# The Web Ecosystem Between Vulnerability and Resilience: The Case of Polyfill.io

**Adelaida STĂNCIULESCU**
Bucharest Court, Bucharest, Romania
adelaida.stanciulescu@gmail.com

**Abstract**

*This report addresses the Polyfill.io security incident, analyzed as a case study to demonstrate the impact that poor management of external dependencies can have on the resilience of the modern web ecosystem. The Polyfill.io case represents one of the most extensive external dependency compromises in recent history, affecting over 100,000 websites through sophisticated mechanisms for injecting malicious code via compromised CDN infrastructure. The analysis reveals how a supposedly harmless JavaScript library, used for cross-browser compatibility, was hijacked and exploited as a global attack vector. The study investigates the mechanisms by which the polyfill.io domain was taken over and used for the conditional distribution of malicious code. Through comparative analysis with other major supply incidents chain (SolarWinds, Log4Shell, XZ Utils), the paper identifies the unique features of the Polyfill.io case - including the passive nature of the compromise, the almost instantaneous speed of propagation, and the unprecedented diversity of victims. The results of the analysis reveal the importance of implementing strengthened security measures for managing external dependencies, such as systematically verifying the integrity of resources, enforcing content security policies, and continuously monitoring ownership changes within open- source projects.*

**Index terms:** CDN compromises, external dependencies, open-source security, Polyfill.io, supply chain attacks

## 1. Introduction

### 1.1. The general context of supply chain attacks chain in the contemporary web ecosystem

The contemporary web ecosystem has undergone a significant structural transition from traditional monolithic applications to distributed architectures based on microservices and extensive integration of external dependencies. This evolution has generated considerable advantages in terms of scalability and speed of software development, but has also introduced new levels of complexity and vulnerability from a cybersecurity perspective [14], [15], [16].

The contemporary web ecosystem is characterized by a complex interconnection of multiple components: web applications, servers, cloud services, content delivery networks (CDNs), and a multitude of APIs. In modern software development, code reuse through external libraries and open-source services is a standardized practice, which accelerates innovation and reduces implementation costs [14], [15].

However, this interdependence raises security concerns: compromising a single external component can generate a domino effect, propagating the vulnerability to tens or hundreds of thousands of applications. Recent examples (SolarWinds, Log4j, Polyfill.io, XZ Utils) have demonstrated that the modern web ecosystem is not just a technological assembly, but also a fragile chain of trust [9], [10], [14], [15].

Software supply chain attacks are one of the most insidious and effective forms of cyberattacks. Unlike traditional attacks that directly target an application or website, these attacks exploit vulnerabilities in the ecosystem of dependencies.

Two main types of attacks relevant to the present study are identified in the specialized literature:

- **Attacks on open- source dependencies** - consist of compromising a popular library published on platforms such as *npm* (Node Package Manager). A notorious example is the "event- stream" incident, where a widely used package was hijacked to insert malicious code.
- **Attacks on distribution infrastructure** - consist of compromising or acquiring content delivery services (especially CDNs), through which code reaches end-user browsers directly. This is exactly the category illustrated by the *Polyfill.io incident*.

**The Polyfill.io case was particularly serious** because the service was deeply embedded in the code of a significant number of websites and had a high level of trust from the developer community. This positioning made it an ideal vector for a supply -side attack. chain on a global scale.

### 1.2. External dependencies: definitions and classifications

In a strict sense, an external dependency is any third-party software component or service integrated into an application. They can be classified as follows:

- Direct dependencies - used explicitly by the application (e.g.: a JavaScript library imported by the developer).
- Transitive dependencies - automatically included through other libraries, without the developer being aware of their presence.
- Runtime dependencies - used in application execution (e.g. scripts loaded from a CDN - Content Delivery Networks).
- Build-time dependencies - involved in the compilation, testing or deployment process.
- The contemporary software ecosystem is also characterized by a series of structural features that amplify both the efficiency and vulnerability of modern applications:
- Recursive dependencies - a medium-sized project can include hundreds or even thousands of external packages, a phenomenon that exponentially increases the attack surface.
- Hierarchical complexity - the structure of dependency trees is often nested and opaque, making it difficult to map and manage them effectively.
- High dynamics of changes - frequent updates of external components require continuous monitoring, auditing and integrity validation processes.

