Deliberative Loading of a Global Polyfill: Compromise Simulation and OSINT Analysis

Adelaida STĂNCIULESCU[1], Ioan C. BACIVAROV[2]
[1] Bucharest Court, Bucharest, Romania
adelaida.stanciulescu@gmail.com
[2] Faculty of Electronics, Telecommunications and Information Technology,
National University of Science and Technology POLITEHNICA Bucharest, Romania
ioan.bacivarov@upb.ro

Abstract

Modern web projects frequently rely on third-party packages and services (CDN, polyfills providers) to ensure compatibility. A polyfill that modifies global objects (e.g. Array.prototype) provides convenient compatibility, but introduces a single point of failure: compromising that provider can lead to the distribution of malicious code to all pages that include it. The purpose of the study is to demonstrate, in a controlled manner, the effects of installing a global polyfill and to show how exposures can be identified and quantified through ethical OSINT techniques. This paper presents a reproducible methodology for simulating the scenario where a polyfill It installs its functionality globally (Array.prototype.findLast () as an example) and thereby expands the attack surface of web applications. Using a controlled environment and ethical OSINT techniques to map adoption and exposure in the public space, the paper assesses operational risks and proposes technical mitigation measures. The methodological emphasis is on reproducibility, non- intrusiveness and validation based on public evidence.

Index terms: CDN security, Client-side integrity, Polyfill compromise, Simulation testing, Supply-chain vulnerability

1. Introduction

This case study presents a controlled experimental scenario designed to illustrate the potential impact of compromising a polyfill provider distributed through a Content Delivery Network. Within an isolated virtual machine, a JavaScript file - conceptually treated as a "malicious" component - is created to globally redefine the Array.prototype.findLast method.

The altered implementation does not perform harmful actions such as data exfiltration or unauthorized code execution; instead, it purposefully modifies the method's semantics or introduces a detectable visual cue in the page interface, allowing the functional effects of the compromise to be observed without producing real damage.

The simulation is carried out by injecting the same modified polyfill into two separate static HTML pages. The first, *trusted.html*, models a legitimate application that consumes the polyfill and acts as the target of a supply-chain compromise. The second, *attacker.html*, demonstrates how an adversarial actor could exploit globally accessible APIs once they have been overwritten or polluted. Both pages are served locally and operate on dummy data: the trusted page performs method invocations and parity checks on findLast, comparing expected outputs with those produced under

the compromised environment, while the attacker page showcases how the globally introduced function can be invoked from an untrusted context.

Through this setup, the case study highlights the broader security implications associated with uncontrolled global polyfills and the risks that arise when core JavaScript functionalities are replaced or tampered with by external providers.

2. Case study

The paper proposes a controlled simulation that aims to highlight the effects of a potential compromise of a polyfill provider distributed via CDN [1]-[3], [9], [14], [16].

In an isolated environment (virtual machine) a file will be created Test JavaScript, conceptually called " malicious ", that will globally install the Array.prototype.findLast method. The intentionally modified implementation will not contain any malicious behavior (e.g. exfiltration or executions), but will only alter the semantics or introduce a visible indicator in the page interface - so that the consequences functional aspects of the compromise can be observed without causing real harm [14].

The experiment is conducted by loading the same compromised script into two distinct static pages: a legitimate page (trusted.html), which represents the vulnerable application or polyfill consumer, and an attacker page (attacker.html), which illustrates how a malicious third party could take advantage of the global availability of the function. Both pages will be served locally and will run exclusively dummy data: the first will execute calls and parity checks for findLast (comparing the expected result with the observed one), and the second will demonstrate access to the globally introduced API [1]-[3], [14], [16].

The procedural point of view , the experiment respects the principles of non- intrusiveness and risk limitation: all activities are carried out in controlled resources, no operations are performed on public infrastructures [11], [12].

Expected simulation results include:

1. confirmation that a compromised polyfill , installed globally, expands the attack surface by providing a common interface to all scripts on the page.
2. demonstrating how a malformed implementation can alter the logic of dependent applications (by changing the semantics of a standard call).
3. highlighting simple detection mechanisms on the client (checking the toString () function, observing the visual marker, analyzing files uploaded to Network / Sources).

