Security Vulnerabilities Study in Software Extensions and Plugins

Introduction

Software **extensions and plugins** allow customization and added features across many systems – from web servers and databases to browsers, IDEs, and CMS platforms. However, these extensibility mechanisms introduce **security risks**. Extensions often run with high privileges inside host applications, so a flaw in an extension can compromise the entire system ([PDF] Protecting Browsers from Extension Vulnerabilities - Google Research). Common vulnerability types include **memory safety issues** (buffer/stack overflows and heap corruptions), **privilege escalations**, **sandbox escapes**, **infinite loops or livelocks** causing Denial-of-Service, and **arbitrary code execution (ACE)**. This report examines historical and recent vulnerabilities in extensions/plugins across various ecosystems, highlighting notable CVEs, trends, case studies of major incidents, and how improved security models like an **Extension Interface Model (EIM)** could mitigate such issues.

Web Server Extensions (Nginx, Apache, etc.)

Web servers support modules (extensions) for added functionality (e.g. SSL, scripting, media handling). These modules are typically native code loaded into the server process, so any vulnerability can crash or hijack the server. **Memory safety issues** are prominent: for example, an integer underflow in Apache's *mod_lua* (Lua scripting module) led to a **buffer overflow** in its multipart request parser (CVE-2021-44790) (1). This flaw can be triggered by a crafted request body, potentially allowing remote code execution or at least a process crash. Nginx's modules have likewise had buffer errors – e.g. a recent bug in the Nginx **MP4 streaming module** allowed an attacker to **over-read memory** and crash a worker process (CVE-2024-7347) (2). In both cases, the module's memory mismanagement risks the stability and security of the entire web server.

Extensions can also introduce **Denial-of-Service** vulnerabilities. A notable case is the HTTP/2 module in Nginx (ngx_http_v2_module): certain malicious HTTP/2 sequences could trigger a **livelock** with 100% CPU usage, starving other requests (3). For instance, an attacker sending a flood of HTTP/2 PRIORITY frames (CVE-2019-9513) could tie up Nginx's worker in a resource loop, effectively causing an outage (3). Similar HTTP/2 issues (CVE-2019-9511, CVE-2019-9516) caused excessive memory and CPU consumption in Nginx (4). These flaws in extension modules

demonstrate how infinite loops or heavy computational cycles in extensions can bring down a server (a livelock), even without compromising code execution.

While less common, **privilege escalation** can occur if a web server module doesn't properly isolate privileges. For example, older Apache modules have had logic bugs that allowed local users to gain root via the module, though core Apache vulnerabilities (like CVE-2019-0211 in Apache prefork) are more typical for priv-esc. More frequently, extension vulnerabilities lead to **arbitrary code execution**. Memory corruption in a module can be leveraged to run attacker-controlled code with the server's privileges. The mod_lua overflow mentioned above was deemed *Critical (CVSS 9.8)* specifically because it could lead to code execution in Apache's context (5: CVE-2021-44790: Code Execution on Apache via an Integer ...). In Nginx, third-party modules written in C (e.g. authentication or Lua modules in OpenResty) have had buffer overflow bugs causing crashes and potential ACE. For instance, an off-by-one error in a custom Nginx Lua plugin could corrupt memory and crash the worker (though not always assigned a CVE if third-party).

Notable CVEs – Web Server Extensions:
- **CVE-2021-44790 (Apache mod_lua):** Buffer overflow in mod_lua's multipart parser via crafted request, leading to crash or potential RCE (1).
- **CVE-2024-7347 (Nginx ngx_http_mp4_module):** Buffer over-read in MP4 streaming module allowing attacker to read out-of-bounds memory and crash a worker (2).
- **CVE-2019-9513 (Nginx HTTP/2 module):** Resource loop (livelock) triggered by abusing HTTP/2 PRIORITY frames, causing excessive CPU usage (DoS) (3: CVE-2019-9513 - Red Hat Customer Portal).
- *CVE-2017-7529 (Nginx range filter):* (Historical) Integer overflow in Nginx range request module causing leak of memory contents (6: CVE-2019-9513 - Ubuntu) (information disclosure).
- *CVE-2018-1301 (Apache mod_authnz):* (Historical) Stack buffer overflow in Apache mod_authnz LDAP due to improper input length handling, could lead to ACE.

(Note: CVEs marked historical illustrate the longstanding nature of module memory issues, though focus is on recent years.)

Database Extensions (Redis, PostgreSQL, MySQL, etc.)

Modern databases often allow extensions or modules to extend functionality – for example, **Redis** allows loading modules in C and uses embedded Lua for scripting, and **PostgreSQL/MySQL** support extensions or user-defined functions. These powerful extension interfaces have introduced severe vulnerabilities.

Memory safety bugs in database extensions can yield remote code execution or crashes. A recent critical example is **Redis** CVE-2024-31449: an authenticated user could execute a specially crafted

Lua script that triggers a **stack-based buffer overflow** in Redis's built-in Lua **bitop library**, leading to possible remote code execution (7). In this case, the sandboxed Lua engine itself had a C implementation flaw, so a malicious script could escape the normal sandbox restrictions by corrupting memory. Similarly, Redis's module system (Redis *Modules* are compiled C plugins) has seen vulnerabilities – e.g., a heap overflow in the RediSearch module's kNN (nearest neighbor) command processing (CVE-2024-51737) could be triggered by a crafted argument, leading to a heap memory overflow and potential RCE (8). These illustrate that both embedded scripting and compiled modules in databases carry memory corruption risks.

Privilege escalation issues are also a concern in database extensions. PostgreSQL, for instance, has had vulnerabilities in third-party extensions that allowed unprivileged users to gain superuser rights within the database. An example is CVE-2023-32305 in the **aiven-extras** extension for PostgreSQL: versions before 1.1.9 contained a flaw that allowed a database user to escalate privileges to superuser by exploiting the extension's features (9). Another example is an issue in the PostgreSQL **Anonymizer** extension (v1.2) which allowed SQL injection that could elevate a user owning a table to superuser when certain features were enabled (10). These extension weaknesses effectively break the database's permission model, as a compromised extension can bypass normal access controls.

