ATG: Remediation Playbooks

Remediation Playbooks for 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

Introduction

Your customers and your staff make decisions based on what a web page, an application, or an equipment panel shows them. They allocate resources, approve payments, reset credentials, read balances, and shut down equipment on that basis. The systems behind those interfaces then carry out what was decided, often with no further authorization. Protecting those interfaces is as important as protecting the systems they drive.

Browsers, mobile apps, point-of-sale terminals, operator consoles, kiosks and vehicle displays all build the interface on the user's device in real time, pulling code, content, and resources from external providers as they go. Your own developers have coded part of the application. External suppliers control the rest, and their code and assets run with the same privileges. Any of those suppliers can change the resource it returns, at any time, to some users and not others. There is no easy patch, because nothing is malfunctioning: client-side rendering is the modern approach for building user experiences on demand, in real time, on the user's device. Attack the Glass is the vulnerability this design creates, and no patch removes it, so an organization reduces it instead.

Remediation sorts into three groups by what each change costs to make. An engineering team can act alone on the first group: inventory what loads, remove what nobody will vouch for, pin and constrain what stays, and put a manual procedure where no technical control reaches. The second group waits on an engineering cycle or a procurement round: assembling less on the client, governing the pipeline that feeds the build, extending browser-grade controls to the surfaces that are not browsers, and establishing who owns and operates each remaining supplier. The third group is not project work at all, and lapses without a named owner and a review date: architecture standards, continuous ownership monitoring, contract terms, and metrics.

None of it shows anyone what actually ran on a customer's device. Current client-side platforms provide no way to monitor what gets built there, and after the fact there is usually nothing left to examine. That limit is real, it is stated plainly at every measure, and The Residual sets out what would have to exist to close it. Start with removal, on every surface, because a component you never took on cannot be sold to someone else, cannot be compromised, and cannot quietly start returning something different.

Short-Term: 0 – 90 Days

The first ninety days cover four kinds of change, and an engineering team can make all of them without rearchitecting anything or reopening a contract. Find out what loads. Remove what nobody needs. Pin, constrain and read what stays. Put a manual check in front of the decisions that move money or equipment.

Set one expectation before starting, because it changes what your team looks for. The code doing the damage is usually valid, well-formed, correctly signed code from a source everyone agreed to trust, doing something it was never meant to do. Nobody here is hunting for a bad file. You are reducing how many outside parties can put a file on your interfaces at all, and making each one that stays accountable to somebody.

Third-Party Script Inventory

Start where the consequences concentrate: login pages, payment pages, transaction approval screens, operator dashboards. For each one, list every script, style, font, frame, pixel and tag the browser loads, and the address each one comes from. Browser developer tools produce that list for a single page. Crawler-based scanners produce it across a property. A Content Security Policy in report-only mode produces it continuously, from real user sessions, which makes CSP a discovery tool before it is ever an enforcement one.

Expect the count to exceed what your application declares, and treat the gap as the finding. Loaded code fetches further code. A tag manager injects whatever its containers hold at the moment of the request. Neither shows up in a build-time manifest. Report-only CSP is the cheapest way to measure that difference, because it reports what the browser attempted rather than what the repository claimed.

Give tag managers their own pass, because a tag manager lets other people add code without asking you first. List the containers, list who can publish through them, and list what each container injects. In most mature deployments the publish list is longer than anyone expects, and the container contents have outlived the people who added them.

Run the same exercise on the build pipeline using a software bill of materials. SBOM tooling enumerates the open-source dependency tree that becomes your shipped bundle. The file it produces is not the goal. Knowing whose code runs on your Glass is the goal, because most of what runs there was written by people nobody on your payroll has met.

Rank what you find by what each resource can reach, not by how large its vendor is. For each one, record what it can touch once it loads, who operates the address it comes from, and what breaks if you remove it. A small vendor with code inside a payment flow outranks

a large one loading a font on a marketing page. That ranking drives every removal, pinning and self-hosting decision you make after this.

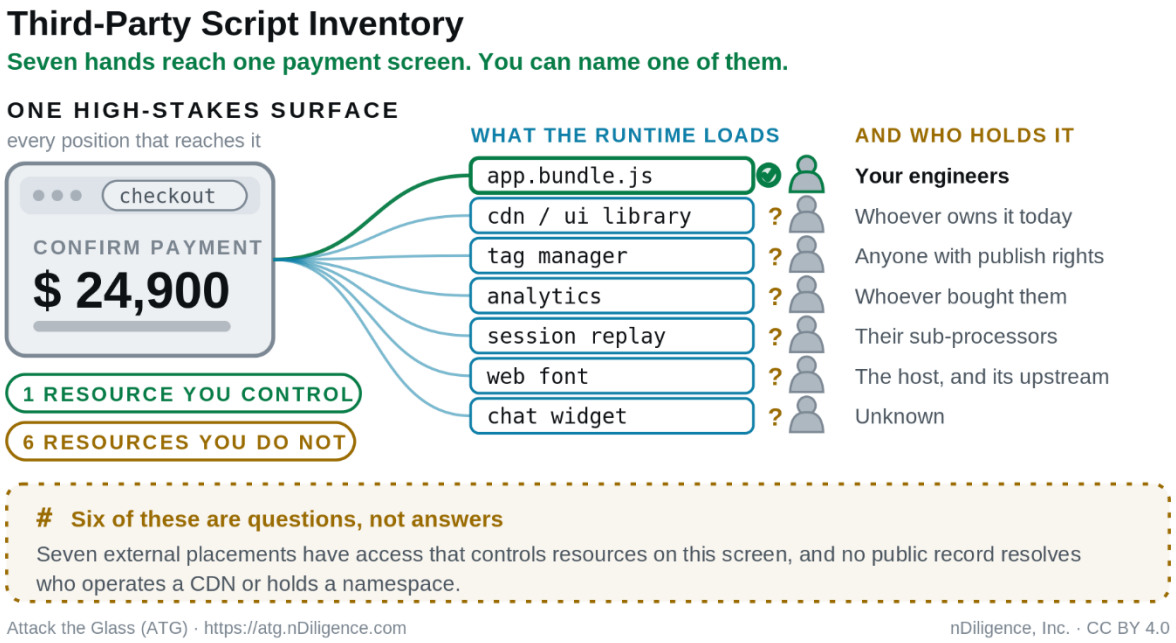

Figure 1: Third-Party Script Inventory

You now have a number for each high-stakes interface: how many outside parties reach it. From here you make that number smaller and hold the remainder accountable.

Removing Unnecessary Third-Party Code

Every third-party resource is code somebody else controls, running on your interface, changed at their discretion. Fewer of them is the cheapest remediation available to any organization.

Delete unused and orphaned scripts first. Most mature web properties carry tags whose owners have left the company and whose purpose nobody can state. Removing a resource nobody will claim needs no analysis.

Ask one question of everything that survives, and it is not a technical question. Most of these were added by reputation: a library or a service gets adopted because it is popular, well branded and widely used, and nobody verifies its behavior or its intent afterward. Widespread adoption feels like herd immunity and is nothing of the sort. So for each remaining resource, ask what anyone actually established about it before it was let in, and treat "everyone uses it" as a missing answer rather than an answer.

Hold high-stakes pages to a stricter standard than the property at large. A payment page does not need a chat widget, a social pixel or a heat-mapping script. Interfaces where a substitution would do the most damage carry the smallest dependency set the business function tolerates, and every resource left on them has a named owner and a stated purpose.

Freeze additions while the cleanup runs. Require sign-off for any new third-party resource until the inventory settles, so the list does not grow behind the team reducing it.

A component you never took on cannot be sold to someone else, cannot be compromised, and cannot quietly start returning something different. Removal is the only change in these ninety days with no structural limit, because everything else here manages risk that is still there and this one deletes it.

Pinning, Self-Hosting, and the CDN Question

Whatever survives the cull gets pinned, brought in-house, or both.

SRI confirms that the fetched resource matches a hash you already trust, which closes the gap between what you reviewed and what was served, for that resource, on that load. It does real work wherever content is static and versioned. Two limits are structural. SRI breaks where content updates continuously or is generated per request, because no advance hash exists to check against. And its declarative form covers script and link elements only. A resource fetched by already-executing code can carry integrity metadata, because the Fetch standard gives every request an integrity field, but that check is opt-in per call, required by nothing and audited by nothing. Apply SRI everywhere content can be pinned, require the integrity field on runtime fetches your own code makes, and write down every place you could not apply it as a flagged gap rather than a silent one.

Self-hosting goes further, and it removes an external supplier from your load path faster than anything else available today. Pull the library once, review the copy, and serve it from infrastructure you operate. Your users stop calling out to anyone else for that code, often within days of the decision. The cost is real and worth stating plainly: you take over configuration and update management by hand. New versions arrive through a review you schedule instead of whenever the remote endpoint ships them. That cost is the point. Approving a source is not the same as approving whatever that source later returns, and once you serve the copy yourself there is no later return to approve.

Self-hosting removes the supplier from your delivery path, and that is the whole of what it removes. It says nothing about where the copy you took came from, so review that copy before you serve it. And a self-hosted script that still fetches data from third-party endpoints at run time leaves that route open, because the change rides the data rather

than the file. A hash-valid, verifiably clean script does the damage using whatever its endpoint sends it at execution. Self-hosting a script does not self-host its endpoints.

Your CDN contract is also something you assigned, which means you can reassign it. Migration between major CDN providers is a well-worn operational path measured in days to weeks, and it is the right move when diligence turns up an operator you would not choose today. Frame the move accurately: switching providers substitutes one company for another rather than removing anyone from the path, so the ownership question travels with the migration. Ask who operates the infrastructure you are moving to. And migrating covers only the CDNs you contract with. A third-party script chooses its own delivery path, and the remedy for that path is self-hosting or proxying, not your CDN agreement.

Pin your versions. Exact dependency versions, committed lockfiles, no floating ranges. An auto-updating dependency approves a package name once and then accepts whatever that name later resolves to. Impose a cooldown on upgrades, adopting new versions days or weeks after release rather than immediately, so a compromised release surfaces in the ecosystem before it reaches you. Disable package install scripts where your toolchain allows it.

