boundary. NERC CIP, TSA directives, DORA, and EO 14028 each fall short at the runtime layer.

15.1: Why ATG Does Not Appear in CVE/NVD

CVE/NVD catalogs vulnerabilities in identifiable software products. Each entry has a vulnerable product, a vulnerable version range, and a path to a patch. ATG presents none of those.

- **No identifiable vulnerable product:** ATG is a type of supply chain injection vulnerability that targets client-side rendering, and it lives in the architectural pattern where a CSR-capable Client Runtime meets a digital supply chain, not in any single product. The product the vendor shipped and the application the user is running can be technically identical and operationally different. The CVE system has no slot for that distinction.
- **No clear vulnerable version range:** the vulnerability lives in the runtime composition, not the codebase. Two users running the same version of the same application can have different runtime exposure based on what their host runtime actually loaded. A version number says nothing about what the runtime assembled.
- **No clear patch:** the patch model assumes a vendor releases an update that closes the vulnerability in subsequent versions. Closing ATG requires architectural change in how runtimes verify and monitor what they execute, which is the point Malice Without Malware makes. Per-product patches do not reach it.

Organizations relying on CVE/NVD as the vulnerability stream will not see ATG-class threats reported there. The frameworks that catalog vulnerabilities need to be extended to cover it.

15.2: ATG Mapped to MITRE ATT&CK

ATT&CK is the working vocabulary of detection and response. The mapping for ATG is partial: some operations register in ATT&CK terms, and some structural patterns do not.

T1195 (Supply Chain Compromise) is the closest technique. Its three sub-techniques cover ATG-relevant operations: T1195.001 (Compromise Software Dependencies and Development Tools), T1195.002 (Compromise Software Supply Chain), and T1195.003 (Compromise Hardware Supply Chain). ATG operations that compromise upstream code, build tooling, or distribution register as supply chain compromise in ATT&CK terms.

Client-side execution techniques in ATT&CK are anchored to the browser. T1185 (Browser Session Hijacking), T1176.001 (Browser Extensions), and T1539 (Steal Web Session Cookie) all assume the browser is where execution happens. ATT&CK has since widened the parent technique to T1176 (Software Extensions) and added an IDE-extension sub-technique, which reaches one non-browser host rather than the landscape. ATG runs in the broader Client Runtime landscape: mobile, desktop shells, vehicle infotainment, IoT and operational-technology dashboards, AR/VR, productivity-tool hosts, and user-generated content interactive runtimes. ATT&CK has no equivalent client-side techniques for the non-browser hosts.

Three ATG-specific patterns lack first-class ATT&CK representation.

- **The dormancy and activation pattern:** ATT&CK's tactic structure has no place for the pattern ATG operations follow: sit quiet for a long time, then activate. The dormancy phase is the absence of malicious behavior; ATT&CK is built on the presence of it.
- **Runtime Anarchy:** vendor and supply-chain code executes with the host application's privileges, on equal footing with the application's own code. ATT&CK's privilege models do not register this case: the vendor's code is not malware in the ATT&CK sense, and the privileges it inherits are not escalation in the ATT&CK sense.
- **The Endemic Flaw:** ATT&CK's enterprise matrix is broad, but its non-browser client-side coverage is sparse, and nothing in the matrix represents a flaw that is architectural and reaches across runtimes.

ATG behaviors fall partly inside ATT&CK and partly outside. The mapping works for what falls inside; the framework needs extension for what does not.

15.3: ATG and NIST SP 800-161

NIST SP 800-161 is the federal government's guidance on cybersecurity supply chain risk management. It addresses supplier identification, supplier risk assessment, supplier monitoring, and incident response involving suppliers. These are the upstream controls that operate before runtime delivery.

Coverage starts at the supplier-relationship layer. Where an ATG operation involves compromise of an upstream supplier (a CDN operator's account, a package maintainer's credentials, an analytics vendor's build pipeline), SP 800-161-compliant supplier risk management reduces the surface.

The framework does not extend to the runtime trust boundary. Verification ends before execution: once the supplier-relationship controls have done their work, a fetched file crosses into execution inside the host runtime. Suppliers may be vetted, contracted, and monitored, and their products may still deliver adversarial content into the runtime through Supply Chain Injection. The runtime-trust-boundary gap appears in every Client Runtime: browser, mobile, desktop, vehicle, IoT, AR/VR, productivity-tool host.