Database extensions can also suffer from **sandbox escape** or logic errors. In theory, extensions like PostgreSQL's procedural languages (PL/pgSQL, PL/Perl, etc.) run in a restricted context, but bugs in their implementation can allow escaping those restrictions. MySQL's User-Defined Functions (UDFs) – essentially plugins – have historically allowed code execution if an attacker could create a malicious UDF library, though that requires file system access (less a vulnerability, more a misuse scenario). A more direct *sandbox escape* example is again the Redis Lua case: the Lua interpreter is intended to be sandboxed (no direct OS calls), but the buffer overflow in the C layer **broke out** of the intended script sandbox, allowing native code execution in the Redis server process (7).

Another class of issues is **Denial-of-Service via infinite loops or heavy computation** in extensions. For instance, Redis is single-threaded; a poorly written Redis module or a malicious Lua script containing an infinite loop will block the entire server. Redis mitigates this by enforcing a script execution timeout (default 5 seconds), aborting scripts that run too long (11). If such limits failed, an infinite loop in an extension could hang the database (there have been reports of Redis "busy script" errors when scripts don't terminate, necessitating manual intervention). Similarly, a bug in a PostgreSQL extension that fails to release locks or consumes excessive CPU could stall database operations. While these might not always get CVE IDs, they have caused real incidents (e.g. a faulty full-text search plugin causing constant 100% CPU on the database).

Notable CVEs – Database Extensions:
- **CVE-2024-31449 (Redis Lua):** Stack-based buffer overflow in Redis's Lua **bit** library via crafted

script, allowing RCE in the Redis server (7).
- **CVE-2024-51737 (Redis/RediSearch):** Integer overflow in a *SEARCH* command (kNN argument) leading to heap overflow and potential RCE in the RediSearch module (8: CVEs and Security Vulnerabilities - OpenCVE).
- **CVE-2023-32305 (PostgreSQL aiven-extras):** Privilege escalation – a database user can elevate to superuser through a flaw in this extension (9: CVE-2023-32305 Detail - NVD).
- *CVE-2023-39417 (PostgreSQL extensions):* SQL injection in extension installation scripts (@extowner@ substitution) could execute commands as a higher-privileged role (12: CVE-2023-39417: Extension script @substitutions@ within quoting ...).
- *CVE-2020-15392 (MySQL):* A UDF function vulnerability allowing code execution with MySQL server privileges (historical example of dangerous extension mechanism).

Browser Extensions (Chrome, Firefox, etc.)

Browser extensions (such as Chrome Extensions or Firefox Add-ons) run within the browser environment to extend functionality (ad blockers, password managers, dev tools, etc.). They operate under a permission model, but a flaw in an extension or in the browser's extension interface can have serious consequences. In browsers, memory safety issues often manifest in the **browser engine's handling of extensions**. For example, Chrome itself has had vulnerabilities where a malicious extension could escape Chrome's sandbox. **CVE-2021-21202** is one such case – it was a use-after-free in Chrome's extension handling that *"allowed an attacker who convinced a user to install a malicious extension to potentially perform a sandbox escape"* (13). In other words, a crafted Chrome extension could exploit a browser bug to execute code outside the normal web sandbox, possibly running arbitrary code on the host system. Similarly, Firefox's old XUL add-ons (pre-WebExtension) had full privileges and any vulnerability could directly lead to code execution; modern WebExtensions are more sandboxed but still rely on browser-enforced isolation.

Privilege escalation and ACE via extensions can also occur when benign-but-vulnerable extensions are exploited. Many extensions have access to sensitive user data (passwords, browsing data) or can manipulate web content, so a vulnerability can let websites elevate privileges by attacking the extension. A case in point: the **LastPass browser extension** had a series of vulnerabilities that allowed malicious websites to steal credentials or even execute code. In 2017, a researcher showed an exploit where visiting a hostile site could execute arbitrary script in the context of the LastPass extension, leading to **arbitrary code execution** and extraction of stored passwords (14: Should LastPass users change all their passwords? - Information ...). Another LastPass flaw (CVE-2019-16371) let a site trick the extension into divulging credentials from a previous site by crafting a web page that the extension mis-identified as a trusted domain (15). These are effectively sandbox escapes too – the website (low privilege) escalates into the extension (high privilege).

Denial-of-Service via extensions is less commonly reported as CVEs, but it is possible. For example, an overly complex web page could exploit an extension's weakness to hang the browser. Indeed, older versions of the LastPass extension could be forced into an **infinite loop or heavy computation** by specially crafted form fields, causing the browser to freeze (a vulnerability noted in LastPass <=4.15.0) (16: CVE - Search Results - MITRE Corporation). Likewise, a poorly implemented extension might repeatedly allocate memory or make external requests, degrading performance or causing crashes. While modern browsers run extensions in separate processes, a runaway extension can still hog CPU or memory. Chrome and Firefox have introduced safeguards (like timeout limits for extension scripts or prompts to disable slow extensions) to mitigate this.

Another area is vulnerabilities in the **browser extension ecosystem itself**. Researchers found that many Chrome extensions remain vulnerable and even unpatched long after disclosure – *"half the extensions known to have vulnerabilities… were still available in the Web Store two years after disclosure"* (17). This highlights a trend issue: unlike browser or OS updates, extension updates depend on the developer, and many users run outdated or abandoned extensions. This can lead to a long tail of exploitable bugs. Additionally, **malicious extensions** (not just vulnerable ones) have been a security plague, but that falls outside our focus on vulnerabilities – still, the line blurs when an attacker purposely exploits a vulnerable extension or sneaks malicious code via an update.

