Attack the Glass (ATG): Technical Primer

Technical Primer 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

The Vulnerability

A journalist checking a source document, a grid operator reading a control display, and a finance officer approving a transfer all make decisions based on what a screen shows them. None of them can check whether the screen is showing what the system actually sent. Modern applications are not shipped finished. They are put together at the moment they are drawn, on the user's own device, out of dozens to hundreds of pieces pulled in live from CDNs, analytics and advertising scripts, payment processors, and identity providers. Whatever those sources send back, the application runs, with every permission the application itself holds. *Attack the Glass* (ATG) is a type of supply chain injection vulnerability that targets that step, client-side rendering, which is very poorly protected.

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. The attacker picks who gets the change: everyone, a country or language group, a segment defined by behavior or demographics, or one named person on one device. It reaches the web browser and the systems beyond it, up to and including critical infrastructure. Attacks of this kind can be configured to leave no forensic evidence on the device and to set off no intrusion detection.

A device trusts a source because of the address it fetched from, not because of what came back. Configure it to trust a domain and it will run whatever that domain returns, without checking that today's response matches yesterday's, or that it matches what anyone else gets from the same request. The server decides what to send. It can send different things to different requesters based on IP address, browser, referring page, or location. That is how personalization, A/B testing, and regional content work, and it is also how an ATG attack gets delivered. Anyone who controls a piece of the supply chain feeding an application can send altered content to the people they choose and clean content to everyone else.

As with other supply chain vulnerabilities, an ATG campaign requires the attacker to influence or control dependencies within the digital supply chain used by a software system, website, application, service, or other technology platform. What is different about ATG is where the attack lands. Content, code, or configuration gets changed or replaced on its way to the screen. The replacement runs inside the application with the same permissions as the real code, and the application never checks it again after loading it. It arrives over a trusted connection. It runs without being verified. It looks exactly like what everyone else is seeing. And it leaves nothing behind in the backend systems, network logs, or audit trails that security teams actually watch.

ATG is not an unprecedented exploit, and its parts are individually familiar. Client-side supply-chain security already exists as a commercial category. Magecart has skimmed payment data from the rendering layer for a decade. Cloaking, serving one response to a scanner and another to a victim, is long established in malvertising. PCI DSS 4.0 reaches part of the surface, requiring payment pages to inventory the scripts they load and to detect unauthorized change to them. What ATG adds is one account that covers all of it, carried past the browser into mobile runtimes, desktop shells, appliances, vehicle infotainment, industrial control interfaces and productivity hosts, where none of those partial measures reaches by default.

Browsers are the biggest case, not the only one. Mobile apps that pull code updates straight from a server after install, without going back through the app store, desktop apps built on web technology, vehicle infotainment systems, and industrial operator displays all build their screens the same way, and they pull from the same CDNs, third-party scripts, and open-source packages. Isolating the network helps less than it looks. Classified networks, military intranets, and air-gapped industrial control environments are built from components that came through the same supply chain, and if a component was compromised before it went behind the wire, the wire does not matter. The supply chain inside a protected network is smaller and better controlled than the open web, so those environments are exposed differently rather than exempt.

None of this means existing defenses are failing at their jobs. Static integrity checks, network controls, Zero Trust, and supply-chain attestation, the signed record of how a component was built and by whom, all deal with malicious code arriving from a malicious source, and they deal with it well. ATG uses legitimate code from trusted sources, and it works in the gap between the last thing those defenses watch and the first thing that matters to the person at the screen. Polyfill.io and Magecart show the mechanism working in production. Nothing in the instrumentation we have shows how often it is used.

ATG is a serious application-security problem. It is also a delivery mechanism for influence operations, and that second reading is what makes it matter outside the security team. Changing what people see is not new. Doing it through the source they already trust, to whichever groups the attacker chooses, while everyone else still sees the correct version, is. Doppelganger, the Russia-attributed campaign running since 2022, built cloned copies of real Western news sites to push fabricated stories, and it works well enough to have kept running for years. It is also findable, because the clone sits at a domain that is not the real one and reaches readers through channels a defender can trace. An operation running on ATG needs none of that scaffolding. The fabricated story arrives from the outlet’s own address, rendered by the outlet’s own page, and a reader who checks the URL finds nothing

wrong. A fact-checker opening the same page sees the original. The only people who could confirm the manipulation are the people who received it, and they have no reason to think they did. The same mechanism reaches a newsroom, a trading screen, and a grid operator's console, because all three are assembled the same way. Paired with content generated per target, that is what the term *weapons of mass deception* names.

