AI Web Crawler Security White Paper

Author: Guancheng Li、Zheng Wang @ Tencent Xuanwu Lab

Abstract

AI application architectures are evolving from simple offline LLM conversations to online, internet-connected Agents capable of invoking tools and autonomously decomposing tasks. Whether handling basic tasks like "search the web and summarize the answer" or executing complex automated task chains such as "search for company information, then query stock prices, and finally provide investment recommendations," browsers are increasingly integrated into server-side systems to enable real-time internet connectivity and content extraction.

However, relocating browsers—which were originally designed to run on the client side—to server-side environments introduces an architectural mismatch that creates significant security risks:

Blurred Trust Boundaries: Browsers are software that parses untrusted external code (such as JavaScript and DOM) and frequently contains high-severity vulnerabilities. When deployed on servers that may have direct access to enterprise internal networks and critical business systems, they become prime entry points for external attackers seeking to penetrate internal networks.

Weakened Security Posture: Server-side browsers often suffer from delayed patch updates and overly permissive runtime privileges. Some vendors disable native sandbox mechanisms to ensure compatibility, resulting in security protections that are often weaker than those of typical client-side browsers.

Greater Attack Impact: Once a browser is compromised, attackers can not only steal task data or tamper with returned results to "poison" downstream decision-making processes—they can also leverage shared architectures to laterally affect other products and users, or use the compromised browser as a pivot point to move laterally and attack other core internal systems.

Our research published at Black Hat also confirmed that the crawlers of multiple AI products have remote code execution risks. Given that server-side browsers have become critical risk points in AI server infrastructure, and that the industry currently lacks systematic protection standards, this white paper aims to fill that gap. We provide a detailed analysis of the risk characteristics in this scenario and propose a defense framework centered on "static attack surface reduction + dynamic behavior isolation." This framework is designed to help enterprise security leaders and technical teams achieve secure deployment and operation of server-side browsers. We have open-sourced this solution on GitHub, hoping to help the industry improve the overall security posture of server-side browsers.

Code Repository: https://github.com/XuanwuLab/SEChrome

1. The Changing Threat Landscape for Browsers in AI Systems

When you launch a browser instance, you are not starting a simple web browsing tool—you are launching a "micro operating system" composed of the V8 engine, WebRTC components, a PDF reader, dozens of audio/video codecs, and a complex rendering engine. A vulnerability in any single component could lead to remote code execution, which is why browsers have consistently ranked among software with the highest number of severe vulnerabilities and the highest proportion of exploitable vulnerabilities.

In the AI era, browsers have transformed from user gateways to the web into foundational components supporting AI business operations. We are placing a browser—whose complexity far exceeds that of typical server-side components and which frequently contains vulnerabilities—onto high-value servers. This change is not merely a shift in deployment location; it represents a fundamental role change that ultimately alters the threat landscape for browsers on the server side.

1.1 Changes in the Attack Landscape

From an attacker's perspective, the transformation of the browser's role introduces deep structural risks. This change can be summarized across four dimensions:

Failure of the Patch + Sandbox Defense Model: Traditional browser security relies heavily on "automatic updates" and "sandbox isolation." However, on the server side, some developers disable automatic updates to maintain environment consistency, while others disable sandboxes to accommodate container architectures. This deviation in operational environments directly undermines traditional defense systems, making N-day vulnerabilities a persistent threat.

Expanded Attack Impact: Traditional browsers affect only individual endpoints. On the server side, browsers are shared components serving multiple users. Once attackers break through, they can affect other users through the shared environment—for example, by manipulating search results for multiple users simultaneously.

More Severe Attack Consequences: Traditional browsers serve humans, but now they serve AI. Attackers' goals are no longer limited to gaining access; they can also "poison" AI knowledge inputs by tampering with web content. In automated "search-decide-execute" chains, such data-layer attacks can directly manipulate AI's final decisions. Furthermore, without proper isolation in server-side environments, attackers can use compromised browsers as pivot points to move laterally into core internal networks, amplifying the damage.

Increased Attacker Motivation: Due to the factors above, server-side browsers have become shortcuts to enterprise core data and business logic. Compared to attacking individual users, compromising server-side browsers offers extremely high returns, motivating attackers to invest more resources (such as purchasing 0-day exploits) to specifically breach these

defenses.