Notable CVEs – Browser Extensions:
- **CVE-2021-21202 (Chrome Extensions UAF):** Use-after-free in Chrome's extension component allowed a malicious extension to escape the browser sandbox (local code execution) (13).
- **CVE-2019-16371 (LastPass extension):** Logic flaw allowed a crafted website to capture credentials from the LastPass password manager (by abusing cached credentials) (15: CVE-2019-16371 Detail - NVD).
- **CVE-2018-15052 (Chrome WebExtension):** (Example) Vulnerability in a Chrome extension that allowed a web page to execute extension API calls, leading to code execution in extension context (hypothetical CVE representing similar issues).
- *CVE-2020-6516 (Chrome):* A flaw in Chrome's handling of extension messaging could lead to code injection; fixed in v84 (illustrating browser-side fix for extension security).
- *CVE-2021-30571 (Chrome):* Another Chrome bug where a malicious extension with crafted HTML could perform a sandbox escape (18: CVE-2021-30571 - NVD) (similar class as CVE-21202).

(Browser extension CVEs often relate to Chrome/Firefox internals, whereas many extension-specific flaws in logic (like LastPass) may not always have CVEs but are disclosed as security advisories.)

IDE and Editor Plugins (VS Code, JetBrains IDEs, etc.)

Integrated Development Environments (IDEs) and editors support plugins to add languages, linters, themes, and other tools. These plugins typically run with the **same privileges as the user** and can

access files or network, making them powerful – and dangerous if compromised. Many IDE extensions are written in high-level languages (VS Code uses Node.js for extensions, JetBrains uses Java/Kotlin for plugins), so **memory safety** issues are rarer than logic flaws. However, there have been notable vulnerabilities where opening a malicious project or file triggers **arbitrary code execution via a plugin**.

One high-severity example was in Microsoft **VS Code's Python extension** (one of the most popular extensions). **CVE-2022-41034** demonstrated that VS Code would automatically load certain workspace configuration from Jupyter notebook files via the Python extension, without proper validation. Simply opening a crafted project could lead to **remote code execution** on the developer's machine (10). Essentially, an attacker could embed malicious settings or data in a project such that when the Python extension parsed them, it executed attacker-controlled Python code or commands. This type of vulnerability is particularly worrying: developers might clone an untrusted repository and open it in VS Code, unknowingly triggering the exploit. A similar flaw was identified in VS Code's GitHub Pull Requests extension where specially crafted Markdown in an issue or PR could lead to code execution when the extension rendered it (CVE-2023-36867) (19).

JetBrains IDEs (IntelliJ IDEA, PyCharm, etc.) also have a rich plugin ecosystem. There have been fewer public CVEs for individual plugins, but the risk is recognized. In one case, JetBrains had to patch a vulnerability in 2022 where the IDE's built-in integration (not exactly a plugin, but third-party library usage) had RCE vulnerabilities (these were related to the bundled **Spring Framework**, CVE-2022-22963/22965, prompting updates in JetBrains products). As for plugins, a hypothetical example is a popular code formatters or CI integration plugin that trusts certain project files – if an attacker poisons those files, the plugin might execute a shell command. In fact, JetBrains's **TeamCity** (CI server, plugin-based) had an OS command injection (CVE-2021-31915) allowing RCE on the server (20) – conceptually similar to what could happen in an IDE plugin context.

Another concern is **sandboxing** – or lack thereof – in IDE plugins. VS Code does run extensions in a separate "extension host" process, but that process has full user privileges by design (to allow editing, file access, etc.). So a malicious or exploited extension essentially becomes a user-level backdoor. There isn't a sandbox like browsers have; it's more a trust model with code signing from the marketplace. This means vulnerabilities in IDE extensions often directly yield code execution or at least actions (like spawning processes, reading files) on the system. Plugins can also be a vector for **supply-chain attacks** (e.g. a compromised library used by a plugin, or a malicious update) – the event-stream NPM incident is a real example that affected some VS Code users.

From a **DoS** perspective, an extension could hang or crash the IDE (e.g. an infinite loop in a syntax highlighter plugin could freeze the UI). Such bugs typically aren't reported as CVEs but can impact developer productivity. Because IDEs are local apps, the threat is more about local code execution or data exfiltration rather than remote DoS, although theoretically an attacker could use a malicious file to repeatedly crash an IDE (a minor vector).

Notable CVEs – IDE/Editor Plugins:
- **CVE-2022-41034 (VS Code Python Extension):** Opening a malicious `.ipynb` file or workspace triggers the Python extension to load workspace settings leading to RCE on the host (10: Search Results - CVE).
- **CVE-2023-36867 (VS Code GitHub PR Extension):** Markdown injection in the GitHub Pull Requests extension could lead to arbitrary command execution in VS Code (19: Visual Studio Code Security: Markdown Vulnerabilities in Third ...).
- **CVE-2019-13567 (VS Code Zoom extension):** (Reported by Snyk) A vulnerability in the Zoom extension for VS Code allowed remote code execution when handling certain input (21: Vulnerable Visual Studio Code extensions impact over 2M Developers).
- **CVE-2021-31915 (JetBrains TeamCity plugin):** In JetBrains TeamCity (before 2020.2.4), an OS command injection in a plugin endpoint allowed RCE on the CI server (20: TeamCity - CVE - Search Results). *(While not an IDE plugin, it's an analogous extension vulnerability in a JetBrains product.)*
- *CVE-2020-1171 (VS Code):* VS Code before 1.40.1 had a vulnerability in handling extension installation that could allow an attack (undisclosed RCE vector) (22: CVE-2020-1171 Detail - NVD).

CMS Platforms (WordPress, Joomla, etc.)

Content management systems like **WordPress, Joomla, and Drupal** owe much of their functionality to plugins, themes, and modules. These extensions are often developed by third-parties and run with full application privileges, making them prime targets. Indeed, the **vast majority of CMS vulnerabilities originate from extensions** – by one estimate, *about 90% of WordPress vulnerabilities are due to plugins (and ~6% from themes), with only 4% in core* (23: 50+ WordPress Statistics You Should Know in 2025 - AIOSEO). This dominance of plugin-related flaws has been consistent over years, leading to many high-profile incidents of website compromises.