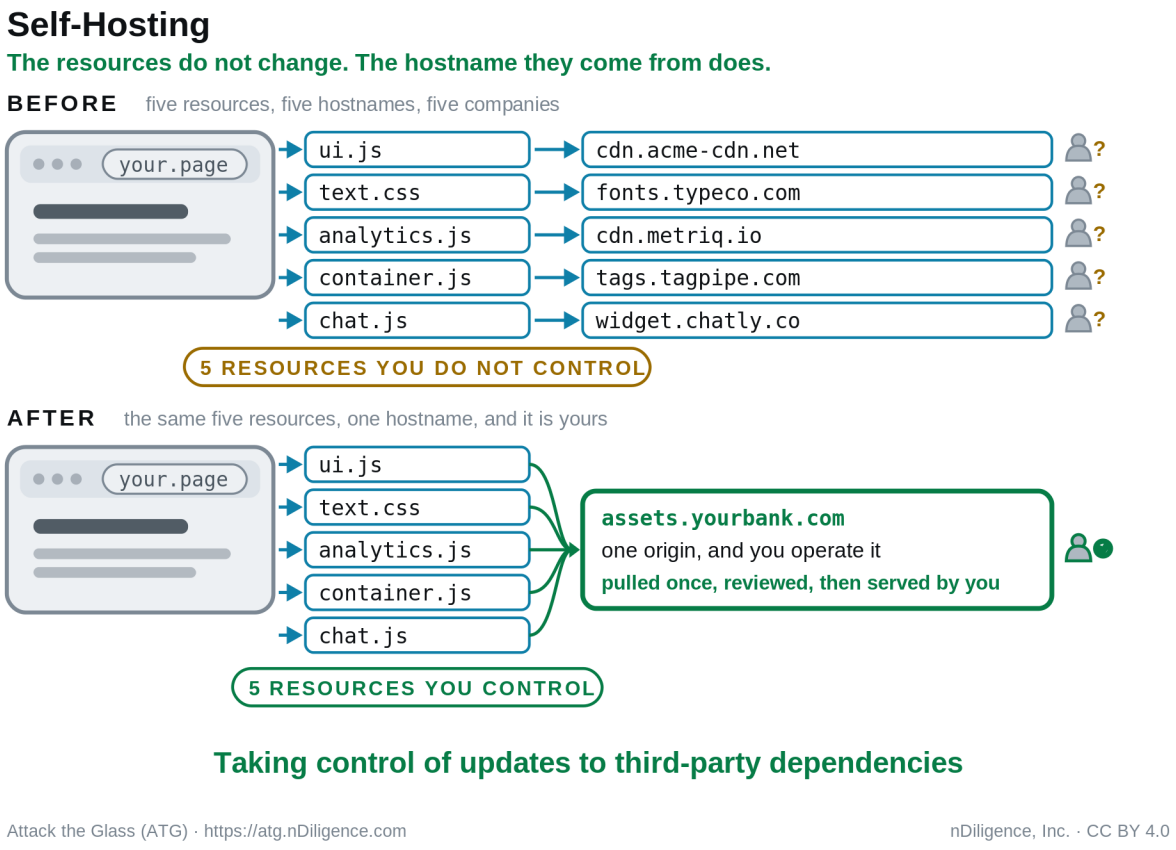

Figure 2: Self-Hosting

Pinning and self-hosting are worth deploying and they move real risk quickly. They verify and control what gets loaded. They do not observe what loaded code then does.

Content Security Policy and the PCI DSS Pattern

Content Security Policy limits which addresses a browser may load from. Its value is real. It removes whole classes of injection outright, and its violation-reporting channel is the cheapest continuous telemetry a defender currently has on what a page tries to load. Deploy it report-only first, then enforce. Its limit is equally real. CSP governs which addresses are approved, not what an approved address sends back, and not what the returned code does once it runs. A URL does not return one fixed resource to everyone, and checking one fetch does not protect any later fetch. An allowlist of addresses does not touch that, because the response is built per request by whoever controls the endpoint at that moment.

Content Security Policy Level 3 lets a hash source expression match an external script when that script element also carries matching integrity metadata. That is genuine content pinning inside the policy rather than address approval. It reaches script and style elements only, it requires the integrity metadata alongside it, and it is checked at load like everything else. Use it on high-stakes interfaces where content is stable enough to pin, and do not let it stand in for coverage it does not have.

PCI DSS 4.0, in requirements 6.4.3 and 11.6.1, obliges script inventory, authorization and change detection on payment pages, and a commercial tooling category has grown up to meet it. Payment pages are where the rule applies. Any interface where somebody moves money or operates equipment deserves the same three verbs: inventory, authorize, detect change.

Constraining where content comes from is worth deploying, and it stops at what that content does.

Static Review and Sandboxed Execution

The inventory tells you what loads. Static review and sandboxed execution tell you what the survivors do. Both are available now, and both hand you findings a person still has to judge rather than verdicts. Every pattern below has legitimate uses, so its presence means look closer, not convict.

Search retained scripts, first-party and third-party alike, for **`eval`**, **`new Function`**, string arguments to **`setTimeout`** and **`setInterval`**, and **`document.write`**. These are the mechanics of code generating more code while it runs, so that what ultimately executes

never existed in a form anyone could inspect beforehand. Existing linters and static-analysis rules surface them in minutes.

Give decoding of fetched data the most respect. **`atob`** and equivalent base64 or hex decoding applied to content fetched at run time is how fetched data becomes executable, or becomes displayed truth, at the moment it is read. It is also how a change arrives in the data rather than in the file. A clean file that decodes whatever its endpoint sends has moved the weapon into data that no pre-delivery inspection will ever look at.

Remove **`'unsafe-eval'`** from your CSP. The browser then refuses string-to-code execution outright, which converts one of those indicators into an enforced control. Test in report-only mode first. Some legitimate libraries still depend on it, and each one that does is worth knowing about.

Finish with dynamic analysis. Execute third-party scripts in an instrumented environment, either a headless-browser harness with a mutation observer or the commercial client-side protection tools you may already have deployed for the PCI requirement, and record what each script touches: form fields, displayed values, links, injected and removed nodes. Baseline that behavior and alert on drift. State the limit without apology: a sandbox is served whatever variant the endpoint chooses to send it. The endpoint decides what your scanner receives too, and conditional delivery is the practice built on exactly that, so clean behavior in your lab is evidence about production rather than proof of it. Baseline-and-drift is worth having anyway. It catches the broad case and raises the cost of the narrow one.

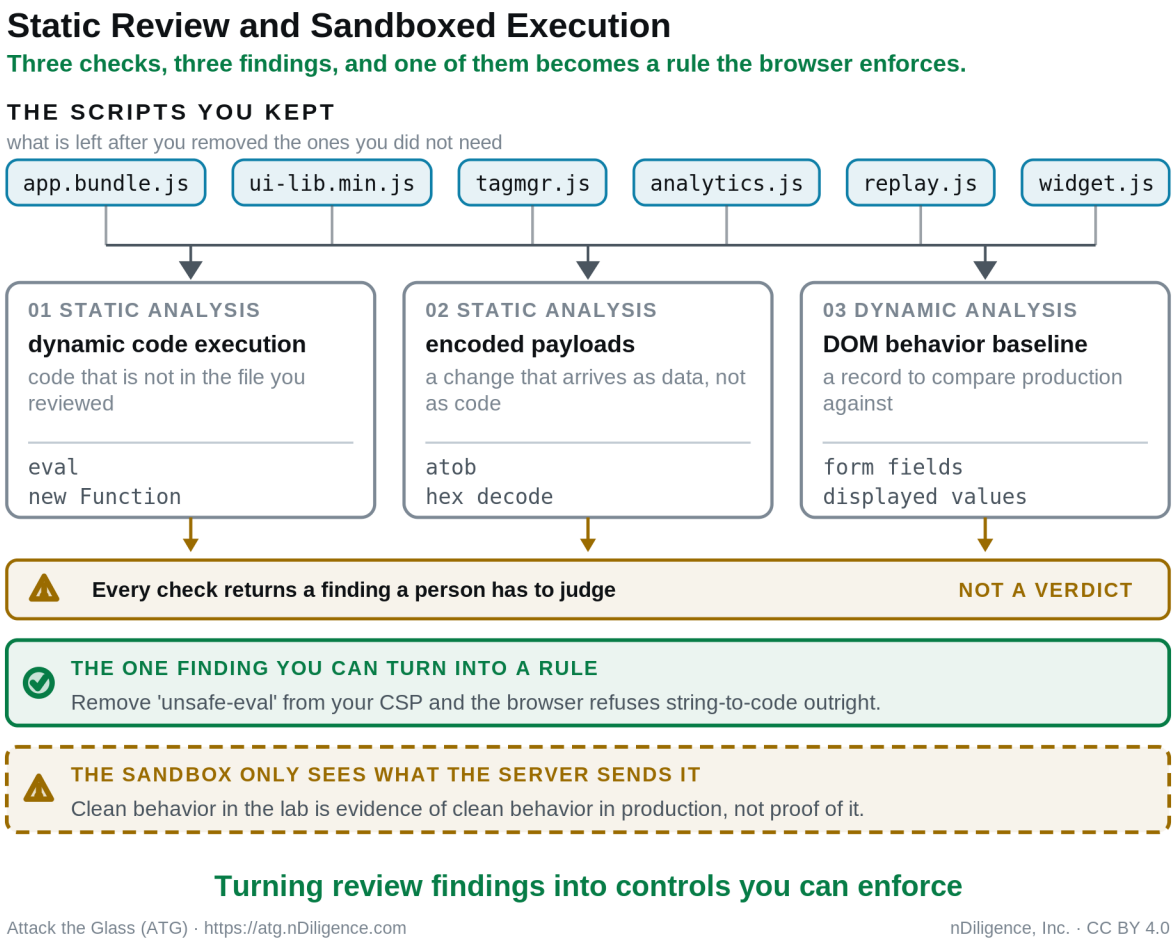

Figure 3: Static Review and Sandboxed Execution

Reading the survivors is the closest these ninety days get to watching behavior, and it examines copies served to an observer rather than the render in front of your customer. Deploy it for what it is.

Monitoring Suppliers for Change of Control

Suppliers you cannot remove can at least be watched for a change of owner. Polyfill.io became an attack through exactly that: the cdn.polyfill.io domain changed owners in 2024, and the existing script tags pointing at it kept fetching from the same address afterward. The infrastructure changed hands. The trust did not. You are watching so that you see the next one in time.

Monitor the domains and endpoints your inventory names. Registration changes, expirations and certificate transparency events, for every third-party origin that reaches a high-stakes interface. Subscribe to supply-chain and vulnerability advisories covering the packages your SBOM names. Both are commodity capabilities today.

A change of owner leaves no technical signature. A company sale, a stolen credential, a maintainer or namespace transfer, a DNS repoint: none of those alter the address, the hash or the certificate, so everything a scanner measures stays green while the party deciding what that asset serves becomes somebody else. And the interval that follows is what defeats detection. An acquired asset is typically operated normally for months or years, serving exactly what its dependents expect, because a normally operating service produces nothing to find. Often the first detectable event is the attack itself. Together they say what this watch is for. It watches ownership and control rather than content, and its whole value is that it can fire during the interval when nothing else can.