1.2 Changes in the Defense Landscape

From a defensive perspective, this role change also creates a mismatch in defense strategies. The core principle of server-side security is the principle of least privilege—components should possess only the minimum capabilities required to complete specific tasks. However, existing browsers are designed for web compatibility and enable WebGL, WebRTC, and various functional interfaces by default. This "default-open" approach contradicts the strict access controls required on the server side, resulting in an expanded attack surface.

Additionally, modern server-side security typically requires high configurability and operability—security policies should be definable and auditable like infrastructure code. Unfortunately, browsers often operate as opaque black boxes, lacking standardized server-side configuration interfaces and structured security audit logs. This makes it difficult for operations teams to constrain browser behavior boundaries as they would configure Nginx, and prevents them from gaining sufficient visibility when attacks occur.

Therefore, we must reassess the risks of server-side browsers and establish a new browser security defense practice that meets the requirements of server-side security operations.

2. Security Risk Assessment for AI Server-Side Browsers

This section provides a systematic reassessment of server-side browser security risks.

2.1 Risk 1: Delayed Patch Updates and Incorrect Sandbox Configuration

Client-side browsers rely on mature security mechanisms: automatic updates with rapid patch deployment, native sandboxes, and multi-process isolation. Chrome on desktop typically pushes updates to most users within days of vulnerability disclosure.

Server-side browsers operate differently. After analyzing AI server-side browser components from multiple vendors, we observed two widespread issues:

Sandbox Configuration Issues: Chrome's user-space sandbox relies on Linux namespaces and seccomp mechanisms. In container environments, if runtime configurations are improper (such as not granting necessary Linux Capabilities or not correctly setting seccomp policies), Chrome will fail to start. To quickly resolve startup failures, many teams add the `--no-sandbox` parameter to bypass restrictions, but this actually disables the browser's most critical security boundary, creating serious risks. Additionally, some container platforms do not grant the system permissions required for Chrome's sandbox by default, as a precaution against

container escape vulnerabilities.

Slow Version Updates: Some teams prefer using older, long-validated image versions due to concerns that version updates might cause changes in page rendering behavior or compatibility issues.

According to Google Project Zero statistics, Chrome has fixed over 1,600 security vulnerabilities between 2020 and 2025, including multiple zero-day vulnerabilities exploited in the wild. CVE database records show that Chrome's V8 engine, Blink rendering engine, and WebAssembly runtime are high-frequency vulnerability areas—these same components are used in server-side browsers.

Disabling sandboxes and failing to update versions promptly expose browsers to N-day vulnerability threats, leading to remote code execution risks in AI server-side browsers.

2.2 Risk 2: Multi-User, Multi-Product Shared Architecture Amplifies Vulnerability Impact

In client-side scenarios, a browser instance serves only one user, naturally limiting the scope of any attack.

In server-side AI systems, browsers typically run as resource pools: simultaneously serving multiple AI products, being reused by multiple task queues, supporting query requests from multiple users, and running continuously in long-lived containers.

Under this architecture, if a malicious page triggers and successfully exploits a vulnerability, attackers may interfere with multiple tasks being processed or awaiting processing, poison LLM data inputs to control behavior, and create cross-user or even cross-product impacts.

2.3 Risk 3: Inadequate Internal Network Isolation Amplifies Attack Damage

In client-side scenarios, browsers run on users' personal computers, which are typically network-isolated from enterprise production systems.

In server-side scenarios, browsers are part of business logic and may be deployed in the same cluster or network segment as business services. Without dedicated network isolation policies, browsers and internal network resources (databases, task scheduling systems, model inference nodes, internal APIs) may be mutually accessible.

Once a browser is compromised, attackers will use the node as a foothold: scanning internal services, reading credentials from environment variables, and accessing metadata services to obtain cloud platform permissions. The impact extends far beyond the browser itself.