Knowing your suppliers' suppliers is a separate problem. SP 800-161 expects organizations to maintain that visibility, and on paper the expectation is fully worked out. In practice, even SP 800-161-compliant organizations have not generally achieved transitive inventory at the depth ATG requires; Log4j is the case where the gap manifested broadly.

SP 800-161 is the right framework for the supplier-relationship layer. The runtime trust boundary needs its own.

15.4: ATG and the Sectoral Regulatory Frameworks

Regulated organizations operate under sectoral compliance frameworks that pre-date ATG.

- **NERC CIP (energy):** the Critical Infrastructure Protection standards address asset identification, vulnerability assessment, and incident response across the bulk electric system, and CIP-013 addresses supply-chain risk management for those systems. ATG-class threats in operational technology dashboards and supervisory control interfaces sit largely outside the framework's current scope.
- **TSA Directives (transportation):** the TSA's pipeline and rail cybersecurity directives are built around network segmentation, access control, continuous monitoring, and patching. Supply-chain integrity is largely absent from them, and the rendering layer is not addressed at all.
- **DORA (EU financial services):** the Digital Operational Resilience Act addresses ICT third-party risk management and operational resilience for financial entities. Its third-

party-risk provisions are the closest thing in current regulation to requiring that an organization know its providers' providers.

- **PCI DSS 4.0 (payment card):** the standard governs systems that store, process, or transmit cardholder data, and its script-integrity provisions require a payment page to inventory the scripts it loads and to detect unauthorized change to them. Coverage is scoped to the cardholder-data environment, and the same scripts on pages outside that scope carry no requirement.
- **EO 14028 and SBOM mandates:** Software Bill of Materials requirements make the contents of a build output visible. What the build output contains is not what ends up executing inside the runtime (Origin Laundering blends and obscures provenance once parts are composed), so an SBOM is necessary but not sufficient for ATG defense.

The gap is consistent across sectors. Each framework reduces some part of the supply chain surface; none reaches the runtime trust boundary.

15.5: Standards Gaps ATG Exposes

Five standards gaps result. The frameworks were built before ATG; each falls short in a distinct way.

- **Vulnerability cataloging gap:** CVE/NVD cannot capture ATG-class threats because its fields do not fit them.
- **Adversary technique cataloging gap:** MITRE ATT&CK partially captures ATG; ATG-specific patterns need first-class representation.
- **Supply chain risk management gap:** NIST SP 800-161 covers supplier relationships; the runtime trust boundary has no equivalent framework.
- **Sectoral regulatory gap:** critical-infrastructure and financial-services frameworks have not yet been updated for ATG-class exposure.
- **Cross-runtime standards gap:** browser-class standards (CSP, SRI, the Permissions API, Trusted Types) have no equivalents across the other nine platform categories.

Each gap sits in a different framework, and none of those frameworks currently treats the runtime boundary as in scope.

Part V - Reference Methodology

The Reference Matrix workbook is a structured data set that scores the language, platform, and provider landscape against the severity ramp, and it carries equal weight with the chapters.

The methodology covers the severity ramp, the conservative inclusion criterion, the four scoring dimensions, user-experience prevalence on the Languages sheet, device scale on the Platforms sheet, D5 effects as cross-sheet score, the critical-count summaries, the methodology caveats, and the per-sheet structure of the workbook.

Chapter 16: Severity Methodology and Reference Matrix Guide

ATG has multiple dimensions: some deeply technical, down at the runtime-language level; others are not technical at all and come down to how an organization runs its controls. The methodology measures ATG across those dimensions, and the Reference Matrix records the score it produces for each row.

The Reference Matrix carries equal weight with the chapters rather than the standing of an appendix. It is meant to be read alongside the framework chapters and to stay understandable on its own.

16.1: The Severity Ramp and Color Encoding

The severity ramp runs from 1 Critical at the most severe end to 5 Low at the least severe end, with 2 High, 3 Moderate, and 4 Limited filling the intermediate steps. A 0 / N/A category sits outside the ordinal scale and applies to cells where the dimension being scored is inapplicable to the row category.