For the testing activities, an isolated network machine was prepared, using the Chrome (version 96) and Firefox (version 143) browsers. File serving was carried out through Python 3 using the command *python -m http.server 8000*, and content editing was performed with the Notepad++ text editor.

File structure is the following: trusted.html (containing the captured fragment), attacker.html (the page demonstrating global access) and an external script cdn_findLast_malicious.js which, for didactic purposes, installs Array.prototype.findLast (or leaves a visual marker) - see Fig. 1.

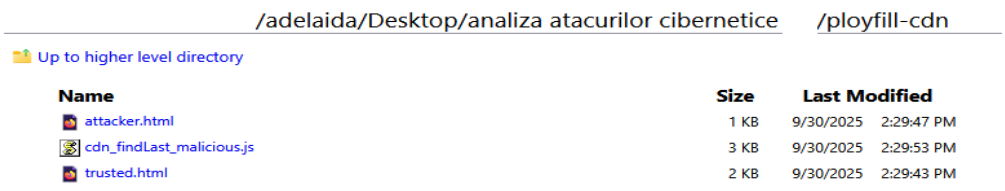

Fig. 1. The file structure

The code snippet illustrates a trusted web page used to test the behavior of the *findLast()* function from the *Array.prototype* object. When the button is pressed, the script executes a function that checks whether the native method exists and then applies a filter to an array of values. The result is displayed on the page, highlighting the difference between the expected native behavior and a potentially altered behavior caused by including an external, possibly malicious script. This demonstration emphasizes the risks associated with JavaScript prototype manipulation and the impact unauthorized code can have on web application execution (Fig. 2):

```
<!doctype html>
<html>
<head><meta charset="utf-8"><title>Trusted - file:// demo</title></head>
<body>
  <h1>Trusted page (file:// demo)</h1>
  <p id="ua"></p>
  <button id="btn">Testează findLast()</button>
  <pre id="out"></pre>

  <!-- incarcare "CDN"  -->
  <script src="cdn_findLast_malicious.js"></script>

  <script>
    document.getElementById('ua').textContent = navigator.userAgent;
    document.getElementById('btn').addEventListener('click', function(){
      var out = document.getElementById('out');
      out.textContent = '';
      try {
        out.textContent += "'findLast' in Array.prototype => " + ('findLast' in Array.prototype) + "\n";
        var arr = [1,2,3,4];
        var res = arr.findLast(function(x){ return x % 2 === 0; }); // expected native -> 4; malicious -> 2
        out.textContent += "Result: " + String(res) + "\n";
        console.log('Trusted result:', res);
      } catch(e) {
        out.textContent += 'Error: ' + e.message + "\n";
      }
    });
  </script>
</body>
</html>
```

Fig. 2. Structure of the file named trusted.html

Executing the above code on the browser Chrome v 96 is shown in the following screenshot (Fig. 3):

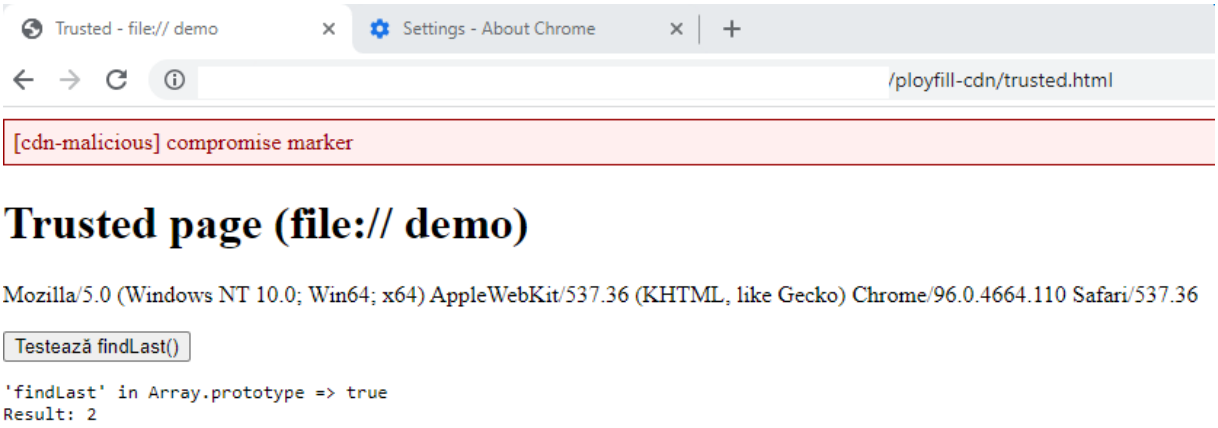

Fig. 3. Executing the code in Chrome v96

The following are identified in the image:

- Red banner [cdn-malicious] compromised marker - **visible indicator** that the external script (simulated as "CDN") has executed and inserted a marker into the DOM.
- navigator.userAgent SHOW **Chrome 96 -** version where Array.prototype.findLast it was **not** natively supported (native implementation appears in Chrome ≥97).
- The line ' findLast ' in Array.prototype => true - **the method exists globally on the** Array prototype, so it was installed by a script (polyfill) and not by the browser 's native engine.

- Result: 2 for arr.findLast (x => x % 2 === 0) - **incorrect behavior from the specification**: correctly (natively) the result would be 4 (last even element), but here the result is 2 (first even element). This indicates a "malformed" implementation that scans left→right instead of right→left.

As can be seen in capture performed with DEVtools, the marker in the "CDN" script executed successfully (Fig. 4).

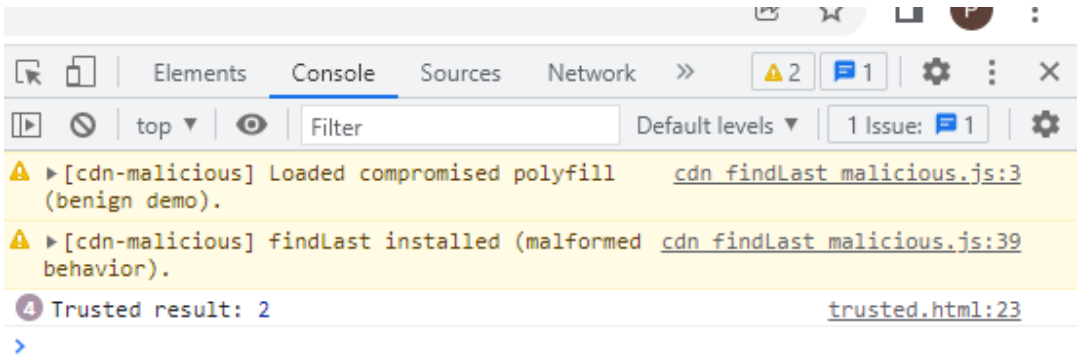

Fig. 4. The marker in the "CDN" script executed successfully

The console indicates:

- ⚠ [cdn-malicious] Loaded compromised polyfill (benign).
  cdn_findLast_malicious.js:3
  → **the marker** in the "CDN" script has been executed. Loaded message compromised polyfill is a console.warn in your test file - it indicates that the script ran. Line 3 of cdn_findLast_malicious.js is where the script did the console.warn.
- ⚠ [cdn-malicious] findLast installed (malformed behavior).
  cdn_findLast_malicious.js:39
  → the script installed a **malformed version** of Array.prototype.findLast (the message is also a console.warn). The indicated line shows the definition/installation point.
- Trusted result : 2 trusted.html:23
  → the result of the call from trusted.html (parity test) is **2**. This confirms the malformed behavior: instead of the last even element (4), the function returned the first even element (2).

This result clearly indicates that a polyfill provided by a "CDN" (in the exposed scenario: cdn_findLast_malicious.js) was executed and globally modified the JavaScript behavior (Array.prototype.findLast). The modification changed the semantics of the function, causing logical errors in applications that expect the standard behavior. In terms of security: the attack surface of the application has expanded - any script on the page (including a malicious one) can use or abuse the global method.

This analysis is relevant for:

- supply chain risk: if the CDN provider is compromised, the malicious code is automatically distributed to all sites that include that script.
- Behavioral breakage: changing semantics produces bugs, regressions, and possible cascading effects in dependent applications.
- Abuse capabilities: an attacker can immediately take advantage of the global API in compromised pages or even third-party pages that include the same script [1]-[3], [16].