These features, while contributing to the acceleration of innovation and flexibility of development, at the same time introduce an increased level of uncertainty and risk from a cybersecurity perspective.

In the JavaScript ecosystem, for example, an average project can include hundreds or even thousands of indirect dependencies, making it nearly impossible to manually verify the integrity of all components. Thus, poor dependency management becomes a structural vulnerability of the web ecosystem.

### 1.3. Vulnerabilities in software supply chain (software supply chain)

Software supply chain chain) represents the set of processes, resources and entities involved in the development and distribution of code. Vulnerabilities can occur at several levels: [14], [15], [16]

- **Compromising source code -** by injecting backdoors into popular libraries (e.g. attacks on npm packages).
- **Infrastructure compromise** - by hijacking a distribution service (e.g.: the Polyfill.io incident, where domain compromise led to the delivery of malicious scripts).
- **Design vulnerabilities** - lack of integrity checking mechanisms (Subresource Integrity, Content Security Policy).
- **Poor project ownership management** - changing or misappropriating an open- source domain/package (e.g. XZ Utils).

These vulnerabilities demonstrate that the security of modern applications depends not only on the organization's own code, but also on the entire ecosystem of integrated external components.

The diagram in the following image (Fig. 1) complements the above analysis, illustrating the typical flow of the software supply chain, from developers and maintainers (***developers who manage software projects, administrators, operators, etc.***) to the final application. The diagram highlights (Fig. 1) the critical points where vulnerabilities can be introduced and through which an attack can be propagated on a large scale.

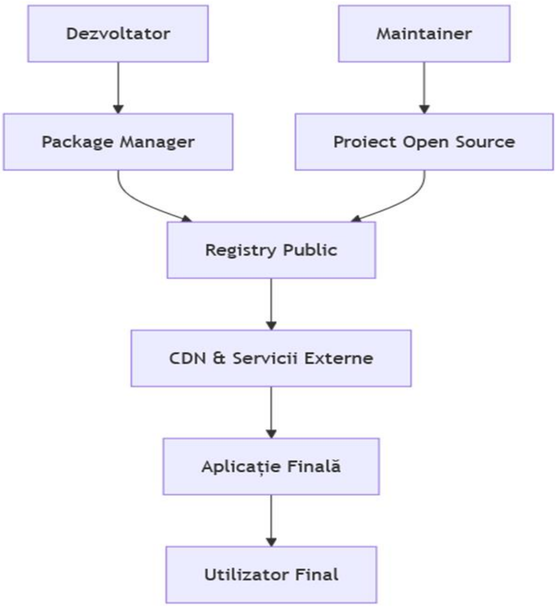


**Fig. 1.** Software supply flow chain and potential vulnerabilities

### 1.4. Methodology used (OSINT, technical documentation, public sources)

The present analysis was carried out through a mixed methodology, combining OSINT investigations with technical analyses and corroboration from official sources [1]-[9], [13], [14], [15].

**Main directions:**

1. **OSINT (Open Source Intelligence):** The collection, indexing, and correlation of information from public sources - technical reports and posts (e.g. Sansec, Cloudflare), press releases, technical articles, and discussions on social platforms (X/ Twitter, Reddit, Mastodon). This activity aims to identify indicators of compromise (IOCs), timelines of events, and community reactions [1], [3]-[9], [13].
2. **Technical documentation:** static and dynamic analysis of JavaScript code provided by Polyfill.io associated servers (download, search for redirect patterns), DNS/WHOIS investigations and examination of the described technical signatures [1], [3]-[5], [7], [8], [12].

3. This component includes **testing in isolated environments** (VMs) to confirm malicious behavior without exposing production systems.
4. **and official sources:** corroborating information with press releases and actions of institutional actors (e.g.: announcements by registry/domain operators, notifications by cloud /CDN providers, warnings issued by search engines or advertising platforms). This source allows for the assessment of the impact on a macro scale (market, reputation, provider involvement) [8], [9], [10], [11], [13], [15].

By combining the three dimensions, a comprehensive perspective is obtained, covering everything from fine technical details to the operational and economic consequences of the incident.

## 2. Technical analysis of the polyfill.io incident

**Polyfill.io** was an open- source **service** that provided JavaScript scripts to ensure compatibility of web applications with older browsers. By using this service, developers could automatically load only the functionality needed for a particular browser, optimizing performance and reducing maintenance effort [1]-[5].