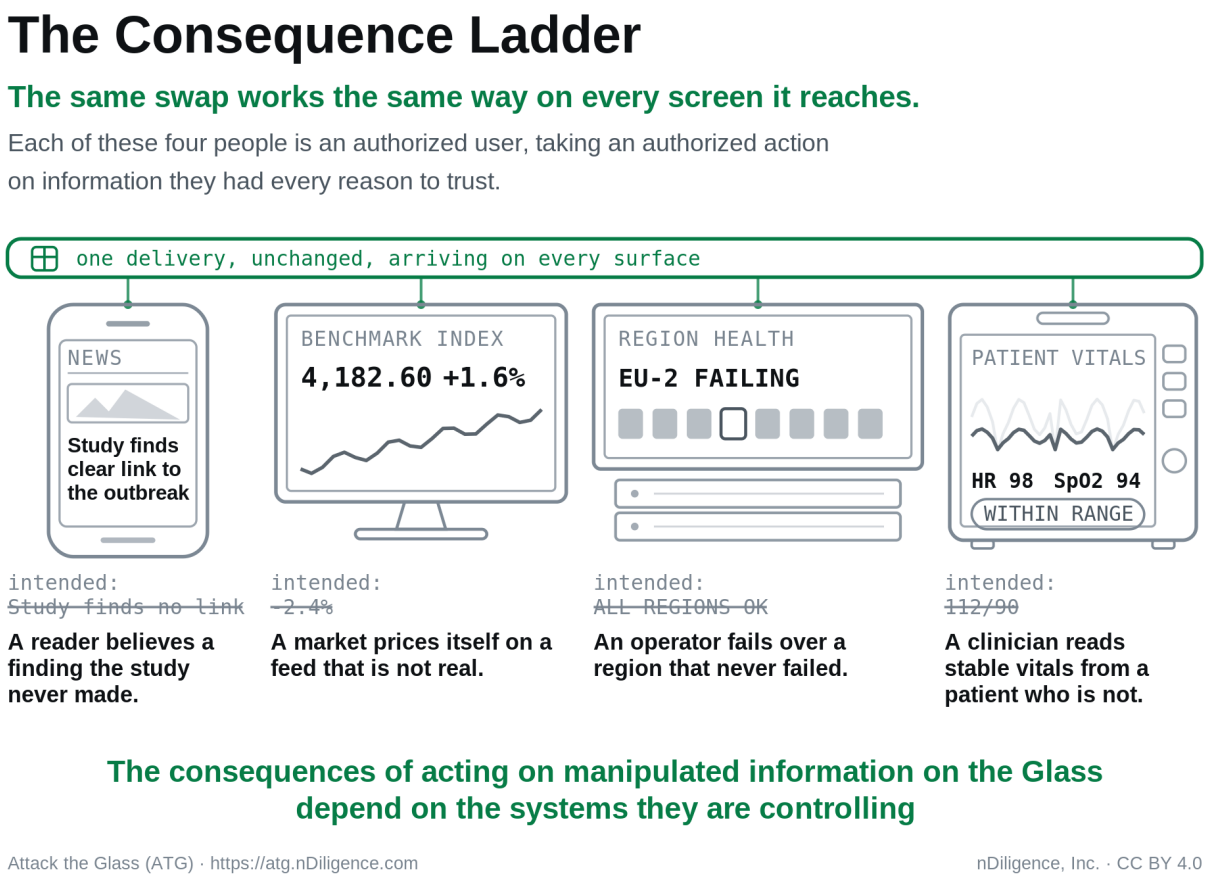

***Figure 31**: Consequence rises with what the affected decision controls, not with the sophistication of the method.*

Inverted ordering is deliberate. Readers trained on CVE and CVSS expect "critical" at the top of the page, so placing 1 at Critical keeps the highest-severity rows where the eye looks first. Color encoding follows standard heat-map convention: deep red at 1 Critical through deep green at 5 Low, with the intermediate steps held consistent across all three sheets, so a reader can compare sheets by eye without relearning the colors.

Inapplicability is not the lowest score. A 0 / N/A cell records that the dimension being scored is inapplicable to the row category. A 5 Low cell records that the dimension applies, but the realized severity is the lowest the ramp carries. The distinction matters because the two scores say different things about the row category. CSS scored against eval is 0 / N/A: CSS has no mechanism that evaluates a string as code, so the dimension does not apply to it. HTML scored against runtime code execution is 5 Low: the capability is present in the rendering pipeline but structurally constrained.

The same distinction applies when a reader scans a whole column. A column with many 0 / N/A cells indicates that the dimension simply does not apply to many of the row categories scored on the sheet. A column with many 5 Low cells indicates that the dimension applies to those row categories and is well defended in practice. The critical-count summary at the foot of each column treats the two cases differently.