1. Client-Side Rendering

Modern applications do not run the program their developers built. Client-Side Rendering (CSR) means an application is put together while it runs, on the user's device, instead of being fixed when it is compiled, and the *Client Runtime* is the layer that does the putting together. Everything the user sees or touches goes through it: what is on the screen, which controls work and what they do, what runs in the background, and whether a warning appears or stays quiet. The runtime pulls executable files over the network and adds them to the running application, which is what a defender would recognize as code arriving. It also pulls in content that code already running then interprets, and interpreting that content produces more executable behavior. ATG applies wherever code or content executes at the screen. It does not apply to runtimes that only run precompiled code, such as some embedded firmware and certain safety-critical avionics. Fetched code and interpreted content both end up executing. Only the first one looks like code arriving.

2. The Just-in-Time App Blueprint

A CSR application ships a plan, not a product. What gets installed is a thin shell that fetches, assembles, and runs the rest from remote sources, *and the Just-in-Time App Blueprint is that plan being built just in time at render*, out of parts pulled live from sources the application owner does not control. A blueprint in any other trade guarantees sameness: hand the same drawings to two builders and you get two identical buildings, because the drawings specify the parts. This one specifies where to go and get the parts, not what the parts are. The pattern stuck because it pays. CSR moved the cost of rendering and computing off the vendor's servers and onto the customer's device, and customers paid for it in hardware, bandwidth, and power while getting the same product at the same price. Split across billions of people, nobody noticed the transfer. That is the *Compute Tax*. Building at render implies a moment when building stops, and interactive applications never reach it. Part of what changes is the user's doing, as they click through the interface, and that part is ordinary and not a security problem. The other part is the parts themselves, which are not the same from one render to the next. Same plan, different parts, different program. What you end up with is a *Perpetual State Machine*. A finite state machine is bounded and testable, because you can list its states and walk them. This one is not, because the parts that define its behavior are not fixed. It is not one application but a new one on each machine, each time. A defender who wants to compare a suspect render against a known-good one has no known-good one to compare it to.

3. Verification Ends at Delivery

Code review, signing, dependency scanning, and supply-chain attestation all happen to files sitting still, before they reach the device. The *Client Runtime Boundary* is the moment a fetched file stops being something you can inspect and starts being something that is running, with all the permissions of the application around it. Everything defenders rely on happens before that moment. Nothing looks at the file after it. Whatever trust was granted at delivery is the only trust the runtime will ever apply. Zero Trust is an architectural model set out in guidance rather than a product anyone can buy, and NIST SP 800-207 states seven tenets. Four of them govern how a system treats something it has already let in, and a CSR runtime breaks all four, not through implementation error but because the architecture requires it. Keep verifying: the runtime verifies once at load and never again. Grant the least access the job needs: anything the runtime loads runs at the application's own permission level, so a third-party script gets the same access as first-party code and there is no way to give it less. Never extend trust implicitly: inside a runtime, loading something and trusting it are the same act. Re-decide as conditions change: the trust decision gets made once, so a new user, a different network, or behavior that should look wrong triggers no second look. All four failures travel wherever the runtime does. A mobile app that draws part of its interface in a WebView is running a browser engine inside itself, and the bridge that lets that web content call the app's native functions hands web-loaded code the app's own permissions, including its storage, its location, and its camera. Vehicle infotainment systems and control-room displays are built on the same embedded engines and carry the same four failures into a car and onto a plant floor. The problem comes from the architecture, not from the platform.