### The role of Polyfill.io in the web ecosystem

- **Universal compatibility** - allows modern websites to run on older browsers, expanding the audience.
- **Reducing developer effort** - instead of testing every feature on all browser versions, developers included a simple <script src ="https://polyfill.io/v3/polyfill.min.js">.
- **Standardization** - Polyfill.io has become a central point, adopted by **millions of sites** (government, commercial, educational), being perceived as an "infrastructure" service of the web.
- **Systemic risk** - it is precisely this centrality that created a critical dependency: compromising the polyfill.io domain meant instantly affecting a huge number of sites, without any action on their part.

### Attack vectors and operating methods

The **incident did not directly compromise the original source code** of the *Polyfill project,* the project code (open-source, published on GitHub) remained untouched. The incident occurred after the polyfill.io **domain and associated CDN infrastructure** were taken over by a malicious actor (by purchasing the domain after it expired) who compromised them by *injecting malicious code* [1]-[5], [7], [8], [12].

Thus, the incident is classified as an **attack on the distribution infrastructure in the software supply chain**, not as an attack on the open- source code itself.

Once the attacker controlled **the CDN service**, they could:

- **Modify the delivered content** - instead of just providing legitimate polyfills, inject malicious JavaScript.
- **Personalize delivery** - malicious scripts were served selectively, depending on browser or region (conditional delivery), which made them harder to detect.
- **It exploited implicit trust** - millions of sites directly included <script src="https://polyfill.io/...">, without integrity checking.

## 3. Case study

Simulating the introduction of **the Array.prototype.findLast() function** in browsers: Chrome, Firefox, Opera and Internet Explorer and the polyfill risk, using **OSINT tools for analysis**.

To assess the behavior and risks associated with introducing certain functions JavaScript through *polyfill* mechanisms, a local testing platform was created in the form of a web page. This platform allowed verification of the existence and functionality of the Array.prototype.includes () method through *feature* techniques *detection*, as well as comparing the behavior when the function is provided natively by the browser engine versus when it is provided by an external polyfill.

The simulation consisted of two controlled scenarios:

- running the page without any additional script, to observe the native behavior of the browser.
- Deliberately loading a polyfill that implements or exposes the function globally, thus simulating the situation where a third-party script - potentially compromised - is present on the page.

The results were collected through screenshots, and the analysis focused on identifying the attack surface introduced by global polyfills.

This experimental architecture provides a reproducible and reliable basis for discussing the implications SAFETY supply-chain in the context of front- end dependencies.

Array.prototype.findLast () function was introduced in:

- Firefox version 143 (2025).
- Opera version 122 (2025).
- Not supported in Internet Explorer.
- Not supported in Chrome version 96 (2022).

In these browsers, the Array.prototype.findLast () function was identified using the query presented in Fig. 2:

```
<!doctype html>
<html>
<head><meta charset="utf-8"><title>Check findLast</title></head>
<body>
  <h2>Check Array.prototype.findLast()</h2>
  <pre id="out"></pre>
  <script>
    const out = document.getElementById('out');
    if ('findLast' in Array.prototype) {
      out.textContent = 'findLast disponibil nativ';
      console.log('findLast disponibil nativ');
    } else {
      out.textContent = 'findLast NU este disponibil — foloseşte shim/ponyfill';
      console.log('findLast NU este disponibil — foloseşte shim/ponyfill');
    }
  </script>
</body>
</html>
```

**Fig. 2.** The Array.prototype.findLast ()

The results are shown in the following images (Fig. 3- Fig. 6):