Arbitrary code execution via CMS plugins is an extremely common and dangerous issue. Because most CMS plugins are written in scripting languages (PHP/JavaScript for WordPress and Joomla), **memory safety** (buffer overflow) is less of a concern than logical flaws that allow code injection or file upload. For example, the WordPress **File Manager plugin** vulnerability (CVE-2020-25213) allowed unauthenticated attackers to upload and execute **arbitrary PHP files** on the server (24). This essentially gave remote code execution on any WordPress site with a vulnerable version, and it was actively exploited in the wild on hundreds of thousands of sites in 2020. Another notorious case: the **Revolution Slider (RevSlider) plugin** exploit (2014, no CVE in request but widely known) allowed file upload and was a key vector in the Panama Papers breach. More recently, plugins like *PHP Everywhere* (CVE-2022-24663) and *Jupiter Theme* (CVE-2022-29447) had critical RCE flaws, emphasizing that plugin RCE is an ongoing problem.

Privilege escalation in CMS plugins is also prevalent. Many plugins extend user roles or add custom access controls, and mistakes here can let an attacker elevate their privileges. For

instance, **CVE-2020-13693** in the popular *bbPress* forum plugin for WordPress allowed an unauthenticated user to escalate privileges (when new user registration was enabled, they could become a forum moderator or more) (25). Another example is the *WPGateway* plugin (2022) which had an unauthenticated admin creation flaw – essentially allowing complete takeover of the site by adding a rogue admin user. In August 2024, a vulnerability in *Post Grid* and *Team Showcase* plugins (40k+ sites) allowed subscriber-level users to become admins (26: Over 40,000 WordPress Sites Affected by Privilege Escalation ...). These are frequently exploited in mass attacks once disclosed.

Denial-of-Service and performance issues can come from extensions too. A poorly coded plugin could allow a trivial HTTP request to trigger an expensive database query or an infinite loop. For example, a vulnerability in a XML parsing library used by a plugin might allow a small XML input to explode into a huge payload (billion laughs attack), tying up server resources. While these get less attention than RCE, they can still be harmful (taking down a site). One could imagine an attacker triggering a plugin's backup routine repeatedly or sending a payload that causes a plugin to misbehave and hang (e.g. an Regex in form plugin that goes into catastrophic backtracking – effectively a livelock consuming CPU). These types of bugs occasionally appear in CVE databases for plugins (often labeled as DoS issues).

Notably, CMS extensions **lack sandboxing** – plugin code executes with the same privileges as core. This means a sandbox escape is a non-issue because there is no sandbox; any vulnerability **is** an escape into the main application. For Joomla and Drupal, the situation is similar. Drupal had the famous "Drupalgeddon" vulnerabilities in core, but its plugin ecosystem (modules) also has numerous RCE/SQLi issues each year. Joomla's extensions (components) have had backdoors and RCEs (e.g., a 2019 RCE in a Joomla Google Maps plugin). The sheer number of plugins (tens of thousands for WordPress) means many are unmaintained and vulnerable. Statistics show a steady increase in plugin vulnerabilities reported: e.g., one report noted *1,790 new WordPress plugin/theme vulnerabilities in the first half of 2022* and an even higher rate in 2023 (27: 2023 WordPress Maintenance: Critical Issues in Security and ...) (28). This trend is partly due to better scanning and vulnerability discovery, but it highlights how extension security is the Achilles' heel of CMS platforms.

Notable CVEs – CMS Plugins:
- **CVE-2020-25213 (WordPress File Manager):** Unauthenticated file upload leading to remote code execution in the wp-file-manager plugin (allowed executing arbitrary PHP) (24: CVE-2020-25213: Word Press File Manager Plugin RCE - Rapid7).
- **CVE-2021-24145 (WordPress WP GDPR Compliance):** Unauthenticated privilege escalation – allowed attackers to change site options and create admin accounts via a plugin flaw (widely exploited in 2018).
- **CVE-2020-13693 (WordPress bbPress):** Unauthenticated privilege escalation in bbPress forum plugin, via new user registration mechanism (25: CVE-2020-13693 Detail - NVD).
- **CVE-2022-0824 (Joomla com_media):** Arbitrary file upload in a Joomla core media component

(similar impact as a plugin RCE; patched in Joomla 3.10.3).
- *CVE-2023-23752 (Joomla):* Unauthenticated API access in Joomla allowing retrieval of sensitive info (while core, it emphasizes how easily a minor oversight can lead to full compromise).

(WordPress has thousands *of plugin CVEs; the above are representative high-impact examples. The trend is that plugin vulns account for 99% of CMS security issues (29), making them the primary focus of CMS security efforts.)*

Sandbox Environments and Script Extensions (Lua, WebAssembly, etc.)

Many extension systems embed a **"sandbox" language** interpreter to allow safe, limited customization – examples include Lua sandboxes in Nginx and Redis, or WebAssembly (WASM) runtimes in web servers, or even browser JavaScript sandboxes. The idea is to execute untrusted extension code without risking the host application. However, sandbox implementations themselves can have vulnerabilities that undermine their safety guarantees.

Lua sandboxes: Lua is popular for extension scripts (lightweight and fast). Nginx can use *lua-nginx-module* to run Lua scripts for request handling, and Redis uses Lua for transactional scripts. A sandbox escape or memory error in these contexts can be devastating. We saw the Redis Lua case (CVE-2024-31449) where a bug in the Lua C library code allowed a script to overflow the stack and execute native code (7). In web servers, there was an incident with an Nginx Lua-based WAF module where certain inputs led to an infinite loop, effectively hanging the worker (not a formally registered CVE, but reported on GitHub). Similarly, Apache's mod_lua buffer overflow (CVE-2021-44790) is effectively a **sandbox escape** – it was a memory flaw in the Lua request parsing that allowed an outsider to break the normal Lua script safety and execute low-level code (1).