16.2: Inclusion Criterion and Scope Boundary

The inclusion criterion draws a precise boundary, and it is deliberately conservative. A vulnerability is in scope for ATG, and therefore eligible for inclusion in the matrix, if and only if it manifests through code or content execution inside a CSR-capable Client Runtime at the rendering surface. Admitting a runtime only when it passes that test keeps the matrix from thinning out into a general supply-chain inventory, and the caution makes the scoring more defensible rather than less. The criterion has two structural parts; each is necessary; neither is sufficient on its own.

CSR-capable Client Runtime. The runtime environment must be one that loads, interprets, or generates content at runtime to influence what is rendered or what executes. Precompiled-only Client Runtimes (runtimes that load and execute only precompiled code, with no runtime modification of behavior) are out of scope by structural definition. ATG cannot occur in a runtime that never modifies its own behavior while running, and the matrix draws its boundary at the same place.

Rendering-surface manifestation. The vulnerability must produce its effect at the rendering surface where the user experiences the system. Vulnerabilities that exist in the supply chain but never reach a CSR-capable Client Runtime, build-time-only compromises that affect server-side build output alone, and configuration-only compromises that do not affect rendered code or content are out of scope even when they share supply-chain

mechanics with in-scope vulnerabilities. Supply-chain mechanics run through all of these cases, but the boundary stays at the rendering surface.

A CDN-delivered third-party script compromised at the source and executing inside the CSR-capable Client Runtime of a mobile WebView is in scope: code execution inside a CSR-capable Client Runtime at the rendering surface, both structural parts satisfied. A build-time CI/CD compromise that injects code into server-side build output that never reaches any CSR-capable Client Runtime is out of scope: the rendering-surface manifestation part fails. The mobile-WebView example is deliberate: the test is where the rendering happens, not whether a browser is involved, and client-side rendering runs on far more hosts than browsers.

The criterion is conservative in two senses. First, it excludes server-side runtime scripting and server-side template injection (SSTI) outright. Second, it requires manifestation in the CSR-capable Client Runtime at the rendering surface rather than treating any supply-chain reach as sufficient. Reading either condition loosely would turn the matrix into a general supply-chain inventory and give up what the rendering-surface focus buys. The criterion sits at the structurally meaningful boundary, and the conservatism is deliberate.

16.3: The Exclusion of Server-Side Runtime Scripting and SSTI from ATG

The exclusion is one of scope, not of relevance. Server-side runtime scripting (server-side template injection, server-side request forgery used as a code-execution vector, RCE in server-rendered languages, server-side eval-class vulnerabilities) is out of scope for ATG and therefore out of scope for the matrix. The exclusion is not a claim that server-side runtime scripting does not matter; it is a claim that the matrix covers the layer where the structural defensive gap sits.

D5 effects can be produced on any tier of a system architecture. Malice Without Malware describes a defensive blind spot that is not unique to client devices; the pattern appears wherever a system trusts fetched code or content at runtime without checking how it behaves. Supply Chain Injection is the general form and applies to any supply chain that delivers code or content into an executing environment. The concepts reach server-side environments. The matrix does not.

Server-side runtime scripting sits inside the traditional defensive boundary of modern cybersecurity frameworks. The boundary is not perfect, but it carries identified controls: input validation at trust boundaries, runtime application self-protection where deployed, network segmentation, perimeter monitoring, server-side WAFs, server-process EDR. Client-side rendering layers carry none of those controls as a matter of architectural default. The matrix records the layer where the defensive boundary is missing; including

server-side runtime scripting would mix two defensive situations that are structurally different and weaken the per-axis scoring.

Server-side environments are addressed by NIST SP 800-161 and adjacent supplier-relationship frameworks; the matrix does not duplicate that work or claim it is missing. ATG-class scope and server-side scope are complementary, not competing. The matrix names what the rest of the framework landscape does not: the unprotected rendering layer where CSR meets the digital supply chain.

16.4: The Four Scoring Dimensions

Four dimensions inform every cell of the matrix. Each dimension is read against the row-column intersection; together they produce the cell's severity score.