4. Herd Trust

Most of the time, the trust granted at delivery was never earned by verification. A library or a service gets into a stack because it is popular, well known, and widely used, and nobody looks at what it actually does. *Herd Trust* is that: picked for its reputation, never checked. Being widely used feels like safety in numbers. It is not. Every team adopting a popular package reasons the same way: thousands of others depend on this, so somebody must have reviewed it. Nobody in the chain is that somebody. Popularity counts how many people made the assumption, not how many people checked. This is why a free analytics library ends up in thousands of systems without anyone asking what it can reach. It is why an adversary who takes over a popular package gets every application that uses it at once, instead of breaking into them one at a time. And it is why the most valuable positions in the supply chain belong to whatever is already everywhere. A defender reviewing

dependencies spends the time on the unfamiliar package, the one added last month, the one with one contributor. The package that has been in the build for six years and sits in ten thousand other builds gets waved through, on exactly the grounds that make it worth capturing. Trusted by the crowd, never checked, and then handed the runtime's full permissions when it executes. The runtime runs on faith.

There is a step before that one. The application decides which outside sources your device will trust, and it writes that decision into its own code. Your device then carries it out. You are not shown the list, you are not asked, you cannot check what any of those sources actually sent, and you cannot switch one off without breaking the thing you are trying to use. You can refuse the whole arrangement. You cannot set terms on it. That is *Conscripted Trust*, and it is the other half of Herd Trust: Herd Trust is why the application picks a source, Conscripted Trust is why its pick binds you. You do not choose whom to trust. The application chooses, and the choice runs on your device.

5. Integration on the Glass

Every system that does something useful for a person finally does it on a piece of glass. A child sees a grandparent two thousand miles away. A surgeon sees inside a patient. An operator sees a plant on another continent. *The Glass* is that last point of contact: the surface where the assembled application meets the user, where the system finally delivers its value, and where the vulnerability sits. Control-room displays, medical equipment, AR headsets, and checkout terminals all have one. The browser gets nearly all of the research attention, which makes it the best-studied piece of glass rather than the last one. And the application does not get put together in the build pipeline, at the package registry, at the bundler, or at the CDN. Those are the places where supply-chain controls get applied. It gets put together at the Glass, on the user's device, after every one of those controls has finished its work, and *Integration on the Glass* (IOTG) is that moment. Server-rendered systems were assembled on machines the operator owned, which meant the operator could log what went into each page, keep a copy, and go back and look at it after something went wrong. CSR systems are assembled on a device the operator does not own, cannot instrument, and will never see again.

6. The Digital Supply Chain

Someone writes a piece of code. Someone else hosts it. A build system changes it. A distribution network pushes it out. A delivery mechanism carries it across the line into the runtime. *The Digital Supply Chain* is that path from producer to screen, and trust at each

handoff is assumed rather than checked: once the first party is trusted, everyone downstream inherits that trust without re-establishing it. Compare what the physical chains carry. A pallet of spinach arrives with the grower's name, a lot code, and enough traceability to find the field it came from on the day it was cut. A drug shipment carries serialized packaging and a record of every change of hands. A chip going into a defense system comes through a trusted foundry with provenance documents at each step. A script loaded from a URL carries no equivalent of any of it. There is no record of who produced it, no documented chain of custody from the producer to the device, and no mechanism to recall it once it turns out to be bad. The same is true of a configuration file fetched at runtime and of a third-party tag deployed across thousands of sites. Those three chains carry food, medicine, and hardware. This one carries the instructions that run the power grids, the hospitals, the payment systems, and the tracking that protects the other three. *The most important supply chain humanity has ever built is the least protected supply chain humanity has ever built.* Risk then piles up wherever a lot of people depend on one source. A *Digital Supernode* is a dependency sitting at one of those hubs. Own the node, own the millions who trust it.

7. URL Overtrust

A URL is a place in the supply chain, not a string that fetches a file. Like any place, it has an owner, a legal jurisdiction, someone who can write to it, and rules, or no rules, about what it is allowed to do. Calling a URL is sending a package. The response is receiving one. *URL Overtrust* is treating that place as a fixed, safe file. The mistake is not that we trust it too much. It is that we are trusting the wrong kind of thing, because there was never a file sitting there. Most people picture a warehouse: someone made the item, it sits on a shelf, and every request takes the same item off the same shelf. *The Static Resource Assumption* is that picture, and signing, hashing, attestation, and build-time integrity checks are all built on it. URLs do not work like a warehouse. Every response gets made when the request arrives, by whoever controls that endpoint at that moment, out of whatever they choose to put in it. Manufacturing on demand, not mass production.