2.4 Risk 4: How Web Content Is Used Amplifies Attack Damage

Server-side browsers are not just data retrieval tools—they are the "eyes" and "ears" of AI

models. Attackers can embed Prompt Injection payloads or malicious instructions in web pages. When browsers fetch page content and feed it to downstream models, this may induce AI to output incorrect results, leak sensitive information, or even execute unintended automated operations. Such attacks transcend traditional software vulnerability layers and can directly poison AI decision logic.

2.5 Risk 5: High Attack Returns Increase Attacker Willingness to Use High-Value Exploits

In AI scenarios, the potential value of server-side browsers has significantly increased. Client-side attacks typically only obtain data and permissions at the individual endpoint level, while server-side browsers are deployed closer to business systems. Once the browser is compromised, attackers can further access internal interfaces, configuration information, business data, or other services. Therefore, attackers are likely willing to invest in more sophisticated attack methods, such as using 0-day exploits.

Thus, "we're using the latest browser version, so we don't have security issues" is a dangerous misconception. The latest version mitigates N-day vulnerabilities but cannot eliminate 0-day vulnerabilities—this is why we emphasize introducing "defense in depth": assuming that unknown vulnerabilities may always exist within browsers and designing "buffer zones" in advance for when defenses are breached.

3. Attack Paths and Real-World Cases

To effectively defend against server-side browser threats, we must first understand the penetration path from an attacker's perspective. Through post-incident analysis of real-world cases, we have mapped out a typical kill chain targeting server-side browsers and selected four representative real-world cases for analysis.

3.1 Server-Side Browser Kill Chain

Attacks targeting server-side browsers typically follow five stages:

Reconnaissance: Attackers first identify functionality that can trigger server-side HTTP requests. Beyond obvious "web scraping" and "URL preview" features, hidden entry points must also be considered, such as image loading in Markdown rendering, PDF generation services, cache warming interfaces, or backend crawlers.

Evasion: To bypass URL allowlists or script filters deployed by systems, attackers use techniques such as 302 redirects, DNS rebinding, protocol parsing discrepancies, or specially crafted HTML structures to evade checks and redirect requests to attacker-controlled websites.

Fingerprinting: HTTP User-Agent headers are easily spoofed, so attackers do not trust them.

Instead, they determine the specific browser kernel version by detecting browser support for specific APIs, CSS properties, or JavaScript syntax features.

Exploitation: Once the version is confirmed, attackers search vulnerability databases for 0-day/N-day vulnerabilities affecting that version and deploy corresponding exploit code. When the server-side browser parses the malicious page, the vulnerability is triggered and attackers gain code execution privileges.

Post-Exploitation: If the browser sandbox is enabled, attackers must use sandbox escape vulnerabilities to break out. If the sandbox is disabled or misconfigured, attackers directly gain shell access on the host, enabling them to probe internal network topology, read sensitive configuration files (such as cloud service access keys), or use the foothold to attack internal databases and other microservices.

3.2 Four Representative Cases

We conducted testing on several products that integrate server-side browsers and found that multiple products had remote code execution (RCE) security risks outside the sandbox. The affected products collectively serve over one billion server-side users, covering products from multiple leading technology companies.

We present four representative cases to provide a comprehensive understanding of this attack surface.

Case 1: Bypassing Allowlists via URL Redirects to Achieve Remote Code Execution

Reconnaissance: An AI search product implemented an access allowlist policy, permitting the browser to access only a preset list of trusted websites.

Evasion: We discovered that certain large search sites on the allowlist create intermediate redirect links for indexed websites, such as xx.com/link?url=. We constructed a request to access an allowlisted site, but with URL parameters pointing to our controlled server. The system verified the initial address was on the allowlist and permitted the request. After the browser opened the trusted site, it read the URL parameters and automatically redirected to our malicious page.

Fingerprinting: After successfully bypassing the allowlist via redirect, our malicious page ran in the server-side browser. We detected that the backend browser was using an older version, Chrome/120.

Exploitation: With the version confirmed, we exploited a publicly disclosed N-day vulnerability for that version combined with a V8 sandbox bypass vulnerability. Since the service did not enable the browser sandbox, we successfully gained control of the server after triggering the vulnerability.