- **Default availability:** whether the capability is available by default in the row category, or requires non-default configuration or installation. Higher default availability raises severity. When a capability is there without anyone switching it on, the matrix scores it as part of the architectural default.
- **Prevailing mitigations:** defenses in widespread practical use against the row-column combination, and their realistic coverage. Stronger prevailing mitigations lower severity; weaker mitigations raise it. The dimension is "prevailing," not "available." A defense that exists but is rarely deployed does not meaningfully reduce the damage an exploited capability actually does. The matrix scores what is operating in practice, not what is theoretically reachable.
- **Blast radius:** the scope of impact if the capability is exploited. The dimension is informed by the user-experience-presence analysis on the Languages sheet, the device-scale analysis on the Platforms sheet, and the reach analysis on the Providers sheet. Larger blast radius raises severity.
- **Exploit precedent:** confirmed historical evidence of the capability being exploited in production systems. Confirmed precedent raises severity: the capability is operationally weaponized, not theoretical. Absence of precedent is treated cautiously. Detection is not deployment, and the dimension is informational rather than determinative.

The dimensions combine through analyst judgment, not a formula, and that is a deliberate choice. CVSS-style algorithmic scoring optimizes for cross-analyst reproducibility at the cost of analytical nuance; the matrix optimizes for analytical nuance at the cost of reproducibility. The rendering surface has too many dimensions for a formula to compress without losing what matters.

Algorithmic scoring is the right choice when reproducibility is the governing concern and the dimensional structure is well-bounded. Neither condition holds here. The capability set, the surface landscape, and the operational profile of the rendering surface together produce a scoring problem where an analyst's judgment does better than a formula. The cost is that two analysts may differ on a cell; the trade-off is named here and handled through the calibration notes in the workbook and the published review schedule.

16.5: User-Experience Prevalence (Languages Sheet)

The Languages sheet scores each language by how much of the deployed CSR landscape its user-experience attack surface appears on, running from a narrow slice up to nearly every surface. The ramp anchors to the User Experience Attack Surface by Language dimension. A language present on a majority of rendering surfaces carries a different total-impact profile than a language present on a narrow slice of them, even if the two languages share a capability set.

Capability and prevalence are independent. The Languages sheet scores capabilities (the seventeen language-level capabilities, eight code-introduction and nine runtime-manipulation) along their own columns; user-experience prevalence sits as a separate axis. A language can carry a rich capability profile and a narrow UX prevalence: languages used heavily in industrial niches with deep capability sets but limited deployed-surface presence. A language can carry a constrained capability profile and a broad UX prevalence: HTML and CSS appear on essentially every web rendering surface but constrain the realized capability set. Both directions exist in the deployed landscape, and the matrix records each separately.

Scoring runs from 1 Critical UX Prevalence at the global-surface band to 5 Low UX Prevalence at the narrow-slice band, on the same scale as the severity ramp so cross-axis reading is consistent. JavaScript, HTML, and CSS occupy the 1 Critical band because the modern rendering pipeline is built on them. Languages present on a narrow slice of the deployed landscape sit at 5 Low. The 0 / N/A category applies where UX prevalence is structurally inapplicable: a language used only in build-time tooling that never reaches a rendering surface scores 0 / N/A on the ramp.

Total impact is a function of capability and prevalence together, not either one alone. A language with severe capabilities and narrow prevalence carries a different operational risk picture than a language with limited capabilities and global prevalence; the matrix preserves both axes precisely so each cell can be read against the language's full operational footprint, not against capability or prevalence in isolation.

16.6: Device Scale (Platforms Sheet)

The Platforms sheet scores each platform category by deployed device scale, the blast-radius argument that runs from a single device to billions. The ramp anchors to deployed device scale, and to the Devices (Global) column that carries the per-category device-count estimates. The estimates are best-estimate at release. Data-sourcing methodology is documented in the workbook, and the caveats section records the per-version review cadence.

Device scale is a blast-radius indicator, not a count of unique devices. Compromise at the platform level reaches every device running that platform by default; the count says how far a compromise travels, not how many distinct devices were tallied. A capability scored 1 Critical against a platform category at 1 Critical Device Scale carries a structurally larger blast radius than the same capability against a category at 5 Low Device Scale. A mobile WebView capability that affects every WebView instance globally and a niche industrial-display capability that affects a few thousand units may share a 1 Critical capability score; the platform-side total impact differs because the deployed footprint differs.

Scoring runs from 1 Critical Device Scale at the global-deployment band to 5 Low Device Scale at the narrow-deployment band, on the same scale as the severity ramp so cross-axis reading is consistent. The 0 / N/A category applies where deployed device scale is structurally inapplicable to the platform category being scored.