8. The Allowlist Fallacy

Putting a source on an allowlist approves an address. It does not approve what that address will send later. *The Allowlist Fallacy* is thinking it does, and most client-side stacks are built on it. An allowlist approves the sender, not the package. It gets worse because loading is transitive. A script pulls in another script. A service worker fetches and caches whatever URLs it likes. A tag manager loads configuration that loads more code. Every one

of those is the runtime doing something, and the runtime does it without writing the relationship down anywhere a defender can look. That is *The Manifest Illusion*. Where an application loads code on the fly, generates code while running, or fetches its parts live, no complete and accurate list of what actually ran can exist. A lockfile pins the versions the build intended to use. An SBOM lists the components the build believed it included. A CSP allowlist names the sources the page is permitted to load from. All three are written before anything runs, and all three describe intent. You have a manifest from the plans, not a true manifest of what was built and runs.

9. Data as Code

Configuration says which scripts to load. Templates carry expressions that execute. JSON gets evaluated as code, dynamic imports take their target from configuration, and where a model is allowed to act on its own, its output decides what the program does next. In many runtimes, a response that looks like data is a way to inject source, and the data path turns into a code path without crossing any line a defender can see. That crossing is *Data as Code*, and it shows up anywhere a runtime takes in content. A car pulls a configuration file that decides which driver-assistance features are switched on. A mobile app asks a feature-flag service which code paths to run for this user. A productivity tool reads a plugin manifest that tells it which extensions to load and what to let them do. In each case the fetched thing is filed as settings, reviewed as settings if it is reviewed at all, and decides what the program does. Code also writes code. Running a string as code, building a function while the program is already running, compiling a downloaded block of bytes into something executable: all of it produces behavior that appears in no source file, no build output, and no stored file. That is *Code That Writes Code*, and what it produces exists only while it is running. An investigator watching the runtime execute can see it. An investigator reading the source or the stored files cannot.

10. The Two-Stage Attack

Supply Chain Injection is adversary-controlled content getting into a trusted delivery path, and ATG is the version that targets the screen. It can happen at the cache, in transit, at the configuration source, at a data endpoint, or in a response from a model. In every one of those cases, the swap happens when the content is delivered, not to the file that was checked. *The Two-Stage Attack* is the hardest version to catch. Stage one is the visible code, the script or bundle a reviewer, scanner, or signing process looks at. It is clean on purpose, and there is nothing in it for a pre-delivery control to find. Stage two is the data that code goes and gets when it runs, from a configuration service, an analytics endpoint,

or a model. That endpoint is the one the adversary controls. Checking stage one tells you nothing about stage two, because what was reviewed is the bundle and what happens is whatever the data tells the bundle to do. The file a defender can inspect is not the thing whose behavior matters. The weapon is in the data, not the code.

11. The Runtime Injection Advantage

Anything injected at runtime shows up after every pre-delivery check is finished, and the runtime cannot redo those checks, because it does not have the thing they checked against. It has only what was delivered, and what was delivered is what runs. The injection can also be aimed: one channel can serve a clean response to a scanner or a researcher and a different one to the target. That makes it hard to reproduce afterward, because an adversary who stops serving the bad version leaves an investigator nothing to find. And it inherits, automatically, whatever trust the runtime gave the source when it loaded. Arriving after the checks, aiming per target, leaving nothing to reproduce, and inheriting trust: together these four are *The Runtime Injection Advantage*, and they are why ATG operations are hard to catch with the tools defenders have and hard to reconstruct once they are over. Checking the delivered file does not reach any of it. Watching how the runtime behaves would, and no runtime does that by default.