Then look at your own property the way an investigator would. Schedule fetches of your high-stakes pages from different networks, geographies and device profiles, and compare the results against each other. A difference between vantage points is one of the few observable signs of conditional delivery you can currently produce, and it works best against the widest cases: a swap served to everyone, or to a whole country, shows up as soon as one vantage point sits inside the target set and another sits outside it. The limit is honest and it stands. A swap conditioned on anything your vantage points cannot express, such as who is signed in, what they did before, which group they were put in, or which experiment bucket they landed in, reaches that group and no observer you operate, down to one named person on one device. Run the comparison anyway. It raises the cost of broad substitution, and the mid-term horizon grows it into an automated program.

Watching for a change of owner turns something that leaves no technical trace from a historical explanation into an operational warning. It shows you that control moved. It does not show you what the new owner does with it.

Out-of-Band Verification Procedures

Attack the Glass deceives a person into acting correctly on wrong information. A trained, authorized operator does exactly the right thing in response to a false reading, and no system anywhere records that anything went wrong. Until rendered content can be verified technically, your countermeasure is a second, independent path to the truth, and that countermeasure is procedural, cheap and available now.

Give irreversible actions out-of-band verification. Confirm wire transfers, payment-detail changes, credential resets and destructive operational commands through a separately established channel before execution, never through the screen that requested them. In 2024 a finance employee at the Hong Kong office of the engineering firm Arup transferred roughly twenty-five million dollars after a video call in which every other participant was a synthetic likeness of a colleague. The screen was accepted as sufficient authority. The procedure exists so that no single rendered surface is ever sufficient authority again.

Give critical operational readings a second source. Where a display drives a consequential decision, whether that is a grid state, an account balance or a system alert, name the independent source that confirms it and the moments when that check is mandatory. The check does not have to be continuous to work. It has to exist at the moments that matter.

And give every operator, administrator and finance officer one sentence of standing doctrine: a screen that contradicts a system of record is a security event. Give that report a named destination that takes it seriously. The person staring at the discrepancy is, for now, the only sensor you have deployed at the rendering surface, and the reporting path is what makes that person part of your defense.

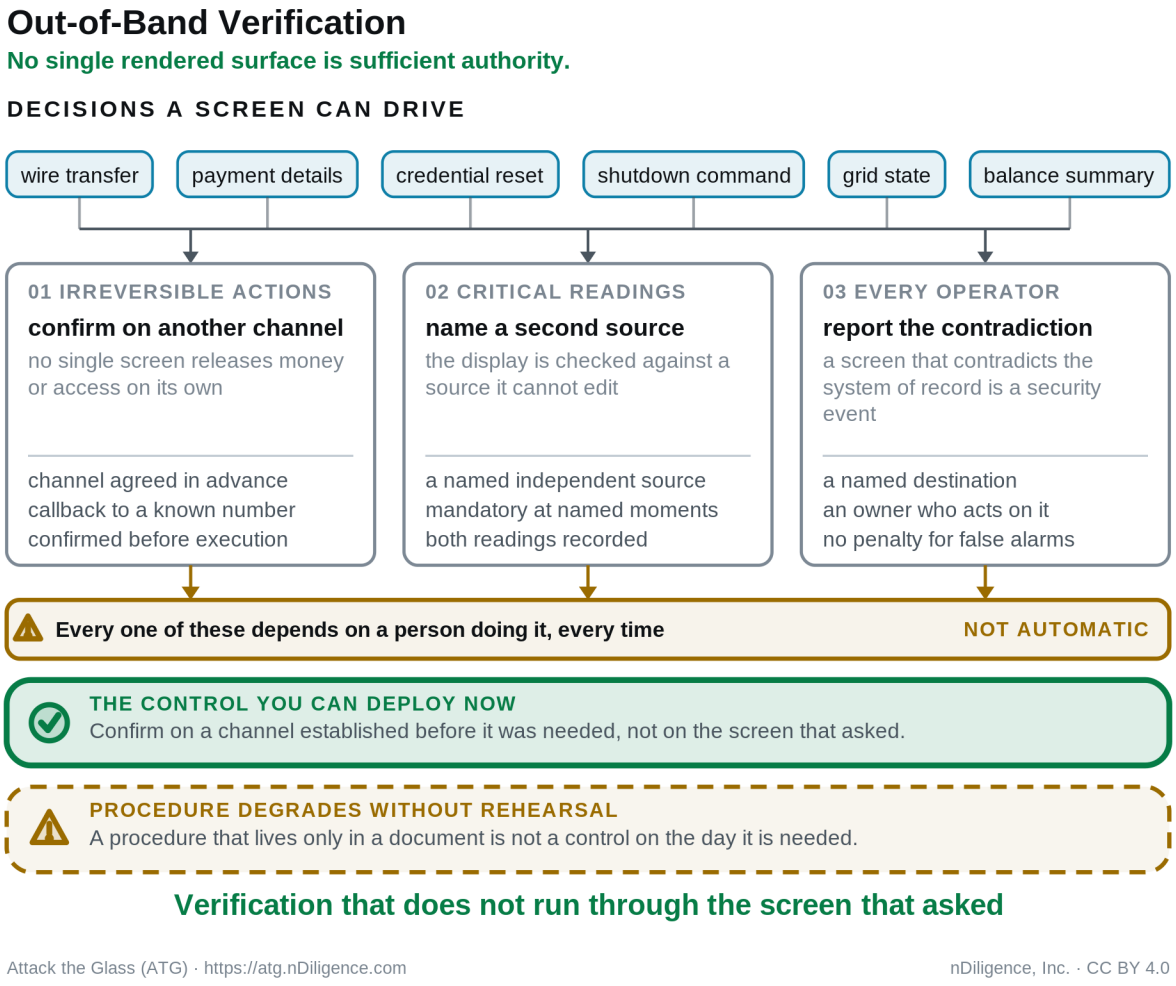

Figure 4: Out-of-Band Verification

Procedure is your one defense at the same layer as the attack, the layer where a human decides. It is also the one that degrades fastest without rehearsal, which is why the mid-term horizon puts it on a tabletop.

Briefing Leadership, Finance, and Operations

The people most worth deceiving are the ones who approve payments, administer systems and operate infrastructure, and in most security programs they are briefed last or not at all. Correct that in the first ninety days. Leadership, finance and operations hear the shape of the threat in plain terms: what a substitution looks like, why a trusted source and a clean-looking page prove nothing, and which verification procedures now apply to them personally.

An attacker chooses who receives the change, and the range runs from every user of an application, through a country or a language community or a group defined by behavior, down to one named person on one device, with different people served different content at the same moment. And the code that delivers it is usually not malware, so your tools stay quiet.

Put the synthetic-likeness video call, the altered payment page and the false dashboard reading alongside phishing in your standing curriculum, under one rule that covers all of them: verify through a second channel before acting on a screen alone.

What Carries Forward

Ninety days of disciplined execution leaves you a shorter list of outside suppliers, each one better held, watched for a change of owner, with a second path to the truth wherever a screen can move money or equipment. Each high-stakes interface now has a count, and somebody owns it.

None of that changed how your interfaces get built. You governed the suppliers. You did not change how a browser or an app puts your interface together on somebody's device. Sandbox analysis and vantage-point comparison both examined what was served to an observer, which is not the same as what renders on a targeted device. That is what the mid-term horizon takes up.

Mid-Term: 3 – 12 Months

What you ship is not a finished application. It is instructions the user's own device follows to put the interface together as it renders, from parts you do not control and cannot inspect at the moment they arrive. The first ninety days governed those parts without touching how they get put together. These months change how it gets built.

Build less of the interface on the client. Govern the pipeline that delivers what does get built. Extend the browser's controls to every other surface where something renders for a person. Find out in depth who each remaining supplier actually is. And detect what can be detected while rehearsing for what cannot.

Reducing Client-Side Composition

Your exposure scales with how much of an interface gets built on the user's device out of parts fetched from elsewhere. Building it there has permanent economics: pushing rendering and computation onto the user's own hardware is what made the approach worth adopting, and it is why nobody is giving it up. How much any one interface leans on it is still a choice you make per interface, and the highest-stakes ones should lean on it least.

Assemble the content on infrastructure you control and deliver it finished. For a browser-hosted interface that can bear the user-experience trade-off, server-side rendering is the strongest defense available today. Your users' devices stop doing that assembly for that content, the gap between what your application declares and what it actually loads largely closes, and code writing more code on the client largely disappears. Two limits keep it partial.

Applications described as server-rendered usually hydrate, meaning the client attaches behavior to the delivered HTML and takes over from there, loading further scripts and pulling in third-party content, which brings the exposure back in proportion to how much the client resumes doing.

Pure server-side rendering with no client-side scripting is the strongest form, and few real applications meet that bar. The server-side build has its own supply chain too. For a transaction approval screen or an operator console, moving the work is still worth its engineering cost.

Fetch analytics, fonts and libraries, review them, version them, and serve them from origins you operate, so your users load only from addresses you control. That generalizes the self-hosting decision from the first ninety days to every third-party resource you can put behind your own origin. A change upstream then has to pass an inspection point on its way to the

Glass instead of arriving directly. The limit is the one self-hosting carries: proxying governs the delivery path, not the data endpoints your proxied code still calls at run time.

For resources that have to stay third-party, use sandboxed iframes, workers and permissions policies to limit what each one can reach inside the page. Browsers enforce those boundaries today. Most sites simply never set them. What they push back against is that once you let code in, it runs with your application's full rights, and nothing in the policy that admitted it has anything further to say about privileged-API access, DOM manipulation, storage access, or what it loads next. Containment does not repeal that. It narrows the rights admitted code inherits, and narrowing is what is available.

Serve live prices, control forms and transaction confirmations from a separate, locked-down origin inside a cross-origin iframe. An iframe boundary works in both directions, and this puts your own content on the protected side of it. The same-origin policy then stops every script in the host page, including the dozens of third-party ones, from reading or rewriting what is inside that frame. Payment providers already use this pattern to keep card fields out of reach of the merchant page, and it transfers directly to any content whose integrity matters more than the page around it.

Two conditions keep it real. The isolated origin has to stay clean, carrying no third-party code, or you gave the protection away at the door. And the boundary protects the inside of the frame, not the page around it: a compromised host page can still hide, overlay, resize or replace the whole frame with a lookalike. Pair it with frame-ancestors restrictions, and treat host-page integrity as the separate, still-open problem it is.

Give each critical application a named maximum number of third-party resources allowed to run in it, put an accountable engineer's name on that number, and review it whenever anyone proposes an addition. A budget turns reduction from a one-time cleanup into a standing constraint.