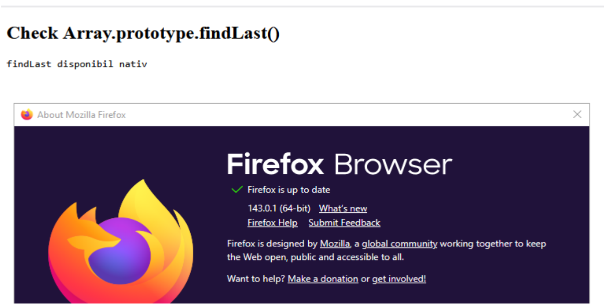


**Fig. 3.** Checking the existence of the function in the browser Firefox

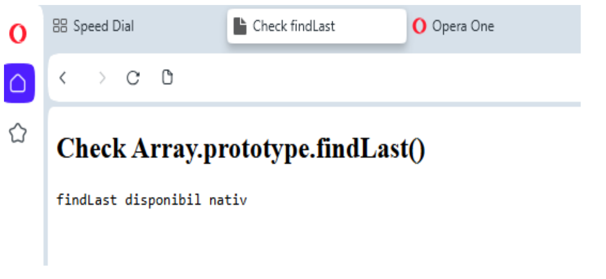


**Fig. 4.** Checking the existence of the function in the Opera browser

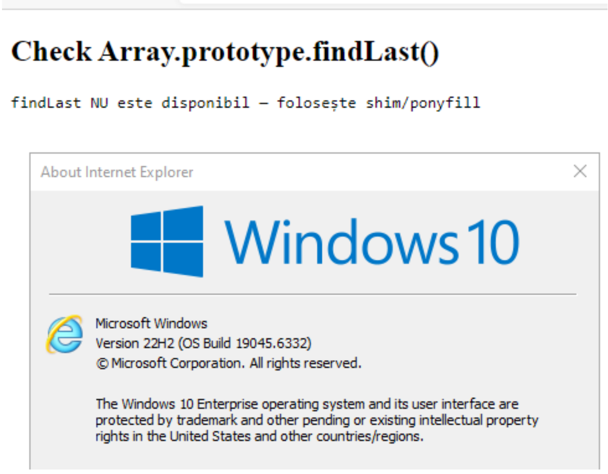


**Fig. 5.** Checking the existence of the function in the Internet Explorer browser

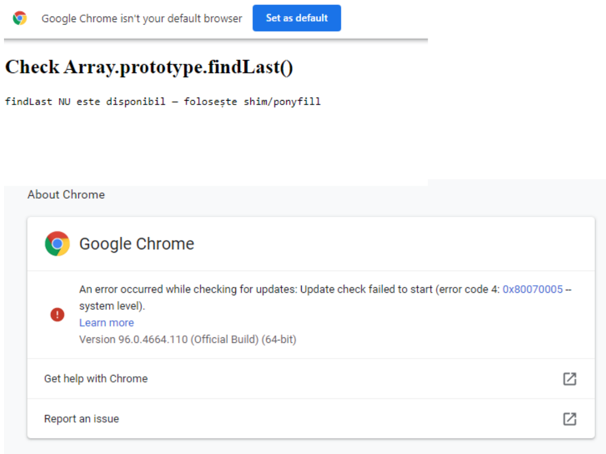


**Fig. 6.** Checking the existence of the function in the browser Chrome version 96

Let's assume that the JavaScript function array.prototype.includes() is standard. Newer browsers (Chrome 87+, Firefox 43+) implement it, but older browsers (Internet Explorer) do not. A polyfill for this function would check if the function already exists in the browser. If it doesn't, it *creates it* using backward-compatible JavaScript code (Fig. 7 - Fig. 10).