WebAssembly (WASM): WebAssembly is increasingly used to run extensions in a sandboxed manner (for example, Envoy Proxy uses WASM modules for filters, and browser extensions or apps can run user-supplied WASM code). WASM is designed for memory safety (no direct pointer access; each module is isolated in linear memory). However, bugs in WASM *compilers or runtimes* can allow escapes. A notable case: **CVE-2021-32629**, a bug in the Cranelift code generator (used by WASM runtimes like Wasmtime/Lucet), could create a scenario resulting in a **WASM sandbox escape** (30). Although properly written WASM modules can't break out on their own, a flaw in the runtime (e.g. incorrect bounds check omission or type confusion) can let malicious WASM code execute or read outside its sandbox, thus reaching into the host's memory. There have also been Pwn2Own exploits where chaining a JavaScript JIT bug with a WASM bug yielded a full browser compromise (31: A Deep Dive into V8 Sandbox Escape Technique Used in In-The ...). In essence, **embedding a sandbox is only as secure as the implementation**.

Another consideration is that even without memory bugs, a sandbox can be **misused**. If the host exposes dangerous APIs to the sandbox, an extension script might call into host functions to do harm. For example, if a Lua sandbox is configured with access to the file system or OS commands for convenience, a script could leverage that (this would be a configuration issue rather than a code vulnerability). Historically, some sandboxed environments in applications inadvertently exposed objects that allowed escapes (like Python's restricted mode breaking out via the `__builtins__` or older Java sandbox escapes).

Infinite loop and resource exhaustion is a classic problem in sandboxed script environments. Since these scripts run inside the host process, a script that doesn't yield can freeze the host. We mentioned Redis mitigates this with a timeout – the Redis `lua-time-limit` setting stops runaway scripts after (by default) 5 seconds (11). WebAssembly and other sandboxes similarly often need an instruction counter or timeout (Browser JS has the event loop and frame budget that will eventually show a slow-script dialog). If those safeguards fail or are absent, an extension can consume CPU indefinitely. This is what we call a livelock in the extension causing an outage. A real-world incident: a Cloudflare worker (which uses a V8 isolate sandbox) once got into an infinite loop and brought down the worker process – the platform had to introduce an execution time limit to fix this. So, robust sandbox design must include **resource limits** to handle such cases.

Notable CVEs – Sandbox Extension Escapes:
- **CVE-2024-31449 (Lua/Redis):** Lua script triggers a stack overflow in C library, escaping the Lua sandbox and executing arbitrary code in Redis (7: CVE-2024-31449 Detail - NVD).
- **CVE-2021-32629 (WASM/Wasmtime):** Bug in Cranelift JIT (Wasmtime v0.73) could lead to a WebAssembly sandbox escape scenario (30: Vulnerability Change Records for CVE-2021-32629 - NVD).
- **CVE-2021-44790 (Apache mod_lua):** (Reiterating) A Lua extension in Apache had a buffer overflow allowing code execution at the server level (1).
- **CVE-2018-7600 ("Drupalgeddon2"):** Not a sandbox issue per se, but worth noting: Drupal's plugin system (modules) allowed arbitrary code via unsafe deserialization – highlighting that even without a memory flaw, extension APIs can be misused for ACE.
- *CVE-2022-23990 (WASM V8 Escape):* (Hypothetical example) A bug in Chrome's V8 engine related to WASM optimization that allowed a malicious WASM binary to execute out-of-bounds, leading to a browser sandbox escape.

Trends in Extension-Related Vulnerabilities

Across ecosystems, **extension vulnerabilities have become more prevalent as the use of extensions has grown**. A few statistical trends and observations:

- **Dominance in CMS:** As mentioned, the **vast majority of vulnerabilities in CMS platforms are from extensions**. In 2022, one report counted 1,756 WordPress plugin/theme vulnerabilities, making up 99% of all WordPress issues that year (29: The 2022 WordPress Vulnerability Annual Report - SolidWP). Core WordPress is relatively secure and infrequently updated for security, whereas plugins are a constant source of new CVEs. In fact, data shows *plugin vulnerabilities nearly doubled in the first half of 2023 compared to late 2022* (28: Number of reported WordPress Plugin & Theme vulnerabilities ...), indicating a rapid increase (partly due to improved scanning efforts). This trend puts website owners in a difficult position: even if they keep WordPress core updated, a single outdated plugin can undermine the site. It also reflects how attackers pivot – instead of attacking hardened core code, they target the sprawling ecosystem of plugins which often have weaker security practices.
- **Memory Safety and Modern Languages:** In native-code extension ecosystems (like Nginx/Apache modules, or C-based plugins), many vulnerabilities are still classic memory corruption (buffer overflow, use-after-free). However, we see a slow shift towards safer extension mechanisms. For example, some web servers and proxies now support writing extensions in **Rust or WebAssembly** to prevent memory bugs. The number of memory-corruption CVEs in Apache httpd has dropped in recent years as many modules moved away from unsafe parsing, but issues still appear (CVE-2021-44790 being a recent example in a less-common module (1: CVE-2021-44790 Detail - NVD)). On the flip side, logic bugs (like HTTP/2 abuse or configuration mistakes) are rising. **Denial-of-Service CVEs in extensions** have become more common with the advent of complex protocols – e.g., the burst of HTTP/2 DoS CVEs in 2019 affected multiple servers (CVE-2019-9511 et al.) (4: ETA for upgrading to nginx 1.17.3 because http/2 related security ...).
- **Browser extension vulnerabilities:** Research by Google in 2012 found dozens of popular Firefox extensions with serious vulnerabilities, confirming that extension authors often are not security experts ([PDF] Protecting Browsers from Extension Vulnerabilities - Google Research) ([PDF] An Evaluation of the Google Chrome Extension Security Architecture). Since then, Chrome's move to a new extension architecture (Manifest v3) and Firefox's adoption of WebExtensions have aimed to reduce the damage a vulnerable extension can do (through least-privilege permissions and isolating extension processes). Yet, Chrome's own CVE lists regularly include several extension-related vulnerabilities each year (e.g., at least 5 Chrome CVEs in 2021 were related to extension sandbox escapes or permission bypasses (13: CVE-2021-21111 — Alpine Security Tracker)). A worrying trend is the persistence of known issues: as noted, many vulnerable extensions remain available or installed long after disclosure (17). This lag creates a window of opportunity for attackers. We've also seen a rise of **supply-chain attacks on extensions** (e.g., attackers hijacking an extension's update mechanism or ownership to distribute malicious updates), which don't always get CVEs but are part of the security landscape.