12. Content Swapping

What a supply-chain position is worth depends on what gets changed on the screen. *Content Swapping* covers the changes an operation can make there: a number, price, or status shows a different value; an image or video shows something the application never asked for; a link goes somewhere else; content the application did ask for never appears; the numbers behind a chart are altered, so the chart is drawn correctly from figures that are wrong. None of this requires the application to break. It keeps working correctly by the only measure it has, which is that it displays what it was handed, and what it was handed is the thing that changed. The change can also arrive later. *Post-Load and Post-Trust Modification* is the latest point it can arrive: code already running in the same context rewrites a value in memory that came in over an authenticated connection and passed every check on the way. Who gets any of this is decided by ordinary advertising and personalization machinery. *Targeted Delivery* keys the response to IP address, device, logged-in identity, or which test bucket someone is in. It is not the ad; it is the ad network. The same targeting that picks your ad can pick who gets the swap.

13. The Integrity Blindspot

Nobody can tell, at the moment it matters, whether what is on the screen is what should be there. That is *The Integrity Blindspot*, and it applies while the content is running and after it is gone. Nothing checks at execution time that a fetched resource matches a known-good reference, so a live check has nothing to compare against. Nothing gets recorded that would let someone reconstruct a swap later, because the swapped resource lived in memory for one session and disappeared when the application closed, and because the connection was encrypted, a network monitor sitting between the device and the source recorded which address was contacted, when, and how many bytes came back, but not what came back. Only an organization that deliberately breaks and remakes the encryption at a proxy sees the content, and most do not. Both problems come from *The Missing Source of Truth*: the application owner declares what a resource is supposed to be at build time, and nothing keeps that declaration current at the moment of delivery. An adversary can close that route deliberately. When an investigator wants to know what a device actually loaded, the browser cache is where they look, because it keeps a copy of fetched files on disk after the session ends. A no-store header in the response tells the browser to use the file and keep nothing. The bad version runs, and no copy of it is written to disk. The cache is clean because it was never used. Absence of evidence is not evidence of absence, and no claim about how widespread ATG is escapes that.

14. Runtime Anarchy

Security controls are concentrated at the door. CORS decides what can be reached across origins, CSP decides what can load and run, and SRI checks that what arrived is what was expected. Once a script gets through and starts running, nothing governs what it does. Every script in an execution context is equal to every other one, free to read and change shared state, the page's DOM, global variables, and other scripts' data. There is no per-script sandbox, no ranking of privileges, and nobody enforcing limits. That is *Runtime Anarchy*. CORS and CSP enforce rules about who gets through the door. Once a script is in the room, there are no rules. A script that gets in reaches everything the application has: its storage, its network endpoints, its camera and microphone where the user granted them, and the sensors and actuators exposed in vehicle, AR and VR, and control-system runtimes. A compromised analytics script can steal payment data. A compromised payment script can capture credentials. The runtime cannot tell that a script is doing something it was never meant to do, because the runtime has no idea what it was meant to do.

15. D5 Effects

How an attack gets delivered and what it accomplishes are two different questions. ATG operations produce five effects. Deceive gets a consumer to accept as true something the system did not mean to show, and to act on it. That consumer is usually a person, and it can equally be an alarm check, a control loop, an agent, or a model, none of which judge whether what they are given is true. Disrupt interferes with a function, so the control is still there and no longer does what it is for. Degrade cuts speed, volume, scale, or response time, so the system does everything it should and does it worse. Deny makes something unavailable at the screen: the control is gone, the route does not resolve, the list that comes back is short. Destroy is code reaching through the screen into a system behind it and damaging that system, so that it cannot work again until it is restored or rebuilt. Deny stops at the screen; Destroy reaches through it.