Case 2: Chaining Multiple Browser Features to Achieve Remote Code Execution

Reconnaissance: An AI product had three independent features: an AI reading feature using

the latest browser version, a sharing feature that generates publicly accessible same-domain links, and a screenshot feature with URL allowlist filtering.

Evasion: We constructed an attack path traversing all three features: first, we used the "AI reading" feature to analyze a webpage containing malicious code, and the system embedded the malicious content in the response page; then we "shared" the response as a public link; finally, we requested the "screenshot feature" to render that shared page. Since the shared link belonged to the site's own domain, it satisfied the screenshot service's allowlist requirements, and the malicious code was successfully loaded.

Fingerprinting: When the screenshot service's browser opened the shared page, the embedded attacker code was executed. We detected that although the AI reading feature used a newer browser, the screenshot service's browser version was older.

Exploitation: For that older version, we tested corresponding exploit code. After achieving shellcode execution capability, we attempted to read the /proc/self/maps file and discovered it could not be opened, indirectly indicating that the screenshot service's browser had the sandbox enabled. Ultimately, we exploited SSRF to access cloud metadata, demonstrating the severity of the risk.

Case 3: Bypassing Script Execution Restrictions to Achieve Remote Code Execution

Reconnaissance: An AI product had a feature to access URLs specified in conversations. This feature filtered all <script> tags in web pages, intending to prevent JavaScript execution.

Evasion: JavaScript execution is not limited to <script> tags—iframe embedding and event handlers can also trigger it. We constructed a page containing no <script> tags whatsoever, triggering code execution via <img src=x onerror="malicious code">. The onerror event handler for image load failures successfully bypassed the script filter.

Fingerprinting: After the malicious code executed, we detected that the backend browser was running an older version, Chrome/121.

Exploitation: Based on the detected version information, we searched for and exploited a publicly disclosed N-day vulnerability along with two V8 sandbox bypass vulnerabilities. When using the first sandbox vulnerability, we discovered the target system had enabled pkey-based runtime write protection for WebAssembly, preventing shellcode from being written to WebAssembly's readable-writable-executable code region. Therefore, we used the first sandbox bypass vulnerability to leak addresses, then used the second sandbox bypass vulnerability to perform JIT spray to hijack program execution flow. Ultimately, combining these three vulnerabilities, we successfully gained control of the server.

Case 4: Discovering Hidden Backend Browser Entry Points to Achieve Remote Code Execution

Reconnaissance: An AI search product's frontend functionality rejected access to our test URL. However, we kept our test server running, and three days later discovered access records from that product in our logs. Investigation revealed the product had a hidden backend indexing system that batch-fetched URLs users had queried during idle periods.

Evasion: This backend crawler was invisible to users and not documented in any product

materials. Attackers did not need to directly bypass frontend real-time checks; instead, they could leave historical records and wait for backend asynchronous fetching to enter the system.

Fingerprinting: When the backend crawler accessed our server, we confirmed it was running an older browser kernel version, Chrome/122.

Exploitation: For this older kernel version, we deployed corresponding N-day exploit code. Since this backend browser did not have the sandbox enabled, we ultimately successfully executed shellcode and gained control of the server hosting the browser.

These cases expose a common problem: limitations in defensive thinking.

Enterprises often believe that adding URL allowlists provides complete protection, overlooking the fact that allowlists are easily bypassed via redirects and blocklists are easily evaded via obfuscation. More seriously, security visibility often misses "invisible" browser entry points such as screenshot services and backend crawlers. Once attackers bypass the first line of defense, the combination of "outdated browser versions" and "disabled sandboxes" often leads to systems being directly compromised by N-day exploits. Notably, even with updates maintained and sandboxes enabled, attackers may still choose to use 0-day exploits against high-value systems.

Therefore, addressing server-side browser threats cannot rely solely on ad-hoc protection schemes designed by developers—it requires a systematic defense-in-depth approach.

4. AI Server-Side Browser Defense Strategy: From Passive Protection to Active Operations

Server-side browsers require a different defense approach than client-side browsers. Addressing the unique characteristics of server-side environments, we break down our defense strategy into four key dimensions:

More Proactive Defense Strategies: Client-side browsers typically rely on vendors' rapid security updates and built-in sandbox mechanisms to counter threats. However, server-side environments often have longer patch update cycles. Additionally, due to container or virtualization environment constraints, browser sandboxes are frequently disabled due to compatibility issues. This means we cannot simply rely on client-side "automatic defense" mechanisms—we must build more proactive defenses.