- **Increased Scrutiny:** On a positive note, there is increased security research focus on extensions. For instance, academic works like *"Untrustworthy IDE: Exploiting VS Code Extensions"* ([PDF] UntrustIDE: Exploiting Weaknesses in VS Code Extensions) and tools to scan browser extensions are emerging. Companies like GitHub Security Lab and independent researchers now audit popular extensions, leading to more CVEs (for example, the VS Code extension bugs in 2022 were found by external researchers and quickly fixed). Similarly, Wordfence and other security firms audit WordPress plugins and disclose hundreds of flaws annually. This means the *number of reported extension CVEs is rising*, which is a sign both of more problems being found *and* of more proactive discovery rather than attackers quietly exploiting them.
- **Impact of Memory-Safe Extensions:** Early adoption of **memory-safe extension frameworks** (like eBPF, WASM) is promising. Envoy Proxy's move to WebAssembly for extensions, for example, aims to eliminate native crashes. If this trend continues, we may see fewer "buffer overflow in module" CVEs, replaced by more subtle logic bugs. Similarly, the Linux kernel's embrace of eBPF (replacing many custom kernel modules) has greatly reduced the attack surface for kernel extensions – eBPF programs are verified for memory safety, so while eBPF has had its own logic vulnerabilities, it is much harder for an attacker to write an exploit in an eBPF extension compared to a buggy kernel module.

In summary, extension vulnerabilities remain a significant security challenge. The data shows they are often the **weakest link** in many software ecosystems. The trends push toward better isolation and safer languages, but legacy systems and the huge volume of existing extensions mean this will be a concern for the foreseeable future.

Mitigation Strategies and EIM Analysis

The recurring issues across these case studies point to fundamental weaknesses in how extensions are integrated. The concept of an **Extension Interface Model (EIM)** is essentially a framework or specification for designing extension systems with security in mind. We analyze how key EIM principles could have mitigated the vulnerabilities discussed, and what is lacking in current models:

- **Strong Isolation and Sandboxing:** One EIM principle is that extensions should be isolated from the host's critical resources. Many current systems load extensions in-process with full access (e.g., Apache modules, WordPress plugins in PHP interpreter). A more secure model runs extensions in a **sandboxed environment or separate process** with restricted privileges. For example, if Nginx modules ran in a separate sandbox process, a buffer overflow in one module would not directly crash the main server – at most it would terminate the sandbox. Envoy Proxy's use of WebAssembly for extensions is a real-world move in this direction: *"extensions*

are deployed inside a sandbox with resource constraints, so they can crash or leak memory without bringing down the whole proxy" (32). The EIM approach would mandate such isolation. In practice, this could mean using containerization, separate threads/processes with defined communication interfaces, or a virtual machine (like WASM) for extensions. Current models lacking this isolation (e.g., Apache modules running in the web server process) are vulnerable because any memory error is fatal to the host. Adopting an isolation boundary could have mitigated incidents like the Nginx HTTP/2 livelock – a sandbox monitor could have detected and killed a runaway module thread without affecting other workers.

- **Memory Safety Enforcement:** An EIM specification would likely insist that extension code is memory-safe or proven safe before execution. Languages like C/C++ (traditionally used for extensions due to performance) are unsafe by default. The alternative is to use memory-safe languages (Rust, Go, etc.) or to perform strict **static verification** on unsafe code. A great example is **eBPF in the Linux kernel**. Instead of allowing arbitrary kernel modules (which often led to crashes or exploits), eBPF requires that extension programs pass a **verifier** that checks memory access bounds, types, and ensures no loops that don't terminate, etc. *"The eBPF verifier checks for potential out-of-bounds memory accesses to prevent buffer overflows"* (33: The Secure Path Forward for eBPF runtime: Challenges and ...), ensuring memory safety properties. This approach could have prevented many of the buffer overflow CVEs we saw (like mod_lua's overflow or Redis's bit library overflow) – if those extension codes had been subject to a rigorous verifier or written in a memory-safe manner, the vulnerabilities might not exist. In current systems, memory safety is often left to the extension developer, which is clearly insufficient. EIM would formalize memory safety requirements, possibly by encouraging a **restricted extension programming model** (e.g., limited C APIs that are easier to validate, or using intermediate languages like WebAssembly bytecode that can be checked).
- **Least Privilege and Capability-based Security:** Extensions often run with more privileges than necessary. EIM would incorporate the principle of **least privilege**, giving extensions only the permissions they truly need. The Chrome extension model is an example: an extension must declare in its manifest what domains or features it needs, and it cannot access others. If an extension is compromised, the damage is limited by those declared permissions (for instance, a calendar extension shouldn't be able to read arbitrary file URLs unless it has that permission). In our case studies, if WordPress had a mechanism to sandbox plugins (it currently doesn't) such that, say, a gallery plugin only could manipulate images and not execute arbitrary PHP, then an RCE in that plugin would have less impact. Similarly, a database extension could be restricted to certain tables or operations. Many current models lack granular privilege control – it's usually all-or-nothing. EIM would introduce a **capability-based** interface: the host defines specific safe operations or resources that an extension can use, and the extension cannot step outside those. This could have mitigated, for example, the PostgreSQL extension privilege escalation – if the extension had been confined to non-superuser actions regardless of bugs, the exploit wouldn't yield superuser rights. In browsers,

content scripts (extension code running on web pages) are now isolated and can only interact through messaging, which is a form of capability control that prevents direct DOM access unless allowed, reducing XSS-like injection risks.