Total platform-side impact is a function of platform capabilities and device scale together. This parallels the Languages sheet's capability-and-prevalence pairing: capability without scale and scale without capability each tell only part of the platform-category risk picture, and the matrix preserves both axes so each cell can be read against the platform's full operational footprint.

16.7: D5 Effects as a Cross-Sheet Score

D5 effects are scored as a cross-sheet axis on every sheet of the matrix. Every row-by-column intersection on the Languages sheet, the Platforms sheet, and the Providers sheet carries a D5 score that records which of the five effects (Deceive, Disrupt, Degrade, Deny, Destroy) the capability or compromise mode is structurally suited to produce. D5 is the operational expression of the Malice Without Malware argument and applies wherever the matrix scores a capability or compromise mode, not only in places where a specific subset of effects predominates.

The cross-sheet treatment is deliberate because operational effects are properties of the operation, not properties of the row or column dimension scored on any single sheet. A capability on the Languages sheet, runtime code execution in JavaScript, can produce

different D5 effects depending on which platform hosts it (the Languages-by-Platforms intersection) and which provider supplies it (the Languages-by-Providers intersection). Treating D5 as a per-sheet axis would split the operational effect across three sheets and leave the reader to combine them in their head; treating D5 as a cross-sheet axis puts the combined reading in each cell.

Scoring is multidimensional, not ordinal. Each of the five effects is scored independently: a cell can be a structural fit for Deceive and Disrupt and not for Destroy, and the matrix records that profile rather than collapsing it into a single number. A short code in each cell carries the five-part reading in the room a spreadsheet cell allows; the workbook walk-through documents the code and how to expand it into the full five-axis interpretation. The short form is a presentation choice rather than a simplification. Each cell still carries all five readings.

D5 carries the information-manipulation hierarchy into the matrix. A cell that fits Deceive sits where the framework places the center of gravity for the ATG threat, because information manipulation is the easiest to execute, the most rewarding to run, and the most dangerous category of ATG attack. That is a claim about prevalence and damage, not an ordering of the five effects against one another, and the matrix must not be read as placing an operation somewhere on a line running from Deceive to Destroy. Scoring D5 in every cell records which effects a capability or compromise mode is structurally suited to produce.

16.8: Critical-Count Summaries

Each column of each sheet carries a critical-count summary at the foot. The summary is the count of cells in the column scored 1 Critical, optionally paired with a secondary count at 2 High. A reader scanning a sheet's columns can identify the dimensions most heavily concentrated at the top of the severity ramp without reading every cell; the summary is the column-glance reading.

The summary is informational, not determinative. A column with a high critical count is a column that warrants attention; it is not a column that supersedes per-row analysis. Two columns with the same critical count can differ structurally if their distributions across row categories differ, and the matrix's per-row analytical work is what surfaces those differences. The summary is a navigational aid; the per-cell analysis is the substance.

0 / N/A and 5 Low counts tell different stories. A column with a high critical count and a high 0 / N/A count is a column where the dimension is critical when applicable but structurally inapplicable to many row categories. A column with a high critical count and a low 0 / N/A count is a column where the dimension applies broadly and concentrates at the

severe end. The two pictures are different operational realities, and the summary surfaces the count without resolving them.

A reader checking the runtime-code-execution column on the Languages sheet finds a high critical count. The capability is structurally critical wherever the language hosts it, and the column directs per-row reading to the languages where the capability is available.

16.9: Caveats and Review Cadence

The methodology's caveats are named plainly.

- **Analyst-judgment-driven: the scoring dimensions combine through analyst judgment, not a formula.** Cross-analyst reproducibility is not the goal and is not claimed.
- **Best-estimate inputs:** the Devices (Global) figures and the exploit-precedent inputs are best-estimate at release. Data-sourcing methodology is documented in the workbook.
- **Conservative inclusion:** the matrix excludes vulnerabilities that fall outside the conservative inclusion criterion. A broader vulnerability inventory is available in the established frameworks: CVE/NVD, MITRE ATT&CK, and the sectoral regulatory frameworks.
- **Living workbook: the matrix is intended to change as the rendering-surface attack landscape changes.** Per-cell scores will change over time as new evidence emerges.
- **D5 multidimensionality:** the cross-sheet D5 score is multidimensional. It is not collapsible into a single ordinal value, and the matrix preserves the per-cell five-axis reading as a methodological commitment.