Attack Surface Reduction and Least Privilege Principle: Client-side browsers enable WebGL, WebRTC, and various sensor interfaces by default for web ecosystem compatibility. In server-side scenarios, we should follow the Least Privilege principle: components should possess only the minimum capabilities required to complete specific tasks. Unnecessary browser features must be firmly disabled to prevent non-essential functional components from being exploited by attackers.

Configurable and Operable: In client-side scenarios, browser protection mechanisms are typically a non-configurable "black box" for users. But server-side security typically requires high configurability and operability. Server-side solutions must allow security teams to customize policies according to business needs rather than passively accepting preset security

configurations, thereby achieving precise control over security posture.

"Assume Vulnerabilities Exist" Defense Philosophy: Since server-side browsers carry extremely high attack value, we cannot rule out the possibility of attackers using 0-day exploits to break through defenses. Therefore, we need to establish a defense system that "assumes vulnerabilities exist." The focus of defense is not only preventing vulnerability exploitation but also minimizing damage when defenses are breached.

Considering all these factors, we designed a protection solution for AI server-side browsers: reducing exploitable attack surface through kernel streamlining, and strictly limiting the impact scope after vulnerability exploitation through configurable file access controls and process execution allowlist mechanisms.

4.1 Attack Surface Reduction

Server-side browser tasks are typically highly specific (such as screenshots or text extraction). Many complex components designed to enrich human experience are not only redundant on the server side but are also high-risk attack entry points.

Disabling Unnecessary Functional Modules

WebGL and GPU-accelerated rendering are typically unnecessary on the server side and can be disabled via the --disable-gpu and --disable-webgl parameters.

WebRTC real-time communication functionality is typically useless in server-side scenarios and may leak internal network IP addresses; it can be disabled via the --disable-webrtc parameter.

PDF plugins should also be disabled if there is no explicit requirement.

The V8 engine's JIT compiler is a high-frequency vulnerability area; if performance requirements allow, JIT can be disabled via the --jitless parameter, at the cost of reduced JavaScript execution performance—this needs to be weighed against business scenario requirements. Server-side browsers should not load any extensions or plugins.

According to CVE data analysis, approximately 16% of Chrome's high-severity vulnerabilities between 2023 and 2025 were related to WebGL/GPU, and approximately 23% were related to V8 JIT. Directly disabling these modules via startup parameters can eliminate nearly 40% of vulnerabilities at the source.

Configuration Recommendations

Below is an example of security-oriented Chrome startup parameters; security teams can adjust according to business needs:

```
# Basic mode settings
--headless=new
--no-first-run
--disable-crashpad
--disable-crash-reporter

# Attack surface reduction (core)
--disable-gpu                    # Disable GPU hardware acceleration (high-risk
area)
--disable-webgl                  # Disable WebGL (high-risk area)
--disable-webrtc                 # Disable WebRTC (prevent IP leaks and P2P
attacks)
--disable-extensions             # Prohibit loading any extensions
--disable-plugins                # Disable PDF and other plugins

# V8 engine hardening (optional, depending on performance requirements)
# Disable JIT compiler—reduces JS performance but immunizes against most V8
vulnerabilities
--js-flags="--jitless"
# Or disable only some optimization pipelines
--js-flags="--no-turbofan,--no-maglev"
```

If performance allows, adding `--jitless` can further reduce attack surface.

Note on Sandboxes

Avoid using the `--no-sandbox` parameter.

Some teams launching Chrome in containers often disable the sandbox directly to resolve sandbox permission errors. This is an extremely dangerous practice. The correct approach is to fix container configurations (such as adding necessary seccomp profiles or using `--cap-add SYS_ADMIN`) rather than dismantling the last line of defense. Given the scarcity and high cost of sandbox escape vulnerabilities, retaining the native sandbox provides extremely high defensive value.