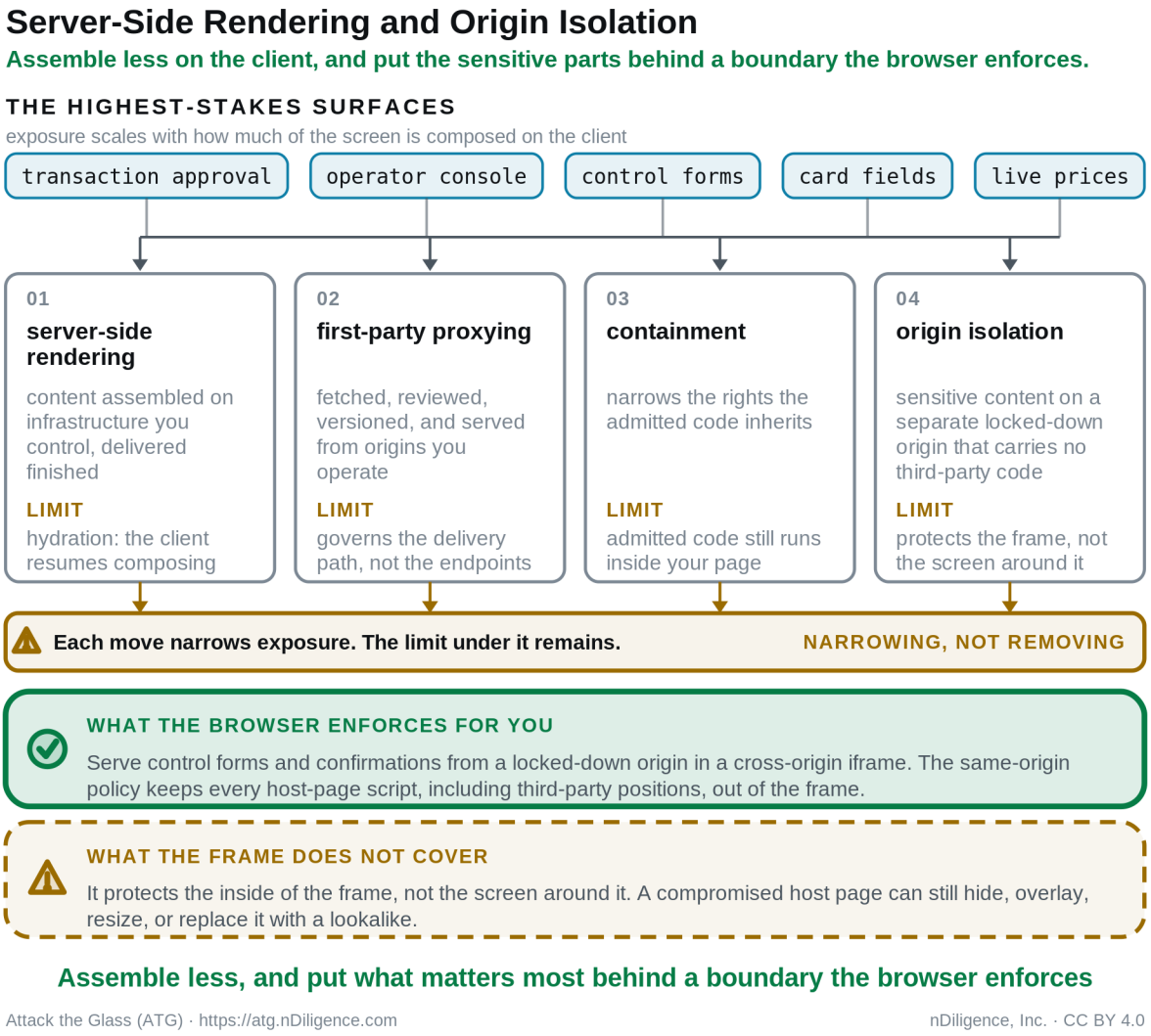

Figure 5: Server-Side Rendering and Origin Isolation

Your parts are secure and the plant that assembles them is not. Building less on the client, and putting prices, control forms and confirmations behind boundaries the browser enforces, shrinks the plant. That is this horizon's answer to an application that ships as instructions and gets built on somebody else's device.

Dependency Pipeline Controls

Most of what runs in a client runtime is open-source dependency code, and contributing to or taking over a package is the quietest route onto the Glass. The pipeline that pulls that code into your builds is a control surface in its own right.

Build from an internal package source holding reviewed, pinned versions rather than pulling from the public registry at build time. A private registry or curated mirror stops the

public ecosystem's churn at its own front door, and adopting anything new becomes a decision instead of a side effect.

Verify provenance attestations at that front door, and treat any critical package you cannot verify as a finding. Major registries now ship them: a signed record of how a package was built and by whom, confirming that a package came from the repository it claims, built by the workflow it claims. The limit is worth stating: provenance confirms where a package came from, not what it does. A maintainer who turns adversary ships perfectly attested compromise. Provenance closes off impersonation, not takeover.

Let new dependencies into the mirror through a human decision informed by maintainer history, ownership, release cadence and install-time behavior. That review is where somebody looks for takeover directly, and the reason it matters is what happens afterward: once packages compile into a bundle, the bundle is what executes but it carries no record of where its parts came from, so code from many sources arrives at the runtime under your own origin, indistinguishable from your own code and from each other. What gets into the bundle deserves scrutiny in proportion to how invisible it becomes once it is there.

Alert on maintainer changes, ownership and publisher transfers, yanked or replaced versions, and unusual release patterns across the whole tree as they happen. Commercial and open-source dependency-monitoring services do this today. A point-in-time review ages fast: XZ Utils passed every review that mattered until control moved to a new maintainer after everyone stopped looking, and somebody found it by accident rather than by process.

Compare each release's dependency tree against the last, and treat a new transitive dependency as what it is: a new party with code in your product, as visible and as reviewable as a code change.

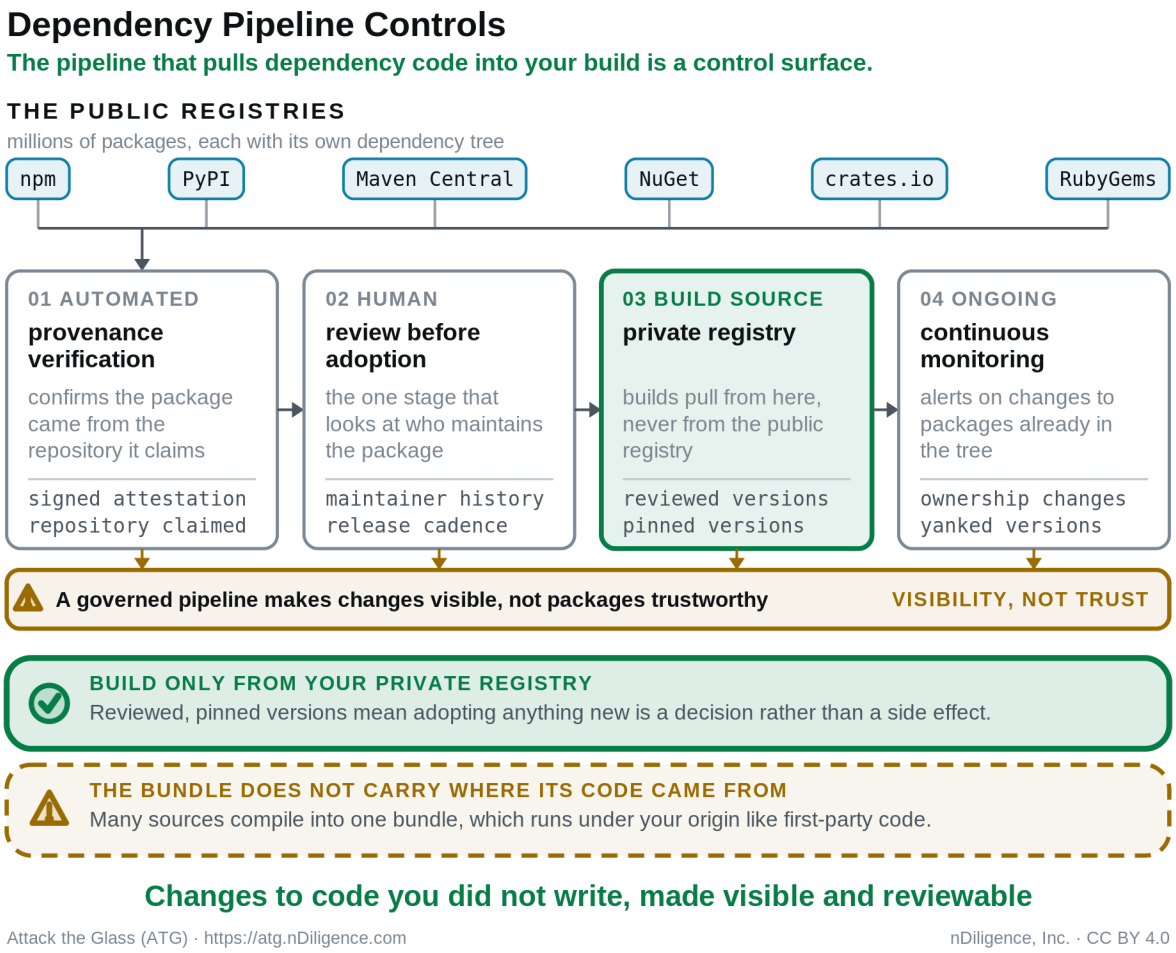

Figure 6: Dependency Pipeline Controls

A governed pipeline does not make your dependencies trustworthy. It makes every change to them visible, reviewable and attributable, which is the control available over code nobody on your payroll wrote.

The Glass Beyond the Browser

The Glass is any surface where something renders for a person, and the browser is only the most familiar. The same build-at-run-time pattern runs in mobile application runtimes and the webviews inside them, in desktop shells built on embedded browser engines, in smart-TV applications, in kiosks and digital signage, in point-of-sale terminals, in vehicle infotainment, in industrial control and SCADA operator screens, in AR and VR scene engines, and in the plugin hosts of productivity tools.

Each of those runtimes inherits the pattern from a shared design rather than catching it from the others: content composed at render time, executed on the Glass. Nothing spreads

between runtimes, and nothing gets eradicated by treating cases one at a time, because a patch to any single runtime does not reach the design that produces the condition.

Your defenses are uneven across those surfaces, and the plan has to account for that. CSP, SRI, Trusted Types, the Permissions API and the rest are browser-engine capabilities, and your application still has to switch each one on. Everything else that hosts a client runtime offers equivalents ranging from partial to absent, and they thin out on exactly the surfaces where the consequences are highest. Over-the-air bundle delivery, where code updates arrive straight from a server after install without going back through the app store, has no equivalent of a runtime integrity check. Hybrid runtimes have nothing like CSP for the bridge that lets web code call native device functions. Desktop shells and development-environment plugin hosts carry no consistent equivalent, and a loaded plugin runs with the host application's permissions. Where a host embeds a browser engine, that engine's primitives are available inside the embedded surface whether or not the host configures them, and some vendors say so in their own documentation. What no host outside the browser gives you is a default, and availability is not deployment.