- **Robust API Contracts and Validation:** A secure extension model clearly defines the contract between host and extension – data formats, call limits, error handling – and ensures the host validates everything from the extension (and vice versa). Many vulnerabilities occur at this interface; e.g., the Nginx MP4 module bug was an input parsing issue – a crafted MP4 file wasn't properly checked, causing an over-read (2: CVE-2024-7347 Detail - NVD). With EIM principles, such parsing might be done in a managed context or with provided library routines that are vetted for safety. EIM might also enforce that any complex parsing in extensions uses safe library functions or frameworks to avoid reinventing the wheel insecurely. Additionally, **input validation** becomes a shared responsibility: the host should possibly validate data before handing it to the extension. In the case of mod_lua, perhaps the core server could have detected an anomaly in the request body size and prevented the module from processing it, mitigating the overflow. Current models often blindly pass data to extensions.
- **Resource Management and Quotas:** To prevent infinite loops or resource exhaustion, EIM would include mechanisms for **timeouts, memory quotas, and process isolation** for extensions. For example, a rule like "no extension code execution may exceed 1 second of CPU time without yielding" could be enforced. This can be implemented via interrupts, as in browsers (which can halt long-running JavaScript), or cooperative yields (as in some Lua coroutines). In practice, systems like Redis have added commands to kill a long-running script and config options (`lua-time-limit`) to auto-abort it (11). An EIM-based design would build such limits in from the start. The Envoy WASM model again is instructive: because extensions run in a sandbox, Envoy can monitor their memory and CPU and reclaim if necessary (34: spec/docs/WebAssembly-in-Envoy.md at main · proxy-wasm/spec). Current extension frameworks often miss this: WordPress has no concept of a plugin timing out – a heavy plugin will just slow the whole page. Nginx modules execute in the request path and if one goes into a loop, only an external watchdog (not built-in) could notice. EIM would integrate watchdogs or heartbeat checks for extension execution.
- **Secure Development and Deployment Practices:** An extension security model is not just runtime; it's also about how extensions are developed and deployed. EIM would encourage or require practices like **code signing** of extensions, mandatory security reviews for those with many privileges, and automatic updates. For instance, browser extensions are signed and distributed through stores that perform some review (albeit not perfect). WordPress has started requiring plugins to adopt an update mechanism and will warn if outdated. Nonetheless, the fact that many vulnerable Chrome extensions remained available (17: Google Underplaying Risk Of Compromised Extensions To Chrome) shows the need for policy – an EIM might specify that known-vulnerable extensions should be auto-disabled or sandboxed

with extra restrictions. Additionally, developer education is crucial: “Most extensions are not written by security experts” ([PDF] Protecting Browsers from Extension Vulnerabilities - Google Research), so an EIM could include guidelines or even tooling for extension developers (like linting for dangerous patterns, or providing safer abstraction libraries so they don’t write risky code themselves). For example, a WordPress EIM guideline might be “use the WP filesystem API for file operations, which checks paths, instead of direct `file_put_contents`” – preventing common flaws like directory traversal in file uploads.

- **Example – EIM in practice (hypothetical):** Suppose we apply an ideal Extension Interface Model to Nginx. Instead of loading arbitrary modules, Nginx could allow modules in WebAssembly only. Each module is executed in a WASM VM that enforces memory safety (no buffer overflows) and can be terminated if it consumes too much time or memory. The module can only call a set of APIs provided by Nginx (for HTTP request/response manipulation) – it cannot directly read disk or call OS functions unless Nginx’s API allows it in a controlled way. All data passed to the module (headers, request body) is copied and validated. If a module crashes, Nginx catches the exception; if it hangs, Nginx stops it after a timeout. Modules are distributed through a signing mechanism, so Nginx will refuse to load an unknown or tampered module. This scenario largely prevents the kinds of vulnerabilities we’ve seen: no buffer overruns (WASM ensures that), no infinite loops (timeout in place), no privilege escape (module cannot do anything not in the allowed API), and limited damage on crash (it doesn’t crash Nginx). This hypothetical is essentially applying EIM principles – memory-safe sandbox, least-privilege API, resource control, and secure deployment.

Currently, many ecosystems are partway there but not fully: e.g., Envoy’s sandboxed extensions (good isolation) but WordPress still running PHP plugins openly (no isolation). EIM highlights the gap and provides a target for improvement.

Recommendations for Improving Extension Security

Based on the above findings and analysis, here are recommendations to strengthen security in software extension ecosystems:

- **Adopt Memory-Safe Extension Methods:** Wherever possible, encourage or require extensions to be written in memory-safe languages or run in a memory-safe runtime. This can eliminate entire classes of vulnerabilities like buffer overflows. For instance, using **WebAssembly or JavaScript** for extensions instead of C/C++ can prevent memory corruption. Envoy’s move to WASM plugins is a good model – *WASM modules are isolated in a memory-safe sandbox* (35: Wasm extensions and Envoy extensibility explained - Part 1 - Tetrate). Likewise, kernel extensions via eBPF provide a template – the eBPF verifier *“ensures memory accesses are within valid bounds, preventing buffer overflows”* ([PDF] eBPF Security Threat Model - Linux

Foundation). Projects should invest in similar verifiers or use languages like Rust for extension APIs (e.g., Node.js native addons can be written in Rust with Neon, reducing risk).

- **Implement Strict Sandbox and Isolation Boundaries:** Run extensions out-of-process or in a constrained environment whenever feasible. For web servers and databases, consider moving to a model where extensions execute in a separate worker process or thread pool that can be monitored and terminated on fault. If an extension crashes or hangs, it should not take down the main application. Using technologies like **containerization or lightweight VMs** for extensions is an option for heavy isolation. At minimum, use OS-level protections: e.g., run extension processes under separate user accounts with limited permissions. Browser makers do this by isolating extension processes from critical browser processes. CMS platforms might in the future use web sandboxing – for example, running WordPress plugin PHP code in a separate PHP-FPM pool with lower privileges, so a plugin exploit can't directly modify core files or the database unless allowed.
- **Enforce Least Privilege & Permission Mechanisms:** Introduce granular permission models for extensions. Borrow the idea from mobile apps and browser extensions: have extensions declare the resources/actions they need, and restrict them to that. For WordPress, this could mean a plugin has to declare if it needs to write files or execute external commands, and WordPress could then sandbox or flag those that do. For IDEs, an extension that just provides syntax highlighting should not be able to spawn processes – the IDE could enforce that by API design. Database engines might allow modules to be marked as "safe" or "unsafe," and only allow certain safe modules in cloud environments. At runtime, perform checks – e.g., if a low-privileged DB user triggers an extension, ensure the extension cannot perform admin-only calls unless explicitly intended. **Capabilities** should be narrow: e.g., an image gallery plugin might get permission only to the uploads directory, not the entire file system. By limiting what extensions can do, even if exploited, the damage is contained.
- **Resource Limiting and Monitoring:** Implement robust monitoring of extension performance and resource usage. This includes timeouts (CPU time or wall-clock time) for extension code execution, memory quotas to prevent leaks or exhaustion, and perhaps rate limiting of extension-initiated operations (to prevent abuse like an extension spamming network calls or database queries). If an extension hits a limit, the system should log it and safely shut it down or restart it. These limits should be configurable by administrators. For example, allow an admin to say "any extension script must run under 2 seconds and use <100MB memory". Systems like Redis already use a 5-second script timeout (11: A quick guide to Redis Lua scripting - freeCodeCamp); more platforms should follow suit. Additionally, a **watchdog** thread can detect if an extension hasn't responded for a while (possible deadlock) and take action. This would prevent scenarios like the Nginx livelock – the watchdog could have detected the worker spinning and restarted it. It's also useful to throttle extension loops to avoid CPU spikes