4.2 Limiting Impact After Vulnerability Exploitation

Even after reducing attack surface, given that browsers are a major source of severe vulnerabilities, we must assume browsers will be compromised. The goal of defense in depth is: even if the browser is compromised, attackers cannot cause actual harm—they cannot

access internal networks, they cannot steal data.

We divide defense in depth into two defensive layers: the "infrastructure layer" and the "browser runtime layer." Infrastructure layer isolation refers to solutions implemented using isolation mechanisms provided by containers, networks, and operating systems. Browser runtime layer isolation restricts what actions a browser can perform after being compromised by monitoring and auditing browser process behavior.

First Line of Defense: Infrastructure Layer Isolation (Coarse-Grained Boundaries)

Network Isolation: Browser containers should be deployed in isolated network zones, permitted only to access the internet, with internal network resource access prohibited. Implementation methods include using separate VPCs or subnets and restricting outbound traffic through network policies (Kubernetes NetworkPolicy or cloud platform security groups). Validation involves attempting to access internal services (such as metadata services, internal databases, other business APIs) from within the browser container and confirming they are unreachable.

Filesystem Isolation: Browsers should run in read-only mode, permitted to write only to specific temporary directories, preventing attackers from achieving persistence by writing scheduled tasks, SSH keys, or other configuration files. Specific measures include using a read-only root filesystem (readOnlyRootFilesystem: true) and mounting temporary directories as tmpfs (in-memory filesystem, cleared after container restart), ensuring the browser process cannot access sensitive configuration files.

Instance Isolation: Ideally, each task uses an independent browser instance, which is destroyed after task completion. This prevents one malicious page from affecting subsequent tasks and prevents information leakage between tasks. If performance requirements do not permit complete instance isolation, at minimum, independent instances should be used between different users or tasks of different security levels.

Second Line of Defense: Browser Process Behavior Control (Fine-Grained Control)

The infrastructure layer provides external barriers, while runtime isolation plants "monitoring probes" inside browser processes. By monitoring and auditing browser process system calls (syscalls), we restrict its behavioral capabilities. Even if attackers successfully launch RCE attacks against the browser, system call-level behavior control can still limit vulnerability impact to a controllable scope. In Chapter 5, we will detail SEChrome, a protection solution designed based on this concept.

Coordination of Two Isolation Layers

Infrastructure layer isolation and browser layer isolation complement each other. The infrastructure layer provides coarse-grained network and resource boundaries as the outer layer of defense in depth; the browser layer provides fine-grained behavior control, intercepting attacks at the initial point of occurrence.

Even if browser layer isolation is breached (for example, through a kernel vulnerability),

infrastructure layer isolation remains effective; even if infrastructure layer configuration has gaps, browser layer isolation can block most attacks. Combined, attackers must simultaneously breach both lines of defense to cause substantial harm.

4.3 Other Risk Points Requiring Attention

Beyond browser vulnerability exploitation, the following risks in server-side browser scenarios also merit attention:

Credential and Session Management: If browsers need to log into certain websites, login credentials and cookies require proper cross-user isolation.

Resource Exhaustion Attacks: Malicious pages may exhaust resources through infinite loops or massive memory allocations, causing denial of service. CPU and memory limits should be set for browser containers, along with page load timeouts.

Supply Chain Security: Tools such as Puppeteer, Playwright, and Selenium also require security updates. These dependencies should be included in vulnerability scanning and update processes.

5. SEChrome: Xuanwu's Browser Protection Solution

We designed SEChrome, a low-cost, high-value browser runtime isolation protection solution. This solution monitors system calls to continuously audit browser behaviors including file access, process creation, and network requests, enabling real-time detection and blocking of browser attacks (including 0-day exploits). Evaluation results demonstrate that SEChrome provides broad defensive effectiveness against multiple browser attack scenarios. We have open-sourced this solution on GitHub (https://github.com/XuanwuLab/SEChrome), hoping to contribute to AI ecosystem security.

5.1 Solution Features

Dual-Engine Isolation

SEChrome employs a seccomp + ptrace dual-engine architecture to balance security, performance, and environment compatibility.