Review cadence is published. The matrix is reviewed and revised on a documented schedule, with per-version diffs maintained to show what changed between releases. The document version and the workbook version are aligned; each document revision carries a corresponding workbook revision.

The rendering-surface attack landscape keeps moving, and the matrix is revised to keep pace. Per-version diffs are how the matrix records that motion, and the published cadence is how readers know when to check.

16.10: The Workbook, Sheet by Sheet

The Reference Matrix workbook is organized as three primary sheets and a small set of supporting sheets. The primary sheets carry the per-row severity scoring; the supporting sheets carry the legend, the change log, the version metadata, and the per-sheet

methodology summaries that let the workbook stand on its own. Critical-count summaries sit at the foot of each column on all three primary sheets.

- **Languages sheet:** rows are the languages used in the deployed CSR-capable Client Runtime landscape. Columns are the seventeen language-level capabilities (eight code-introduction; nine runtime-manipulation), plus user-experience prevalence as a separate axis and a D5 cross-sheet score per cell.
- **Platforms sheet:** rows are the frameworks, runtimes, and libraries grouped under the ten platform categories. Columns are the eight platform-level structural dimensions, plus the Devices (Global) column that anchors device scale and a D5 cross-sheet score per cell.
- **Providers sheet:** rows are the providers within the twenty provider categories from the provider landscape. Columns include the six provider dimensions (injection vectors, defendability, trust model, update authority, aggregation depth, dependency chain depth), the five ways an adversary gets in (account takeover, network-path hijack, ownership change, insider compromise, cyber exploitation), the reach indicator, and a D5 cross-sheet score per cell.

The supporting sheets are what let a reader interpret the workbook without outside help. The legend expands the severity ramp and the D5 notation. The change log records per-version diffs on the published review cadence. The version metadata names the workbook version, the release date, and the document version alignment. The per-sheet methodology summary restates the relevant methodology on a single page, sheet by sheet, so the workbook is interpretable on its own.

Per-cell D5 notation is a compact five-character code, one position per effect (Deceive, Disrupt, Degrade, Deny, Destroy), with documented values per position. The code is short because a spreadsheet cell has limited room; the legend sheet documents the values and how to expand them. A reader can expand any per-cell D5 code into the full five-axis reading by consulting the legend.

The workbook is interpretable on its own. Held without the framework chapters, the matrix is read through the per-sheet methodology summaries. Held without the workbook, the framework chapters give the analysis without the per-cell scoring. Either is usable on its own. Together they give the full picture.

Closing

ATG is not a flaw in a single product. It is a property of how modern interfaces are assembled: built at runtime, on the user's own device, from code and data pulled across a supply chain, trusted by address rather than by content, and executed after every upstream check has already passed. Because that is how essentially every browser, application, and connected screen now works, the exposure is not confined to one platform or one vendor. It reaches the phone, the control-room display, the vehicle dashboard, and the trading terminal alike.

The public record already shows this class of attack in operation, through CDN acquisition, third-party script injection, open-source maintainer takeover, and AI-generated deception, and the confirmed cases are growing in frequency, reach, and sophistication. The defenses the field relies on are not failing at what they were built to do. They stop at the runtime boundary, before the point where the content a person actually sees is assembled. Static verification checks the delivered file, not how it behaves; source authentication proves where a response came from, not what it does; and outside the browser even those partial protections mostly do not exist.

Closing the gap means watching how content behaves as the screen is drawn, inside the runtime, on every platform that builds its interface at runtime, and treating good code from trusted sources as something still to be checked at the moment it executes. That capability does not yet exist as a default anywhere in the deployed landscape. Building it is an engineering problem, not a mystery, and it is the work that decides whether large-scale, tailored deception stays a described risk or becomes a routine one.

The Reference Matrix is a starting point for that work: a way to see the language, platform, and provider surface, to score where the exposure concentrates, and to make the shape of the problem clear enough to act on. It is meant to be argued with and revised, not taken on faith. The measurements will change as the landscape changes; the structure is what carries forward.

Setting all of this down in one place is meant to make the threat understandable enough to answer. The right response to a description like this is not to accept it but to test it: to check the model against what you know, to find where it overreaches or falls short, and to help build the defense the runtime still lacks. The threat is described here as accurately as current knowledge allows. What happens next depends on the people who read it.

END OF DOCUMENT