As can be seen from the console analysis, the external script cdn_findLast_malicious.js ran, globally modified Array.prototype.findLast and introduced behavior incompatible with the specification - a demonstration scenario of benign compromise.

CALLING file attacker.html show What **can make a page harmless** (but controlled) then when a dependency (polyfill) is installed globally (Fig. 5).

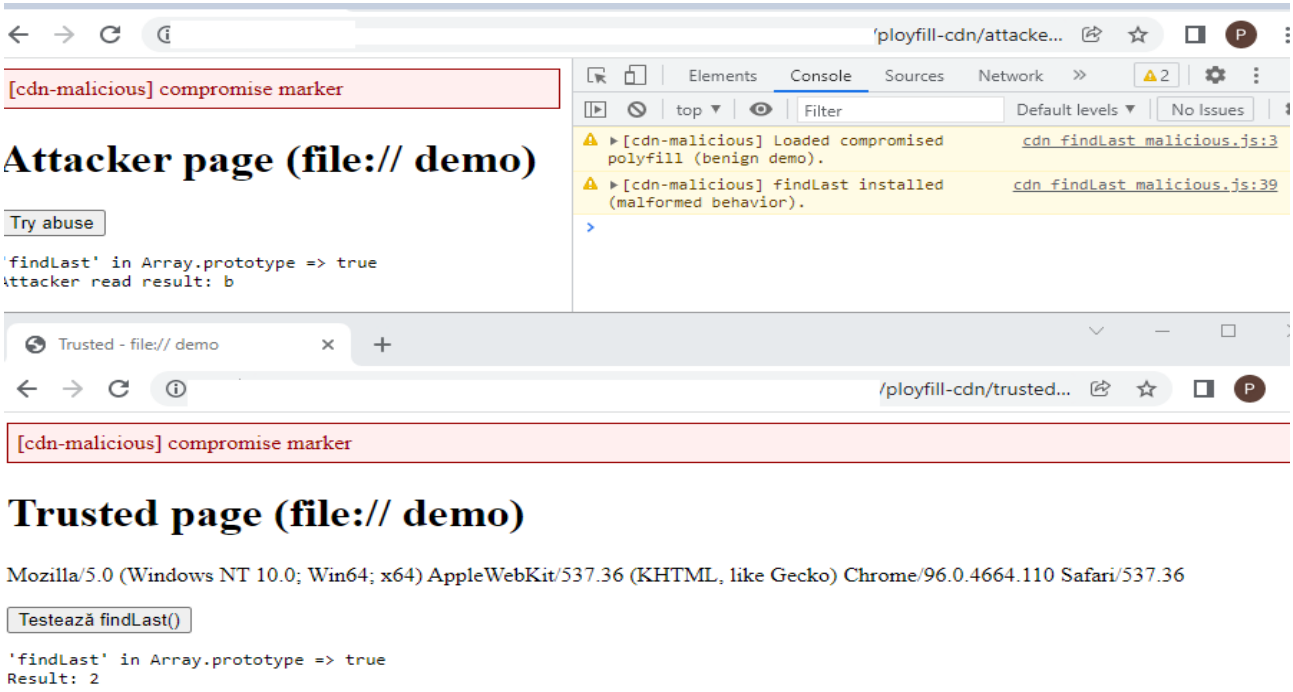

Fig. 5. CALLING file attacker.html

3. Conclusions

The study demonstrated, in a controlled environment, that including a polyfill distributed through a CDN at runtime can have significant effects on the behavior of the application. and on the attack surface of the web ecosystem.