Seccomp uses Linux kernel BPF functionality to filter system calls directly in kernel space, with minimal performance overhead (<1%), suitable for high-concurrency production environments. However, it requires containers to have CAP_SYS_ADMIN privileges and may be restricted in some managed container services (such as AWS Fargate).

Ptrace uses the Linux process tracing mechanism to intercept and audit system calls in user space. It requires no special privileges and is compatible with almost all Linux environments, supporting finer-grained parameter-level auditing and logging. However, due to context

switching between user space and kernel space, it has some performance overhead (approximately 20%).

Ready to Use

The solution is provided as a wrapper layer for browser runtime. Deployment does not depend on external network configuration, container runtime versions, or cloud platform features. Security policies are distributed with the browser and behave consistently across any environment.

We have also adapted the solution for Puppeteer, Playwright, Cypress, and Selenium, with ready-to-use configuration files available in the GitHub repository's demos directory for direct cloning and use.

Fine-Grained Permission Control

Based on the least privilege principle, the solution provides three-layer control covering filesystem access, program execution, and network requests. Non-essential permissions are blocked by default, with targeted interception of high-risk operations through precise system call filtering. Path rules use allowlists (default-deny) to constrain file reads/writes and program execution, naturally avoiding "misconfiguration = permit." Compared to relying solely on blocklists or environment assumptions, this is easier to audit and prove permission boundaries. The solution ensures that even if RCE or logic vulnerabilities occur within the browser, attackers cannot obtain the critical capabilities to read/write sensitive system files or execute malicious programs, thereby significantly reducing impact scope.

Flexible Configuration and Extension

The solution productizes browser isolation policies into a composable, reusable, and evolvable policy system rather than a one-time "configure and forget" manual sandbox. It uses stable policy primitives (path-level access control covering read/write/execute, syscall rules for defining kernel capability boundaries) to express permission requirements, enabling policies to extend via "baseline policy + scenario-specific additions": new business requirements typically only require adding a few allowlist or exception entries, minimizing change scope and regression costs. Additionally, the engine provides audit logs, supporting quantitative identification and iterative refinement of policy gaps during development, with the refined policies ultimately migrated to execution paths for production. This design of "same-semantics cross-engine execution + baseline reuse + audit-driven iteration" constitutes the solution's truly sustainable extensibility.

Furthermore, based on in-depth analysis of system calls required for normal Chrome browser operation, we have preset security policies suitable for mainstream server-side scenarios. Policies follow the least privilege principle: permitting system calls required for normal functionality such as page loading, JavaScript execution, DOM manipulation, and screenshots; blocking high-risk operations such as arbitrary file reads/writes, process spawning, and command execution. Enterprises can use preset policies directly or adjust them according to business requirements.

5.2 Protection Effectiveness Analysis

To evaluate the solution's protective effectiveness, we tested multiple high-severity vulnerabilities across different modules on older Chrome versions. Our protection solution successfully blocked all attacker attacks. The protection results for some tested vulnerabilities are as follows:

CVE ID	Affected Component	Impact	Protection Result
CVE-2021-30551	V8 JIT	Remote command execution	Started with --no-turbofan parameter; vulnerability module not enabled; vulnerability ineffective
CVE-2021-38003	V8 Runtime	Remote command execution	Can only execute harmless commands on allowlist; immediately blocked and reported when attempting to execute commands outside allowlist
CVE-2023-4357	Libxslt	Arbitrary file read	Can only read harmless files on allowlist; immediately blocked and reported when attempting to read files outside allowlist
CVE-2023-4863	Libwebp	Remote command execution	Can only execute harmless commands on allowlist; immediately blocked and reported when attempting to execute commands outside allowlist
CVE-2024-10230	WebAssembly	Remote command execution	Can only execute harmless commands on allowlist; immediately blocked and reported when attempting to execute commands outside allowlist

5.3 Performance Testing

To quantify the performance overhead of both isolation approaches, we conducted multiple rounds of comparative testing based on the latest stable version Chrome/143.0.7499.146 across Chrome's three officially recommended performance benchmark platforms:

Benchmark	seccomp	ptrace Solution Overhead

	Solution Overhead	
Speedometer (Comprehensive Performance)	0.69%	33.10%
MotionMark (Graphics Rendering)	0.05%	23.72%
JetStream (JS/WASM Engine Performance)	1.82%	11.31%

Test results show that the seccomp solution maintains near-native runtime efficiency across all core performance dimensions, fully meeting server-side high-concurrency, low-latency business requirements. The ptrace solution, while having some overhead in rendering and comprehensive performance, still meets security isolation needs for medium-to-low load scenarios and offers stronger environment compatibility without requiring special system privileges.