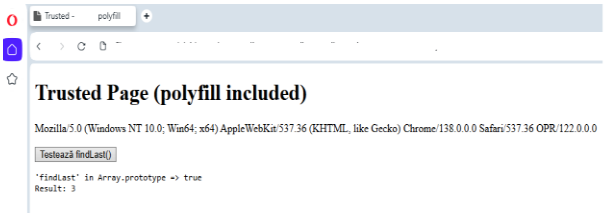


**Fig. 7.** Firefox/143.0

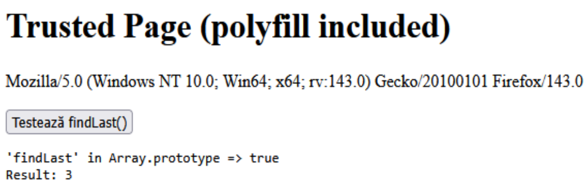


**Fig. 8.** Chrome/138.0.0.0

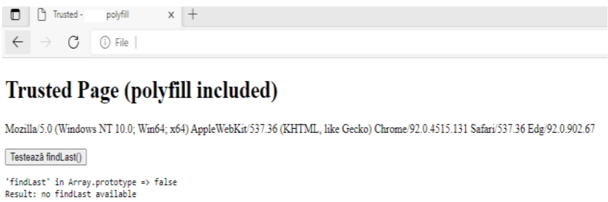


**Fig. 9.** Chrome/92.0.4515.131

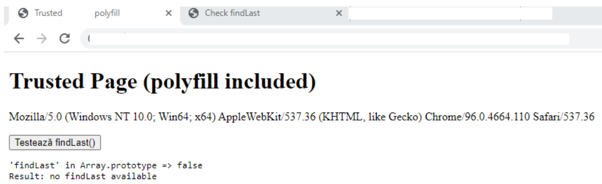


**Fig. 10.** Chrome/96.0.4664.110

Simulating a page that loads the polyfill (simulates a page that could abuse it if the polyfill is global). The code run is presented in Fig. 11.

**After running the code in the browser Chrome v.96** (*no* native implementation); result is false.

**The polyfill is not installed globally** - the "safe" version is running which does not expose the method globally (deliberate intention for security).

*ponyfill / shim* was used which **does not modify** Array.prototype (and then findLast does not appear globally, it remains available only through the helper).

**The result " no global findLast " indicates that a malicious page cannot call arr.findLast (...) directly** (in this session), so this concrete attack surface does not exist at the time of capture.

In Fig. 12, the attacker *does not* have access to the global API.

```html
<!doctype html>
<html lang="ro">
<head>
  <meta charset="utf-8">
  <title>Attacker - demo polyfill</title>
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <!-- Attacker includes the same script (simulating a compromised CDN) -->
  <script src="findLast-polyfill.es5.js"></script>
</head>
<body>
  <h1>Attacker Page (polyfill included)</h1>
  <p id="status"></p>
  <button onclick="tryAbuse()">Try abuse (read global)</button>
  <pre id="out"></pre>
  <script>
    function tryAbuse(){
      const out = document.getElementById('out');
      out.textContent = '';
      try {
        out.textContent += "'findLast' in Array.prototype => " + ('findLast' in Array.prototype) + "\n";
        if ('findLast' in Array.prototype) {
          var arr = ['a','b','c'];
          var res = arr.findLast(x => x === 'b');
          out.textContent += 'Attacker read result: ' + String(res) + '\n';
          console.log('Attacker read:', res);
        } else {
          out.textContent += 'no global findLast';
        }
      } catch(e){
        out.textContent += 'Error: ' + e.message;
      }
    }
    document.getElementById('status').textContent = navigator.userAgent;
  </script>
</body>
</html>
```

**Fig. 11.** Simulating a page that loads the polyfill

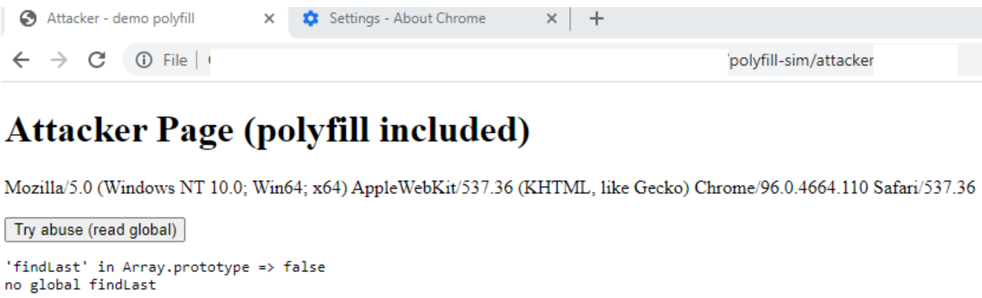


**Fig. 12.** The attacker *does not* have access to the global API

**After running the code in the browser Firefox (v ~143) notices that** *findLast* is present in Array.prototype (value is **true**) (Fig. 13).

This value can be the result of:

- **The browser 's native implementation** (in your case Firefox 143, findLast is natively supported from versions much older than 143).
- **Polyfill that installed the method globally** (if you loaded an external script that had modulated Array.prototype).


**Fig. 13.** The running the code in the browser Firefox (v ~143)

Since the user -agent indicates Firefox v143 (updated browser), most likely findLast is **native**, but we cannot rule out the fact that the page also loaded a legitimate polyfill (which would also indicate the value true when calling *findLast*).