So transfer what you already know how to do. Inventory the embedded runtimes you operate, recording what each one loads at run time and from where. Apply the browser-grade disciplines wherever the platform supports them, including inside embedded browser surfaces where the primitives exist but nobody ever switched them on: pinned content, constrained load sources, first-party serving. And for fixed-function displays, add the control a browser cannot have. A signage player or an operator console needs a short, known list of endpoints, enforced at the network layer, with an alert on anything outside it. A general-purpose browser cannot carry an egress allowlist. A device that renders one dashboard can, and should.

Weigh isolated networks accurately rather than exempting them. Isolation does not remove Attack the Glass. It relocates the supply chain, because every component crossed the boundary at some point and carried its exposure in with it. Well-run industrial control environments compensate with perimeter measures, URL pinning, mutual TLS, network segmentation and hardened runtime engines, all of which materially narrow the script environment compared with a commercial deployment. None of those reach the supply-chain source of the rendering-layer components, which is where Attack the Glass sits. The supply chain inside a protected network is nonetheless smaller and more controlled than the public internet, which is exactly why inventory and pinning pay off fastest there. Put that difference in the plan, and do not let it become a sense of immunity.

Your least browser-like screens are usually your most consequential and your least defended. Transferring the browser's discipline to them is unglamorous work with a high

return, and it is bounded by the same fact that motivates it: the patterns translate across runtimes, and the code that implements them does not.

Supplier Diligence: Ownership, Operator, and Jurisdiction

Every third-party resource that survives reduction belongs to somebody, and vendor risk management is where you find out who. Free components are not outside this. Many technologies offered free exist to establish *Placement and Access*, a standing right to execute inside your runtime, and an analytics snippet or an open-source component grants that whether or not money changed hands. Govern the free ones exactly like the paid ones, and remember that what a free component established can change owners along with the component.

A questionnaire is not diligence. For every provider whose code or content reaches the Glass, establish six things. Where the technology originates: who wrote it, where it is developed, who maintains it now. The full operational footprint: the data centers, networks and staff locations actually serving you, not the headquarters on the letterhead. The data flows: what leaves your runtime through their code, where it lands, who can read it. The sub-processors: the enumerated parties behind the name on your contract, because the ones that matter are often two or three contracts removed from the signature. Ownership and control: who operates the infrastructure, not only who brands or finances it. And the insider surface: who inside that provider holds administrative control of what reaches you.

Scope the exercise honestly, because the public record it draws on is thinner here than in any comparable industry. For physical assets a land registry says who holds title, securities filings say who controls a listed company, and threshold reporting leaves a trail when large sums move. Nothing equivalent covers the infrastructure that delivers code to a screen. A company registry names an owner. Nothing names the operator, and nothing files a report when a CDN is sold, a package is handed to a new maintainer, or a publisher account is transferred. So diligence here is mostly the provider's own answers, tested for consistency and re-tested on a cycle, and your assessment should say so rather than implying a verification it cannot perform.

An investor with a minority stake in a delivery provider has influence over a company. Whoever operates the network, runs the servers and holds administrative control of the paths the bytes travel has the delivery path itself, and can shape what renders without asking anyone. Assessments confuse the two more often than they confuse anything else. Legal ownership and operational control are routinely not the same people: whoever holds commit access, publishing credentials, authority over runtime configuration or platform-administrator rights can be replaced without anyone outside being told. When both appear in one supply chain, the operator is the sharper risk, and your assessment should say so.

Assess the insider surface just as concretely. Publish rights to the package, signing keys, deploy access to the serving infrastructure. Somebody can gain that access by recruiting or pressuring one person on the inside, and a provider who cannot tell you how many people hold those keys has answered the question in the way that matters.

Map jurisdiction and governing law before you need them. Where the vendor is incorporated, where its operations and infrastructure actually sit, and which law governs the contract decide whether you can enforce the notice, audit and remedy clauses you negotiated at all. If the map says you cannot enforce a clause, do not count it as a control. Jurisdiction also decides which governments can lawfully compel the provider, which bears directly on who could one day be serving your users.

Then write what you learned into the contract. Notification of change of control. Notification of material changes to served content and to the sub-processor list. Audit rights. These terms are ordinary to negotiate and meaningful to hold, provided the jurisdictional map says they can be enforced. They are also the only instrument that turns an unannounced change of owner into a notified one for the providers you contract with, which makes that the highest-value clause in the set.

Finally, score suppliers on what they could do rather than on what kind of company they are. Scope of effect, persistence, pre-operational intelligence required, attribution difficulty, reversibility, resource investment, time horizon, detection difficulty. The useful question about a supplier is what whoever controls it could do, how widely, for how long, and how you would find out. That follows from the access they hold, not from the size or the flag of the company holding it.

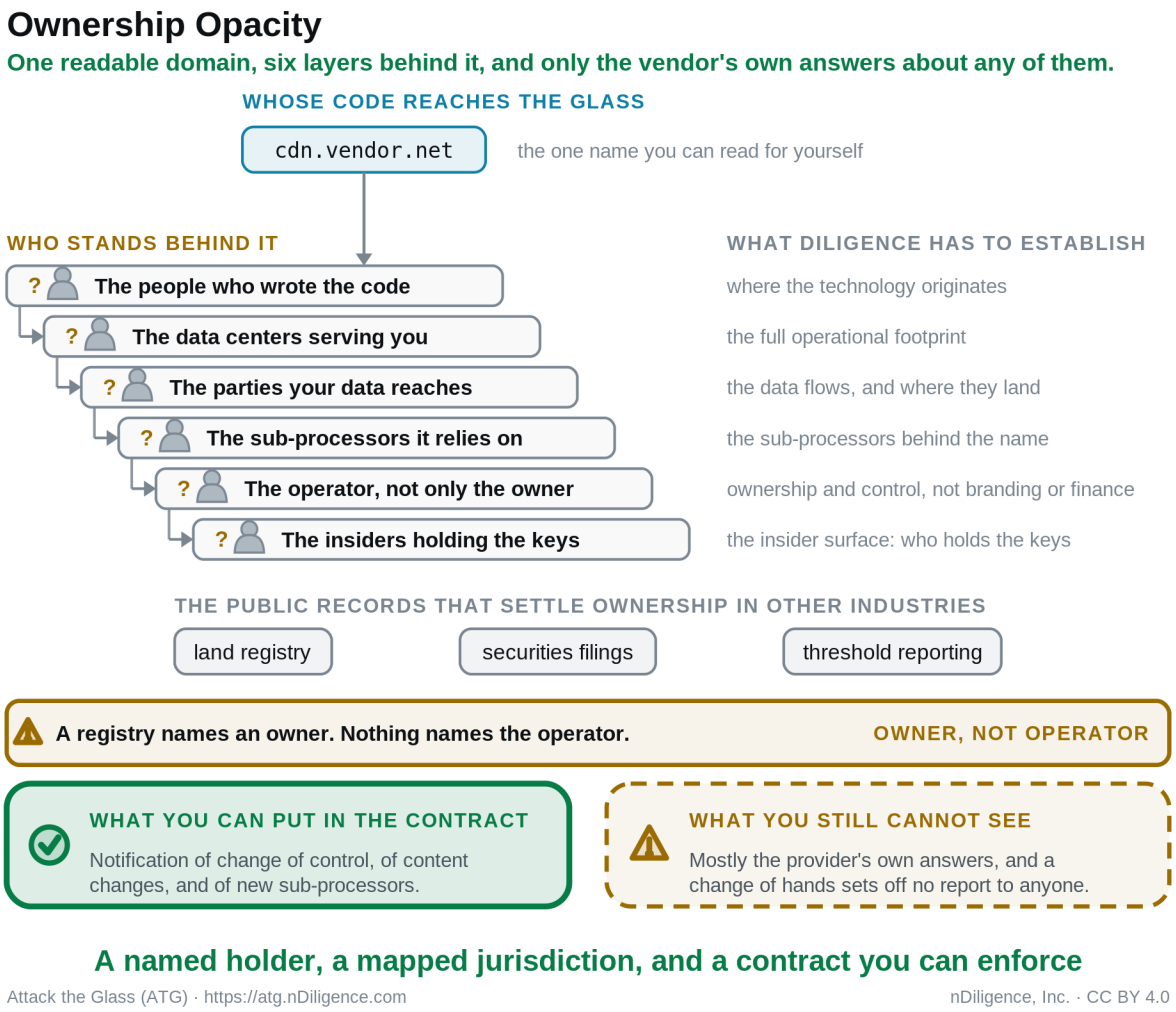

Figure 7: Ownership Opacity

Diligence does not remove a supplier from your interface. It replaces an unknown owner with a named one, a mapped jurisdiction and an enforceable contract, which is the difference between trusting a supplier and merely depending on one.

Client-Side Monitoring and Rendering Integrity Checks

Detect systematically what can be detected.

Deploy script monitoring on the interfaces that warrant it, and state its coverage plainly: it watches the pages and the vantage points you point it at, with the permissions you granted it. The commercial script-monitoring and client-side protection category that grew up around PCI DSS 4.0 inventories scripts, alerts on change, and detects classes of tampering on designated pages. It covers pages an observer can watch, and that is the whole of its reach.

Then turn the ad hoc vantage-point comparison of the first ninety days into a scheduled program. Fetch and fully render your high-stakes interfaces from multiple geographies, networks, device profiles and browser engines on a fixed cadence. Capture the served resources and the rendered DOM. Compare against a known-good baseline and against each other, and alert on drift rather than reviewing by hand. Synthetic-monitoring and headless-browser tooling supports every step today. What this catches that single-point monitoring cannot is divergence: one address returning different things to different people.

Conditional delivery decides what a resource returns based on who is asking, using the same conditions any content-personalization system already uses: source address and inferred geography, device and user-agent characteristics, time of day and session timing, who is signed in, what they did before, which group they were put in, and which experiment bucket they landed in. You detect a swap exactly to the extent that your vantage points differ along the condition somebody used. Geography and device profile are cheap to vary, so region-scoped and device-scoped substitution is genuinely within reach. Who is signed in, what they did before and which group they belong to are not properties a synthetic observer has, so a swap conditioned on those reaches those users and nobody you operate, whether that group is a hundred thousand people or one named individual on one device. Build your vantage points to vary along the conditions that can be varied, say plainly which ones they cannot express, and treat that list as a stated blind spot rather than a footnote.