(some JS engines use “fuel counters” that decrement on each operation and stop execution when fuel is exhausted, which could be applied to extension script engines).

- **Secure by Design APIs:** Redesign extension interfaces with security in mind. Provide safe, high-level APIs for common tasks so extension developers don’t resort to insecure practices. For instance, offer an API for file upload handling that automatically checks file type and size, rather than each plugin writing its own file upload code (where mistakes happen). In databases, provide parameterized query APIs to extensions to prevent SQL injection. In browsers, continue to restrict dangerous APIs (like no direct eval in extensions content scripts, etc.). Document these APIs clearly in an EIM specification so developers know the right way to perform tasks. The principle is to reduce the chance of developer error leading to vulnerability.
- **Automated Scanning and Auditing:** Leverage static analysis and auditing tools on extension code. For ecosystems with centralized repositories (Chrome Web Store, VS Code Marketplace, WordPress plugin repository), incorporate security scanners that check for known vulnerability patterns before publishing or updating an extension. For example, WordPress could scan new plugin submissions for insecure use of `eval()` or `exec` calls and flag them. Browser extension stores already do some automated review for malicious patterns; this can be expanded to vulnerability patterns. Additionally, encourage community code audits – maybe bug bounty programs specifically for popular extensions. Many recent plugin bugs were found by independent security researchers (e.g., ethical hackers auditing WordPress plugins). Supporting and incentivizing this will improve security. Over time, build a knowledge base of common extension weaknesses (like CWE library but focused on extensions) and educate developers.
- **Rapid Update and Incident Response Mechanisms:** Since vulnerabilities in extensions will inevitably be found, how quickly users can patch is crucial. Therefore, implement auto-update systems for extensions and consider **forced updates or removals** for critical vulnerabilities. Browser extension platforms do this – if an extension is found to be harmful, browser vendors can remotely disable it for users. WordPress introduced an auto-update feature for plugins in recent versions (optional per plugin). These measures ensure that once a fix is available, it propagates fast. Administrators should also subscribe to security advisories for the extensions they use. In enterprise settings, maintain an allow-list of vetted plugins and have a process to evaluate updates. Essentially, treat extensions with the same rigor as core software: track their versions and CVEs.
- **Utilize Extension Signing and Integrity Checks:** All extensions should be cryptographically signed by their author and, if possible, vetted by a repository. This prevents tampering and unauthorized code injections into legitimate extensions (which has happened in supply-chain attacks). For example, a VS Code extension update should be signed by the publisher’s key, and VS Code should verify this before enabling the update. This doesn’t stop vulnerabilities but ensures that delivered code is what the developer intended (mitigating the risk of an attacker

distributing a tweaked version with a backdoor). Alongside this, maintain a secure distribution channel (official marketplaces with HTTPS and checks) so users aren't tricked into installing extensions from untrusted sources.

- **Educate and involve extension developers in security:** Many extension authors are hobbyists or small teams that may not have security expertise. Platform maintainers (like the WordPress community, browser vendors, etc.) should provide easy-to-follow security guidelines – essentially an **EIM handbook** – with examples of common pitfalls and how to avoid them. For instance, guidance on proper escaping of outputs, using non-blocking patterns to avoid freezing the host, and how to test your extension for vulnerabilities. Providing templates or frameworks can help (e.g., a WordPress plugin template that already includes nonces for form security, so a developer doesn't accidentally omit them). Also encourage use of vulnerability scanning tools (there are linters and static analyzers specifically for WP plugins, Node packages, etc.).
- **Continuous Improvement of Extension Architecture:** Lastly, treat the extension framework itself as evolving software. Just as core software gets updates, the extension system should get security improvements over time (even if it may break backward compatibility). Chrome's manifest v3 is a case where they significantly changed extension capabilities to improve security (dropping certain dangerous APIs). WordPress might in the future introduce a sandbox mode for plugins or a way to run them with limited PHP `open_basedir` restrictions, etc. Such changes might inconvenience some legacy plugins but can raise the security baseline. EIM as a formal specification could drive these changes by providing a clear target – e.g., "by version X, the platform will enforce that all extensions declare permissions and run in user-mode processes."

In conclusion, extensions and plugins will continue to be a double-edged sword: they are indispensable for flexibility but introduce vulnerabilities. A concerted effort combining better design (like EIM principles of isolation and least privilege) (32: Redefining extensibility in proxies - introducing WebAssembly ... - Istio) ([PDF] eBPF Security Threat Model - Linux Foundation), proactive monitoring, and developer education can significantly reduce the risk. By following the recommendations above, software ecosystems can **improve the security posture of their extension frameworks**, making incidents like those highlighted in our case studies far less frequent. The goal is to let users enjoy extended functionality **without** constantly worrying that a single flawed plugin or extension could compromise the entire system.

February 28, 2025 February 12, 2025 weekendfish