**OSINT elements used in the study**

In this study, the analysis of polyfill adoption and exposure was based on a structured set of *Open Source Intelligence (OSINT) techniques* and tools. The goal was to obtain verifiable and reproducible evidence of the use of polyfill functions. JavaScript (in particular Array.prototype.findLast) and polyfill distribution mechanisms, without resorting to intrusive or access-violating actions [1]-[9], [13]-[15].

The analysis capitalized on a set of complementary sources:

- repositories (e.g. GitHub).
- Package registry (e.g.: npm) - consulted to detect packages that provide shim / ponyfill and to understand the distribution mode (dedicated packages vs. CDN inclusions).
- archives and indexes (e.g. Common Crawl, Wayback Machine) - used for quantitative estimates: frequency the appearance of references to certain CDNs in historical or current web pages.
- Search engine - for identifying mentions in articles, issues and technical discussions, useful in contextualizing decisions.

Certificate transparency / crt.sh - to check related events related to domains hosting polyfills or CDNs (only for passive mapping of public domains).

## 4. Conclusions

The research exclusively targeted public data and open resources; any private or restricted information accidentally identified was ignored and not collected or stored.

Results from the analysis were written to support risk management, not to facilitate exploitation. OSINT tools, used within an ethical framework and reproducible, allow the identification and assessment of polyfill exposure and distribution practices in the web ecosystem.

The described methodology provides a solid basis for informative estimates, useful both for research/investigation and for risk management plans at the organizational level [14], [15], [16].

## References

[1]. Sansec Forensics Team, "Polyfill supply chain attack hits 100K+ sites," Sansec, technical analysis and timeline of the compromise affecting the polyfill.io domain and the injection of malicious code via CDN scripts, 2024.

[2]. A. Sharma, "Polyfill.io supply chain attack hits 100,000 websites - all you need to know," Sonatype, blog post summarizing impact and initial recommendations for organizations, 2024.

[3]. J. Graham-Cumming and Cloudflare Team, "Automatically replacing polyfill.io links with Cloudflare's mirror for a safer Internet," The Cloudflare Blog, discusses mirroring/URL rewrite and practical mitigations, 2024.

[4]. Akamai Security Blog, "Examining the Polyfill attack from Akamai's point of view," Akamai, technical analysis and CDN/security provider perspective, 2024.

[5]. Snyk Research, "Polyfill supply chain attack - analysis & mitigation guidance," Snyk, contextualizes the incident in the JavaScript ecosystem and proposes fixes and scanning approaches, 2024.

[6]. S. Sarva, "Polyfill.io supply chain attack: what you need to know," Qualys, technical note with detection and scanning recommendations for organizations, 2024.
[7]. Censys Threat Research, "Polyfill.io supply chain attack - digging back the web of compromised domains," Censys, investigation mapping domains and hosts involved, 2024.
[8]. FortiGuard Labs, "Polyfill.io Supply Chain Attack - Threat Signal Report," Fortinet, summary of indicators of compromise (IoCs) and recommendations, 2024.
[9]. The Hacker News, "Polyfill.io attack impacts 380,000+ hosts," The Hacker News, journalistic coverage of scope and follow-ups, 2024.
[10]. NVD, "CVE entries related to Polyfill.io usage (e.g. CVE-2024-38526)," NIST National Vulnerability Database, technical CVE details and references to affected packages, 2024.
[11]. Cloudflare, "Automated rewrites & mirrors; guidance for customers to remove polyfill.io," The Cloudflare Blog, describes rapid large-scale mitigation steps, 2024.
[12]. FOSSA, "Polyfill supply chain attack: details and fixes," FOSSA Blog, explains attack mechanics and remediation steps for development teams, 2024.
[13]. Checkmarx, Invicti, Arctic Wolf, Kaspersky, "Polyfill.io supply chain attack - vendor advisories and practical recommendations," multiple vendor security advisories, 2024.
[14]. OWASP, "Software Supply Chain Security Cheat Sheet," OWASP Cheat Sheet Series, practical recommendations for assessing and mitigating software supply chain risks (SCA, SRI, self-hosting, SBOM, etc.), 2023.
[15]. NIST, "Software Supply Chain Security Guidance (EO 14028)," NIST guidance documents on policies and standards for software supply chain security, 2021-2024.
[16]. "On the prevalence of software supply chain attacks," and related academic studies on supply chain attack taxonomy and investigation frameworks, 2020-2024.