Absorb the substitution scenario into incident response, and absorb the missing evidence with it. A substituted resource lives in memory for one session and is gone when the tab or the application closes. A network monitor records metadata, timing, size and destination, never the content. Whoever serves a resource also decides whether the client keeps it, so it can be served once, marked **`no-store`**, and leave the cache an investigator would search entirely clean. Write a runbook for the day somebody finds a served resource or a rendered page altered, and start it with capture: how client-side evidence gets preserved before it vanishes, who cuts off the supplier, who notifies affected users, who calls the provider. Then rehearse it. A tabletop built on a false operational reading, an altered payment flow or a targeted executive deception, run with the people who would actually take the call, is the difference between a procedure on paper and one that survives contact. The out-of-band verification from the first ninety days is only real if it holds up in that rehearsal.

Track four numbers so the year can be reviewed. How many third-party resources each interface loads, trending down, with a name attached to every increase. What share of high-stakes interfaces serve pinned or first-party resources. What share run an enforced CSP and monitoring. What share have verification procedures in place. Where those

numbers show gaps, state the gaps plainly, because they are what the long-term roadmap picks up.

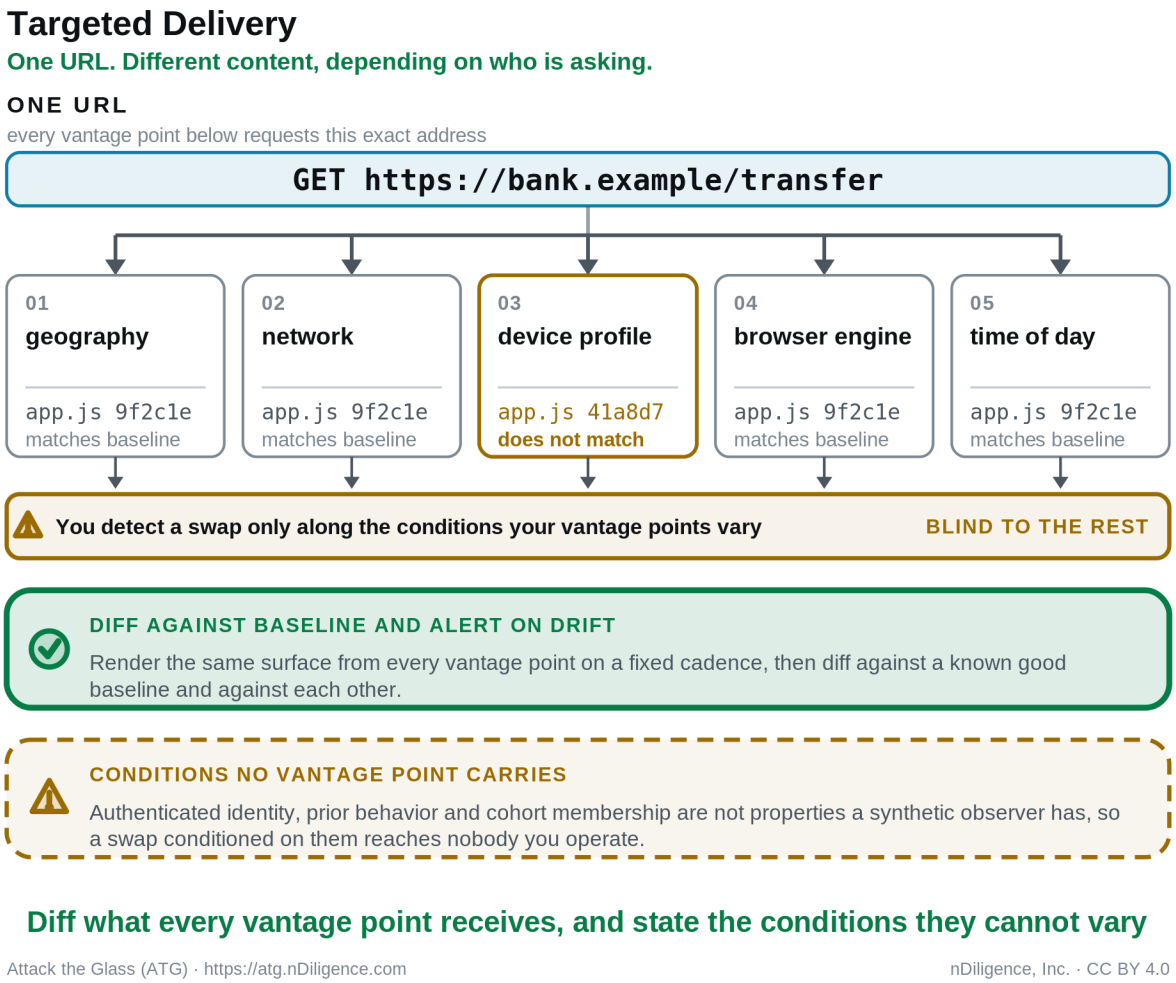

Figure 8: Targeted Delivery

You can detect a swap served to a whole region, network or device class, with tooling that exists today, and it is worth running. It limits how widely a substitution can spread before somebody notices, along the conditions you can vary. It will not catch a swap aimed at who the recipient is, and nothing deployed today will.

What Carries Forward

By the end of the year less of the interface is built on the client, your prices and confirmations sit behind boundaries the browser enforces, every change to the dependency pipeline is visible and attributable, browser-grade discipline reaches the surfaces that are not browsers, your remaining suppliers are named and contractually

bound and mapped to a jurisdiction, and detection runs across every condition you can actually vary.

Every one of these has an end date. Dependency counts drift upward once no budget holds them down, diligence findings age as providers change owners, and procedures decay without rehearsal. Keeping what you gained is what the long term takes up.

Long-Term: 12-Month+

The first two horizons are projects, and projects end. The pressures that produced your original exposure do not. The market keeps shipping architectures that build more of the interface on the client rather than less, because the economics still pay: a provider ships instructions instead of a finished product, the user's device pays to build it, and that cost splits across billions of devices with no line item anywhere. The same economic fact saves the provider money and leaves the runtime open, which is why nobody is giving the design up. What a longer horizon buys you is every mid-term control converted into a standing function with an owner, a metric and a renewal date, plus something no shorter horizon can buy: leverage over what gets built next, in your own architecture and in the industry's.

Past twelve months, enforce architecture standards at design review, monitor supplier ownership as a standing function, use procurement as market pressure, join the collective work on the problems bigger than any one organization, and write the risk itself into how your organization is run.

Architecture Standards and Design Review

Months three to twelve rearchitected the high-stakes surfaces you already had. Design review governs the ones that do not exist yet.

Make new systems justify every third-party resource at design review, the way they already justify open ports and data flows. The dependency budget stops being a cleanup measure and becomes an architecture standard: a named ceiling for each surface, owned by someone, applied at review, so a new dependency has to be approved rather than simply added. At every proposal, ask what popularity makes it easy to skip: what did anyone establish about this component besides the fact that a lot of people use it? The cheapest dependency to defend is still the one you never took, and design review is the only point in a system's life where refusing costs nothing.

Default high-stakes surfaces to server-side assembly in new builds. Building on the client has permanent economics and nobody here pretends otherwise. The claim is narrower: how much any one surface leans on the client is a choice, and for consequential screens the conservative end should be the default that somebody has to argue against rather than for.

Design consequential workflows with a second path to the truth. The first ninety days bolted out-of-band verification onto processes that already existed. Now build it in. Treat a workflow whose only source of truth is a single rendered screen as a design finding, recorded and remediated like any other. Specify out-of-band confirmation and dual-source

checks when the workflow is designed, test them when it changes, and let every system that implements the workflow inherit them.

And sunset what you said you would sunset. The counts and coverage metrics from the mid-term feed an annual reduction target with an owner's name on it. Schedule a rebuild for surfaces that cannot meet the standard. Waivers expire, and the register remembers.

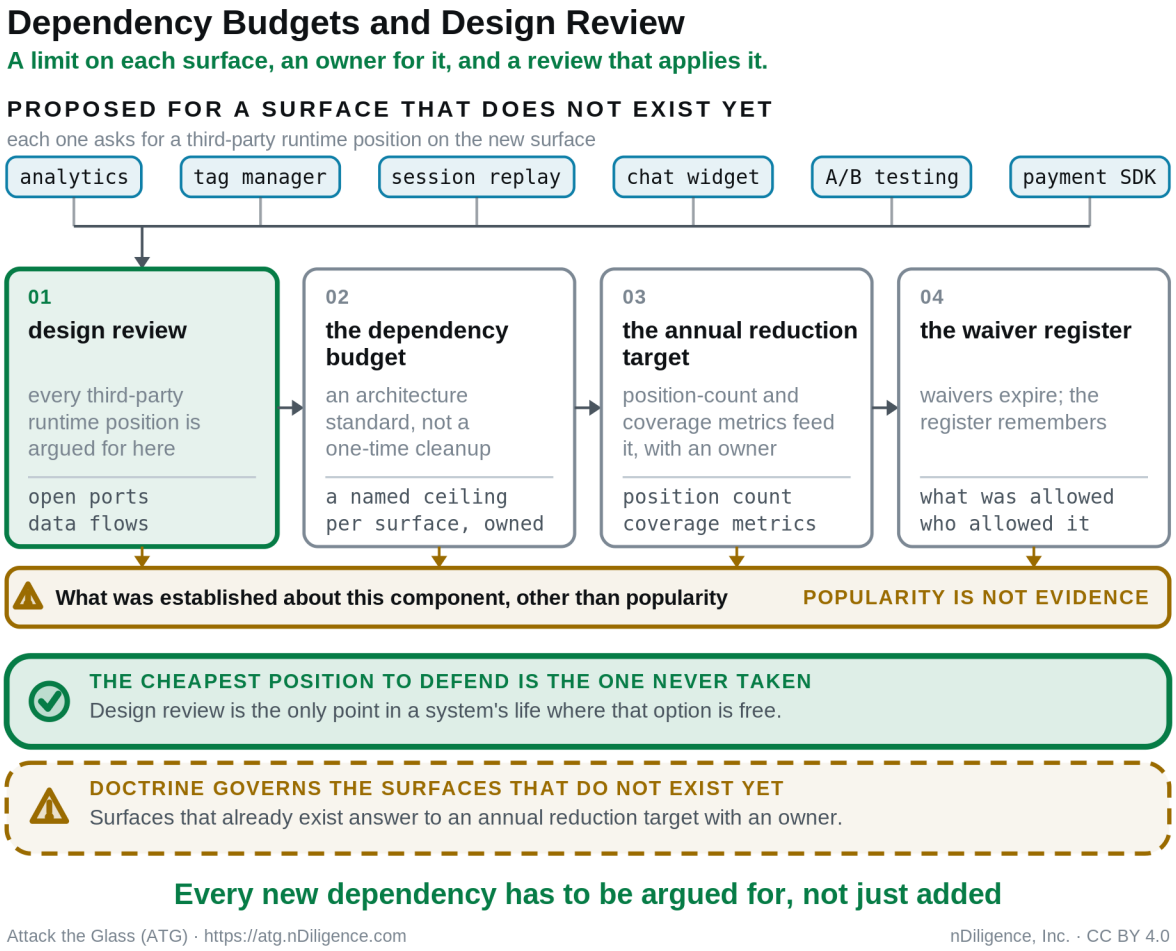

Figure 9: Dependency Budgets and Design Review

Design review is how a remediation outlives the team that performed it. Everything else in this horizon assumes it.