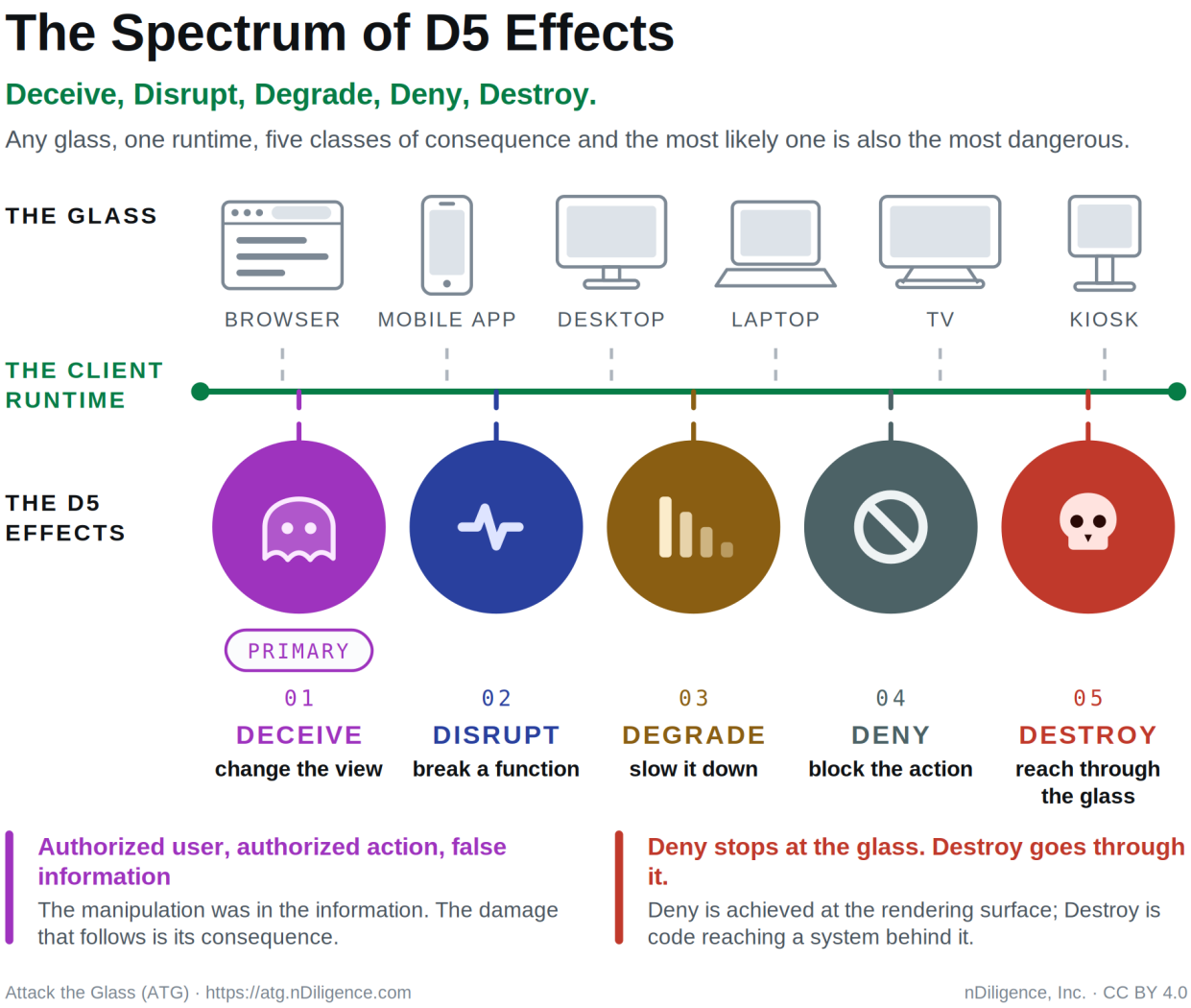

***Figure 1**: Five effects an injected resource can produce. The set is an index, not an escalation.*

The five are not separate kinds of attack, and they are not five degrees of one thing. An operation can use them one at a time, combine them, or run them in sequence, and the whole set is what D5 effects refers to. Any of them can be aimed at one device, at a defined group, or at everyone using the application. Which of them are available at all depends on what the application does and on what the session is trusted to do. A reference page offers no way through to anything behind it. An administrative console offers a way through to a great deal.

Deceive matters most in practice. Changing the data, content, media, and features a person sees takes the least capability, returns the most for the effort, and works below the level at which the user could check anything. Breaking the application outright, by contrast, takes more skill to pull off, gets noticed because something visibly stops working, and produces less damage when it does happen. And the same trust works twice. It gets the operator to act on the false reading, because the display is the authority. Then it protects the action afterward, because the reviewer asking whether the operator did the right thing looks at the same screen, sees the same value, and finds that they did. When that action breaks something, the effect was Deceive and the breakage is its consequence. The system did what it was told. What it was told rested on something false.