6. Assessment Checklist and Implementation Recommendations

6.1 Risk Assessment Checklist

Security leaders can use the following checklist to quickly assess their enterprise's server-side browser risk status:

Asset Information

- How many places in the system use server-side browsers? Does this include backend services and non-obvious entry points?
- What browser version is currently in use? How does it compare to the latest stable version?
- Is the browser sandbox properly enabled? Is the --no-sandbox parameter being used anywhere?

Isolation Status

- Can browser containers access internal network resources? Is there potential for unauthorized lateral access?
- Is the browser container's filesystem read-only? Is the browser process running in unprivileged mode?
- Is instance isolation implemented for browser tasks to prevent data leakage and cross-task impact?

Operational Processes

- What is the browser version update cycle? Is there risk of delayed updates?
- Is there a clear security review process for configuration changes? Are there monitoring

and auditing mechanisms for browser behavior?

6.2 Implementation Recommendations

Security protection for server-side browsers requires a multi-dimensional approach, with various measures working together to form a complete defense system:

Strengthen Critical Security Boundaries: Ensure sandbox mechanisms are properly enabled to avoid exposing browsers to high-risk environments due to misconfigurations.

Reduce Attack Surface: Reduce potential vulnerability exploitation by disabling unnecessary functional modules and regularly updating versions.

Runtime Behavior Isolation: Restrict high-risk operations and monitor anomalous behavior to maintain control over attack impact even after vulnerability exploitation.

Infrastructure Isolation: Build defense in depth through network, filesystem, and instance isolation to further enhance security.

Defend Against Non-Traditional Attacks: Address hidden risks such as resource exhaustion, credential protection, and supply chain security to prevent overlooked vulnerabilities from becoming breach points.

Each measure is not independent but works together to build the security foundation for server-side browsers. Enterprises should gradually improve their protection systems based on actual circumstances, ensuring that security protection and system operational requirements are balanced.

7. Conclusion

As critical infrastructure in AI systems, server-side browsers can lead to severe consequences when compromised, such as controlled AI output content, poisoned Agent decision chains, and penetrated enterprise internal networks. However, as server-side browsers have migrated from client to server side, their role has changed significantly: browsers are no longer individual user tools but shared components serving multiple users and tasks, running in privileged environments and often directly connected to enterprise internal networks and critical business systems. This change amplifies both the harm and impact of server-side browser exploitation.

Our case studies found that enterprise-designed protection schemes (such as simple allowlist filtering) typically have bypass risks and cannot effectively resist sophisticated attacks. Therefore, we need a systematic set of protective measures, with key focus on:

Static Attack Surface Reduction: Following the least privilege principle to disable unnecessary functional modules and reduce potential attack entry points.

Dynamic Behavior Auditing and Restriction: Controlling post-exploitation impact scope to a minimum through runtime behavior control and multi-layer infrastructure isolation.

This protection framework is not only logically clear but also highly actionable, helping enterprise security teams effectively reduce the attack surface of server-side browsers while

limiting post-exploitation impact scope, laying the foundation for building comprehensive server-side browser defense capabilities.

Appendix: Terminology

RCE (Remote Code Execution): A vulnerability allowing attackers to execute arbitrary code on target systems.
Sandbox: An isolation mechanism that restricts program execution to a controlled environment.
0-day Vulnerability: A vulnerability not yet patched by the vendor.
N-day Vulnerability: A vulnerability that has been patched but the system has not been updated.
JIT (Just-In-Time Compilation): A compilation technique that improves JavaScript performance but is often targeted for attacks.
Prompt Injection: An attack method that manipulates AI behavior through malicious prompts.
seccomp: A Linux kernel mechanism for restricting system calls.
ptrace: A Linux process tracing tool used to monitor and control process behavior.