Continuous Supplier Ownership Monitoring

A hostile supplier sits undisturbed because nobody outside can find out who controls it, and what is missing here is the record rather than the technology. A land registry says who holds title to a building. Securities filings say who controls a listed company. Threshold reporting leaves a trail when large sums move. Nothing equivalent covers the CDNs,

package namespaces and publisher accounts that deliver code to a screen: a company registry will name an owner, but nothing names the operator, and nothing files a report when a CDN is sold, a package is handed to a new maintainer, or a publisher account is transferred. The most important supply chain ever built has no way of telling anyone when the people controlling it change. What is available today is not a technology. It is sustained, unglamorous diligence, run as a standing function rather than as a procurement gate.

Monitor who controls your suppliers continuously: ownership, operating control and jurisdiction for the providers, packages and infrastructure your runtimes depend on, refreshed on a cycle and on event. Corporate-registry research, registration and certificate monitoring, and commercial due-diligence services all exist for this. What the long term commits to is running them continuously and joining the results to your inventory, so every entry in your dependency list carries a current answer to who controls it.

Change of control is the alert that matters most, and you have to hunt for it rather than wait, because control moves through channels that leave no technical trace: a company sale, a stolen credential, a maintainer or namespace transfer, a DNS repoint. None of those alter the address, the hash or the certificate. Everything a scanner measures stays green while the party deciding what that asset serves becomes someone else. Polyfill.io and XZ Utils were both changes of control before they were incidents, and both were visible beforehand, one publicly and one in the project's own records. Then comes the interval that defeats detection: whoever acquired the asset operates it normally, serving exactly what its dependents expect, for as long as they choose, and often the first detectable event is the attack. Monitor so that the next change of control reaches you as a warning while it is still only a warning, because the quiet stretch afterward will not produce one.

Keep the operator separate from the financier as a standing discipline, treat the operator as the sharper risk wherever both appear, and never take legal ownership as a proxy for who holds the credentials. Then watch for asset cycling. The pattern runs in four stages: somebody acquires an asset that arrives with its trust and its installed dependents already attached, runs an operation through it, abandons it before sustained attribution can resolve, and re-emerges on new infrastructure related to the abandoned operation but not formally linked to it. Each stage exploits a specific gap in the record, and the last one sends an investigator back to the start. What that means operationally is that a takedown is not an ending. The same operator returns under a different name, sometimes reusing code patterns, naming conventions or recognizable behavior, and often reacquiring an overlapping population of dependents. Track operators and not only events, or you will keep finding the same operator behind new names.

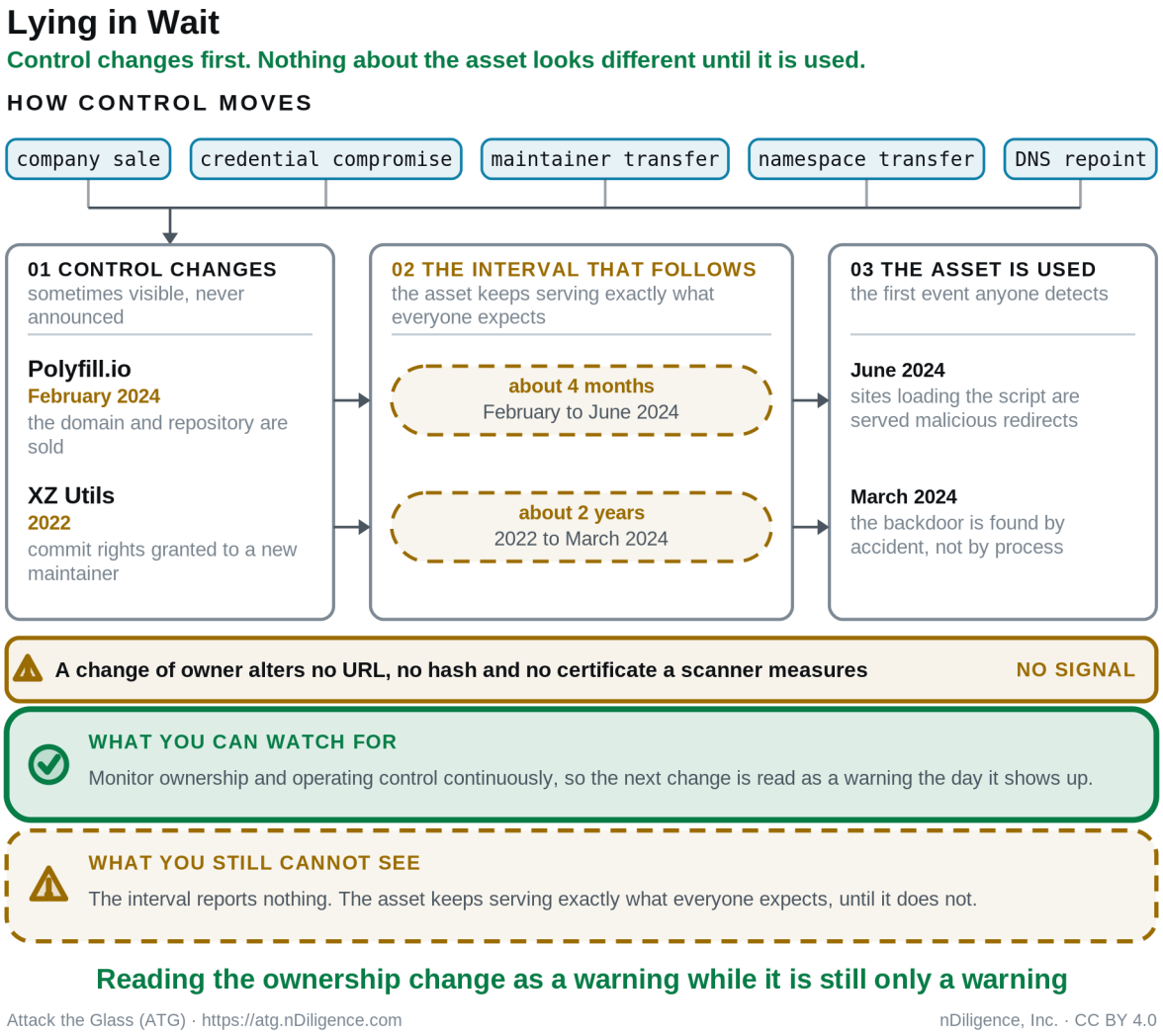

Figure 10: Lying in Wait

Monitoring does not close the industry's ownership gap, and no single organization can. It closes your instance of it, one supplier at a time, and turns an unbounded unknown into a watch list somebody manages.

Procurement Terms and Flow-Down

Write the runtime supply chain into the terms of every purchase whose code renders for your people. One complaint changes nothing about a market; buying criteria applied for years do. Write five terms into every one of those contracts, all of them following directly from the mid-term diligence work: disclosure of ownership and operating control, notification of change of control, enumerated sub-processors, provenance attestation for delivered software, and audit rights. These are ordinary clauses to negotiate, and their power is cumulative. A supplier who meets them for one customer can meet them for the

next, and a market segment where buyers routinely demand them is a segment where they become the default cost of doing business. Change-of-control notification carries the most weight of the five, because it is the only one that turns an unannounced change of owner into a notified one for any provider you contract with.

A vendor who cannot tell you who operates their delivery infrastructure has answered a different question, and that answer belongs in your scoring. Weighting supplier selection toward the ones who can answer moves a market in a way no security review ever will.

And make the terms flow down. Primes pass them to subcontractors. Platform customers ask platforms what they load at run time and from whom. The suppliers that matter are often two or three contracts removed from the one you signed, and a term that stops at the company you signed with governs almost nothing.

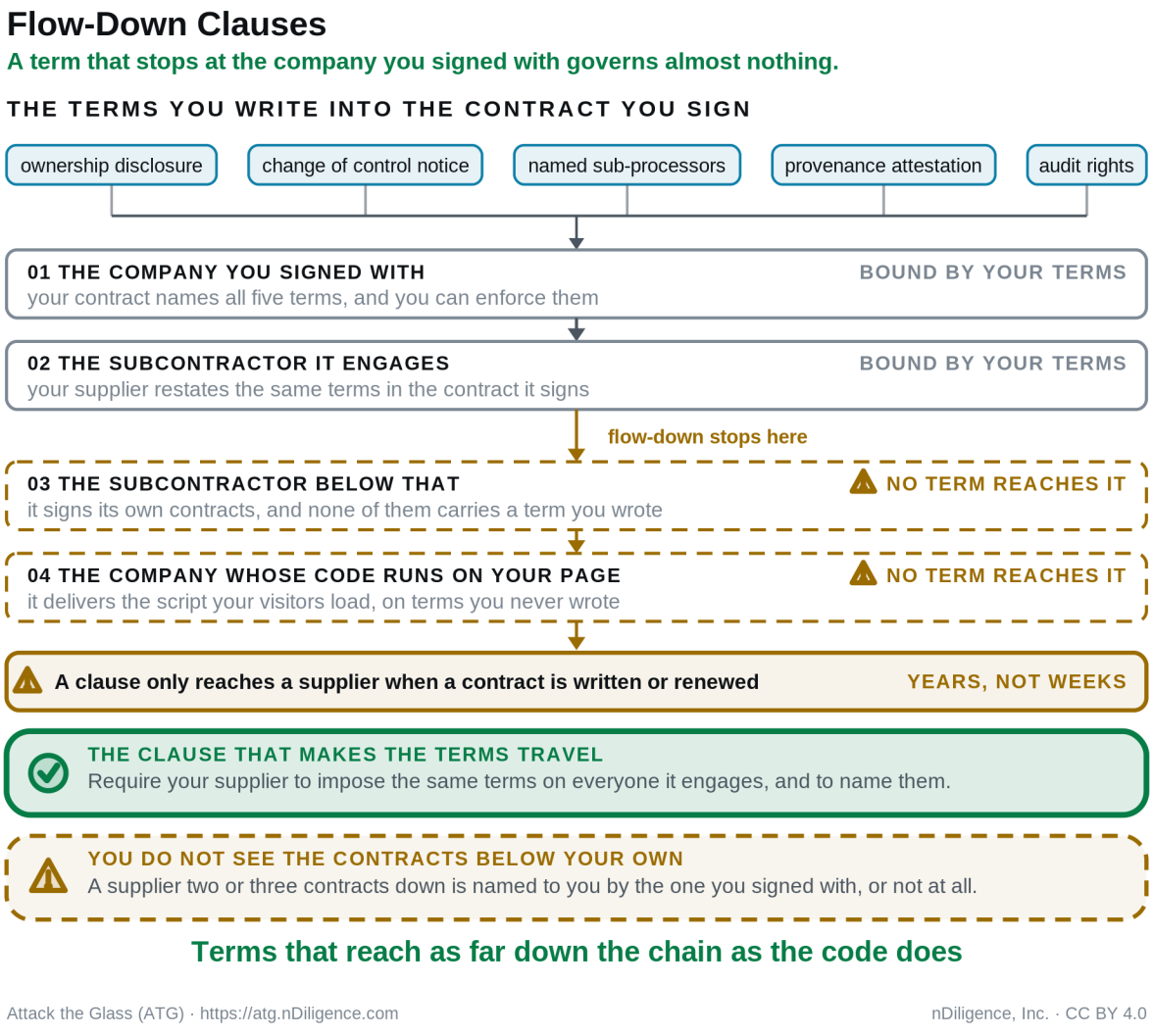

Figure 11: Flow-Down Clauses

Procurement is how your requirements reach organizations that will never read a security whitepaper. It is slow, it compounds, and for the supply chain that reaches the Glass it is badly underused.