16. Placement and Access

Every ATG operation starts with a position. *Placement and Access* (P&A) is what an adversary has to have: control over something the target's runtime depends on, often something the target does not know it depends on. Breaking into the target is not the route, and usually it is not necessary. What an adversary can do follows from the position, not from how big they are. A developer inside a widely used analytics SDK reaches every site that loads it. An engineer on a mobile over-the-air update channel reaches every device on it. A position used once is an incident. Use the same position repeatedly, across many of the people who depend on it, and you are no longer exploiting a system so much as steering what a set of people see and decide, which is Influence. Hold it long enough and widely enough that the affected population has no unaffected reference to check against, no second screen and no second source, and the steering becomes their only input. That is Control. What separates the three is reach and duration, not technique. Adversarial Placement and Access (APA) is when the party holding the position is hostile to the people depending on it, and an Adversarial Influence Operation (AIO) is what they do with it. A position *Lying in Wait* is the hardest thing for a defender to find, because everything observable about it is genuine. The company ships real software to real customers, bills them, supports them, and passes vendor review, because all of that is true. The only hostile

part is who owns it, and ownership is not something a scanner reads. The threat is there and doing nothing, and there is no event to detect.

17. The Open-Source Open Door

Most of what runs on the Glass was written by people nobody at the target has ever employed. Public dependency analyses put the third-party open-source share of a typical bundle far above the share the application's own team wrote. Getting a position inside an open-source project is cheaper than any other way into the supply chain. Getting hired into a vendor means interviews, identity checks, a background check, and an employment record that ties a real person to the access they are given. Acquiring a vendor takes capital and produces filings, contracts, and a change of ownership other people can see. Open-source projects ask for maintainers in public, accept a history of useful contributions as the whole credential, and in plenty of cases hand over control because the existing maintainer agreed to it in an email. Nobody verified an identity, and nothing about the handover was recorded anywhere a downstream consumer would look. That gap is *The Open-Source Open Door*, and XZ Utils went through it end to end. Nothing was broken into. The door was open by design. Provenance then disappears at the build. Dependency tools read the manifest; the runtime runs the bundle, which contains the dependencies of those dependencies, code the build tools generated, and code the bundler copied in directly, none of which the manifest lists. *Origin Laundering* is that loss. The bundle runs as one unit but carries no record of where its parts came from. The runtime cannot tell a dependency from code the application owner wrote.

18. The Silent Transfer

A defender cannot treat a position as hostile without knowing who is holding it, and for digital infrastructure there is usually no way to find out. Real estate requires a number of public records, helping us validate who owns and controls properties and buildings. Companies are required to register in various government databases so the public can validate who controls a business. Financial institutions have to report transfers above a set threshold, so large movements of money leave a trail an investigator can follow. None of these public disclosure pathways exist for who controls deeply trusted internet resources and infrastructure. Nobody has to record the sale of a CDN, the handoff of a package to a new maintainer, or the transfer of a browser extension to a new publisher, and no registry anywhere notes that it happened. Even where the owning company can be identified, the people who hold commit access and publishing credentials are frequently not the people who own it, and an operator can replace those people without anyone outside noticing.

Adversaries work inside the resulting *Ownership Opacity*, buying a trusted asset, running a campaign through it, and moving to new infrastructure before anyone can attribute it. *The Silent Transfer* is a change of hands with no technical signature: a company sale, a stolen credential, a namespace handoff, a DNS change. None of those alter the URL, the hash, or the certificate. Everything a scanner looks at stays green. The URL never changed. The person behind it did.

19. Tailored Deception

AI does not create a new kind of vulnerability. It makes the existing one worse in two ways. A model loaded into the runtime is one more file the runtime fetched and trusted. And an inference endpoint, the address an application calls to get an answer from a model, is a data endpoint like any other, so its response is stage two of a Two-Stage Attack with a model in the middle. The main technique against it is prompt injection: content arriving in the model's input gets read as an instruction rather than as data, so whoever controls the input controls the answer. The runtime knows the response arrived. It has no way to tell whether the response is what the application owner would have wanted the model to say. Put those together and you get *Tailored Deception*: generative AI running in the runtime, producing content aimed at one person, on that person's device, at machine speed, using the files, history, contacts, location, and app state the runtime is already allowed to read. Every defense we have against spam, fake websites, and mass phishing depends on many copies sharing something in common. Content made for one person shares nothing with anything. Each instance is a population of one. And when the deception is manufactured by the user's own device, the user's trust in that device becomes part of the attack surface.