Reproducibly showed how a compromised script, even if demonstrably benign, can:

a. install a new method globally (Array.prototype.findLast),
b. semantically alter the function (behavior incompatible with the specification), and
c. makes that API available to any code executed in the page, including untrusted pages (e.g.: attacker.html)

Key findings

1. Easily accessible global modification: a polyfill that modifies global prototypes provides immediate access to the same functionality for all scripts on the page.
2. Possibility of behaviour-breakage: non-compliant or compromised implementations can change the logic of applications (demonstrated by example 4) expect vs. 2 observed), introducing subtle bugs or regressions.
3. **Client-side detectability:** indicators of compromise are detectable by simple methods (checking the toString() function, DOM marker, analyzing files in Sources / Network).
4. **OSINT useful for external assessment:** searches in public repos / npm / web indexes allow estimates of public exposure to such resources, without resorting to intrusive methods.
5. Mitigation: simple measures (ponyfill, self-host, SRI, CSP) significantly reduce the risk, and automated controls (CI checks, SCA) can detect unauthorized changes. [11], [12], [13].

Security implications

Compromising a centrally distributed polyfill translates into a "one-to-many" risk: an incident at the provider can very quickly affect a large number of sites. Furthermore, behavioral changes can remain undetected for a period of time (logical issues, regressions), which complicates detection and remediation. Therefore, third-party trust policies for front- end resources must be replicated through technical and procedural controls.

Technical recommendations

1. Avoid global runtime polyfills - prefer ponyfills or integrating polyfills into the build process (self - hosting) [7], [8], [10], [16].
2. **Self- host** critical resources and include them in your build pipeline to control exactly what code runs in production.
3. **Subresource Integrity (SRI)** - use SRI to block the loading of modified scripts, if a CDN must be used.
4. **Content Security Policy (CSP)** - restrict script sources to approved ones.
5. **CI/CD checks** - run automated checks that detect the presence of external polyfills, compare hashes, and inspect toString() on critical functions.
6. **SCA & pinning** - use SCA tools, pin package versions, and periodically review dependencies.
7. OSINT Monitoring & Alerting - monitor public mentions and relevant CDN inclusions to estimate exposure and detect changes [11], [12].

Organizational recommendations and procedural

- Implement a clear policy on including resources at runtime and responsibilities for third-party review parts.
- Prepare response procedures (rollback, isolation, internal and external communication) for supply chain incidents.
- Include didactic exercises (lab) as part of developer training to increase risk awareness.

Study limitations

- The experiment was performed in a controlled environment (local / VM) and uses demo scripts; production impact may involve additional variables (CDN cache, optimization tools, browser extensions).
- OSINT analysis provides estimates of public exposure, but does not replace internal inventory (SBOM) for organizations.
- The demonstration focused on one example (findLast); other APIs or polyfills may have different and sometimes worse effects.

References

[1]. Sansec Forensics Team, "Polyfill supply chain attack hits 100K+ sites," Sansec Blog, 2024.
[2]. A. Sharma, "Polyfill.io supply chain attack hits 100,000 websites - all you need to know," Sonatype, 2024.
[3]. J. Graham-Cumming et al., "Automatically replacing polyfill.io links with Cloudflare's mirror for a safer Internet," Cloudflare Blog, 2024.
[4]. Akamai Security Blog, "Examining the Polyfill attack from Akamai's point of view," Akamai, 2024.
[5]. S. Sarva, "Polyfill.io supply chain attack: what you need to know," Qualys, 2024.

[6]. Censys Threat Research, "Polyfill.io supply chain attack - digging into the web of compromised domains," Censys Blog, 2024.
[7]. NIST, "Software Supply Chain Security Guidance (EO 14028)," NIST, 2021.
[8]. OWASP, "Software Supply Chain Security Cheat Sheet," OWASP Cheat Sheet Series.
[9]. FOSSA Blog, "Polyfill supply chain attack: details and fixes," FOSSA, 2024.
[10]. T. Zimmermann et al., "On the prevalence of software supply chain attacks: empirical study," 2022.
[11]. M. Bazzell, "Open Source Intelligence Techniques: Resources for Searching and Analyzing Online Information", 9th ed., 2021.
[12]. Europol, "Guidelines for OSINT investigations," Europol Publications, 2020.
[13]. T. Rid, "Cyber War Will Not Take Place", Oxford Univ. Press, 2013.
[14]. Mozilla Developer Network, "Using polyfills in JavaScript."
[15]. npm Registry, "Postmortem incident event-stream," npm Blog, 2018.
[16]. Snyk Research, "Polyfill supply chain attack - analysis & mitigation guidance," Snyk, 2024.