Standards, Incident Sharing, and the Capability Gap

Four problems outrun any single defender: what the runtime platform specifies next, how little anyone knows about actual scale, who owns the delivery infrastructure, and a defense nobody sells yet. Each of them has something you can act on today.

Take seats in the standards bodies that specify the runtime's future integrity mechanisms. Web platform, package ecosystem and security standards groups extend the load-time primitives, and whoever shows up to define rendering-layer verification will define it. Participation is not a gesture. It is how an operator's requirements become a platform default a decade later. Carry the same three requirements into every one of those rooms: one authoritative source of truth for what a runtime actually loaded, detection of content swapped after load, and protection against tampering with an application's state in memory while it runs.

Share every well-documented substitution incident through your ISAC or the equivalent sector body, both of which exist now. Nobody knows the true scale of ATG-class operations, and the reason is structural rather than anecdotal: nobody has deployed, at population scale, the instrumentation that would show a swap at the rendering surface, so finding nothing tells you nothing about whether swaps are happening. Absence of evidence is not evidence of absence, and a blind spot is exactly where not seeing something differs from its not being there. That cuts against a defender's optimism and an analyst's alarm equally. Sharing one incident turns a private loss into collective visibility and sharpens the field's picture of actual use.

No registry, no filing and no report exists for an outsider to check an ownership claim against, and that gap is what your monitoring program spends money working around. Support industry disclosure norms, research into infrastructure ownership, and transparency initiatives, all of which push on the gap itself rather than on the workaround. Observe it and engage, and stop there. Substantive policy design sits outside this.

Nobody sells a capability today that watches how content behaves at the rendering layer, on every device class, and no plan here pretends otherwise. It is also not a mystery. Partial versions exist in narrow forms: in-browser script-behavior telemetry from client-side attack surface management vendors, runtime sandboxing layers that observe a subset of API calls, and policy layers that check a model's output against fixed rules before an application uses it. None of them is a runtime default with rendering-layer breadth. Part of

the obstacle is economic, because inspecting what running code has changed as fast as the runtime changes it costs CPU, memory and energy, against the same incentives that put the rendering cost on the user's device to begin with. A per-vendor answer will not do, because every runtime inherits this from a shared design, and a patch in one runtime does not reach that design. What you can do today is create demand: state the requirement in R&D agendas, in procurement signals and in standards contributions, so that what eventually ships is shaped by the operators who need it. Funding research toward it, where you have the means, is the strongest form of the same signal. You can state the requirement today. You cannot buy the capability, and never let one stand in for the other.

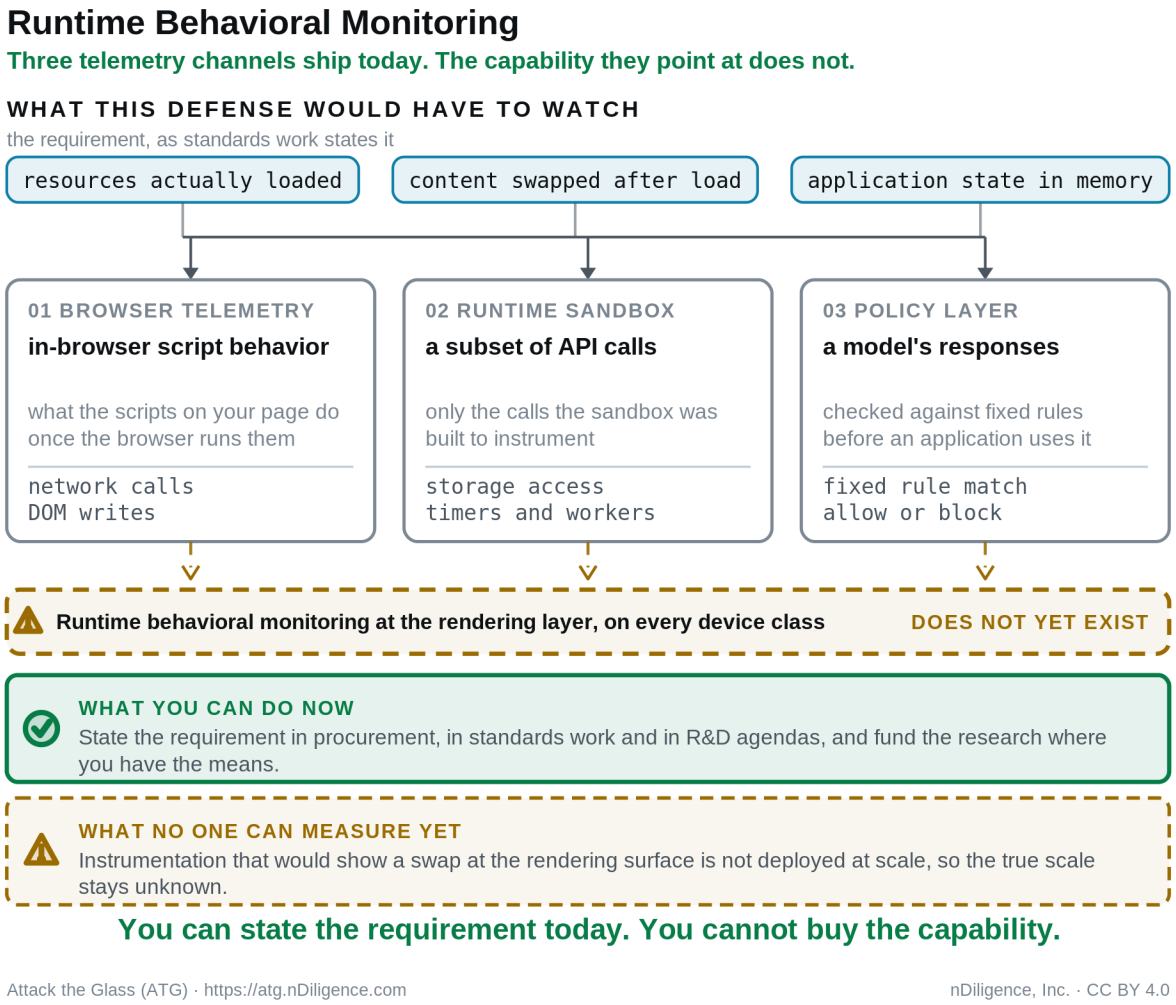

Figure 12: Runtime Behavioral Monitoring

None of those four protects you this quarter. All four decide what protecting you costs in five years, and they are the only actions here whose benefit reaches defenders who never took them.

Risk Registers, Metrics, and Training

Put the Glass in your enterprise risk register as a named risk with an owner, a stated appetite, its current controls and its honest residual: rendered content is not behaviorally verified on any surface this organization operates, and no standard capability currently closes that gap. Written that way, the residual is not an admission of failure. It is the sentence that keeps the risk funded.

Show leadership the numbers on a cycle: third-party resource count per interface, coverage of pinning and monitoring, how current your diligence is across the supplier base, and compliance with the verification procedures, reviewed like any other control family. Those numbers already exist from the mid-term. Making them permanent means they stop being a project's report and become a function's obligation.

Keep training permanent and role-specific. Operators, finance staff, administrators and executives each get shown a different kind of false screen, and each group's curriculum, tabletop cycle and out-of-band verification drills continue for as long as the architecture that motivates them does. One point belongs in every version, because it is the one that ages worst if left out: an attacker chooses who receives a change and can serve different content to several groups at once, so what a colleague saw on their screen is no evidence about what anyone else saw. Rehearse the procedures, or they are documents rather than controls.

And retain the expertise deliberately. Supply chain intelligence, client-side architecture and dependency governance are skill sets, not projects. This survives staff turnover only if those roles are permanent and have named successors, rather than initiatives attached to individuals.

The register, the metrics, the calendar and the roles are what separate an organization that remediated Attack the Glass once from one that stays remediated.

What Carries Forward

Past twelve months you have an architecture that builds less on the client by standard, a supply chain whose owners are known, watched and contractually bound, workflows that never rest on a single rendered screen, a market pushed deliberately toward transparency, and an institution that carries all of it beyond the tenure of the people who built it.

The Residual

Every change in these three horizons stops at the same line: none of them watches how content behaves on a user's own device at the moment the screen is drawn.

Where you check a resource, you check it as it was delivered rather than as it behaves, because every check you can deploy runs before the code does and the code runs afterward. Where you observe behavior, you observe it on a copy served to an observer, and any condition that observer does not carry is a condition it cannot see. And afterward there is usually nothing left to examine, because the substituted resource lived in memory for one session and whoever served it decides whether the client keeps a copy at all.

Runtime behavioral monitoring at the rendering layer is the control that would cross that line. It exists today only in narrow forms: in-browser script-behavior telemetry, sandboxing layers that watch a subset of API calls, output checks against fixed rules. As a default capability of a client runtime it exists nowhere, on any device class. Building it is an engineering problem rather than a mystery, and part of what holds it back is economic, because inspecting what running code has changed, as fast as the runtime changes it, costs the processing, memory and energy this whole design exists to push onto somebody else's device.

So write the gap down and keep it where the people who fund the work can see it. In a risk register it reads as one sentence: rendered content is not behaviorally verified on any surface this organization operates, and no standard capability currently closes that gap. Written that way it is not an admission of failure. It is what keeps the work funded until somebody ships the capability that retires it.

Everything in these three horizons is what an organization does about a vulnerability nobody can patch. Fewer parties reach the Glass, each one is named and watched, a second path to the truth stands behind every screen that moves money or equipment, and the requirement for what nobody sells yet is stated out loud, in procurement, in standards rooms and in research agendas. That is not the whole answer. It is the part available now, and an organization that has done it is the one still standing when the rest arrives.