20. Malice Without Malware

Existing defenses look for malicious code coming from a malicious source. ATG produces a malicious result with neither: clean code from a source everyone trusts. A scanner comparing that code against known-bad signatures finds no match, and a monitor watching for traffic to known-bad destinations sees a routine request to an approved one. Nothing fires, because there is nothing present that the tools were built to recognize. That is Malice Without Malware. The malice is real; the malware is absent. Each of the partial defenses does real work and then stops somewhere the architecture keeps going. CSP makes an attack meaningfully more expensive in the browser; it checks sources rather than what those sources send, so an approved source that changes hands delivers bad content inside the policy. SRI genuinely protects a static file fetched from a known URL; it checks the file, not the data the file goes and gets. Certificate pinning fixes the identity of an

endpoint rather than the content it returns, so an adversary who comes to hold that endpoint satisfies the pin and delivers straight through it. Zero Trust raises the cost of getting in substantially; it stops at the moment of load. Server-Side Rendering really does reduce exposure, because the page gets assembled on the operator's own servers and arrives at the browser already built. The trade-off is that such pages respond more slowly and cost more to serve, so most teams do not stop there. They ship the server-built page and then load JavaScript that reattaches behavior to it in the browser, which is called hydration, and that step pulls the same third-party code from the same sources into the same runtime. Most of what SSR removed comes back. All of them are worth deploying, and none of them is enough against ATG. Because the problem comes from the architecture, it turns up on its own in every runtime built that way, and that is The Endemic Flaw: not an infection that spreads, but a pre-existing condition every Client Runtime is born with. There is no patch for CSR. There is no "out."

Bottom Line

The screen is where decisions are made and where the defenses are thinnest. Verification stops at the point of delivery and then never resumes. Injecting at runtime has structural advantages that checking files before delivery cannot reach. And what defenses exist are concentrated in the browser, leaving everything else largely undefended.

Polyfill.io shows content being swapped for chosen targets through a trusted node someone bought. Magecart shows injected client-side scripts reading and stealing what the page displays. XZ Utils shows the maintainer-takeover route working; it does not show that route reaching a client screen. The Arup fraud, in which a finance officer approved transfers worth about twenty-five million dollars after a video call with colleagues who were all synthetic, shows the kind of payload an ATG operation would carry rather than the delivery route that would carry it. The Two-Stage Attack follows from the architecture and has been partly demonstrated by payloads seen in the wild that stayed dormant until they recognized the right target. Whether or not ATG is being run at scale today by well-resourced and nation-state actors is an assumption adopted for operational reasons, not an established fact.

The Great Cannon attack in 2015 shows a nation-state executing an ATG attack. Requests heading for servers that host common analytics and advertising scripts were intercepted, and a fraction of them, picked by address, came back with different JavaScript than the one that was asked for. Browsers outside China ran it and were used, without their owners knowing, to attack two websites. Researchers at the Citizen Lab attributed the system to the Chinese state. The way in was different from Polyfill.io: nobody bought or broke into the script's owner; they sat on the wire in between. Everything after that point is the same, and it is the part that matters for ATG. Those scripts were sent unencrypted, and the researchers noted that encryption was the fix. Encryption helps but it is not the whole answer, because it protects the wire rather than the two ends, and the delivery networks that carry most of the web decrypt at their own edge by design. Encryption changes who the party in the middle can be. It does not remove the position held by the site's own delivery provider still holds the content in readable form before passing it on.

How widely ATG is actually deployed cannot be measured with the instruments that exist today because deception leaves nothing for signature-based detection to find. Not detecting something is not the same as it not happening. Answering it would take three things that do not exist yet: monitoring of how the runtime behaves at the screen, telemetry at population scale, and visibility that reaches past the browser into every other runtime. That is an engineering problem and no Client Runtime ships it today. Building it decides whether tailored deception at scale stays a described risk or becomes a routine one.