AI 网络爬虫安全白皮书

作者：李冠成、王征 @ 腾讯玄武实验室

摘要

AI 应用形态正从单一的 LLM 离线对话，逐步演进为能够调用工具、自主拆解任务的 Agent（智能体）的在线联网形态。无论是处理基础的“联网搜索并综述答案”，还是执行“先搜索公司信息、再查询股价，最后给出投资建议”这类复杂的自动化任务链，浏览器都会被集成到服务端系统中，以提供进行实时联网和内容提取。
然而，将本该运行在客户端的浏览器“搬”到服务端运行，这种架构上的错位带来了不容忽视的安全隐患：

- 信任边界模糊：浏览器作为解析外部不可信代码（如 JS、DOM）且高危漏洞频的软件，其所在服务端环境却可能直连企业内网和关键业务系统，极易成为外部攻击渗透内网的突破口。
- 安全水位更低：服务端浏览器常面临补丁更新滞后、运行权限过高等问题；部分厂商为兼顾兼容性甚至关闭原生沙箱机制，导致其安全防护能力往往低于普通客户端浏览器。
- 攻击危害更大：浏览器一旦被攻破，攻击者不仅能窃取任务数据或篡改返回结果以“污染”后续决策流程，利用共享架构横向影响其他产品和用户，更可能以此为跳板横向移动，攻击内网其他核心系统。

我们发表在 Blackhat 的一项研究也证实了多个 AI 产品的爬虫具有远程代码执行风险。鉴于服务端浏览器已成为 AI 服务端系统中的关键风险点，而行业内尚缺乏系统性的防护标准，本白皮书旨在填补这一空白。我们详细分析了该场景下的风险特征，并提出了以“静态攻击面收敛 + 动态行为隔离”为核心的防御框架，助力企业安全负责人和技术团队实现服务端浏览器的安全部署与运维。我们已在 GitHub 上开源了这套方案，希望能够助力行业整体提升服务端浏览器的安全水位。
代码地址：https://github.com/XuanwuLab/SEChrome

一、浏览器在 AI 系统中的攻防态势变化

当你启动一个浏览器实例时，你启动的不是一个简单的网页访问工具，而是一个由 V8 引擎、WebRTC 组件、PDF 阅读器、几十种音视频解码器及复杂渲染内核 组成的“微型操作系统”。任何一个组件的漏洞，都有可能引发远程代码执行，因而浏览器一直都是高危漏洞数量、以及可利用漏洞占比都是最多的软件。
在 AI 时代，浏览器从用户通向 Web 世界的入口，转变为了支撑 AI 业务运行的基础组件。我们将一个复杂度远超一般服务端组件且漏洞频发的浏览器，放置在攻击价值较高的服务端。这种变化不仅仅是部署位置的迁移，更是带来了一种角色的变化，这种角色变化最终导致了浏览器在服务端的攻防态势的变化。

1.1 攻击态势的变化

从攻击视角来看，浏览器角色的转变引入了深层次的结构性风险，这种变化可以归纳为以下四个维度：

1. 补丁+沙箱防御范式的失效：传统浏览器安全高度依赖“自动更新”与“沙箱隔离”。但在服务端，部分开发者为了维持环境一致性而禁用了自动更新；部分开发者为了适配容器架构而关闭了沙箱。这种运维环境的异化，直接导致了传统防御体系的失效，使 N-day 漏洞成为常态化威胁。
2. 攻击影响范围的扩大：传统浏览器仅影响个人终端。而在服务端，浏览器既是多用户共享的组件，攻击者一旦突破，通过共享环境影响其他用户，如批量控制其他用户联网搜索的结果。
3. 攻击导致的后果更加严重：传统浏览器的消费者是人，而现在的消费者是 AI。攻击者的目标不再仅仅是获取权限，更可以通过篡改网页内容来“投毒”AI 的知识输入。在“搜索-决策-执行”的自动化链路中，这种数据层面的攻击将直接操纵 AI 的最终决策。此外，服务端环境若没有恰当的隔离，攻击者可以利用其作为跳板，横向移动至核心内网，放大攻击者的危害。
4. 攻击意愿的提升：由于上述因素，服务端浏览器成为了通往企业核心数据与业务逻辑的捷径。相比于攻击个人用户，攻陷服务端浏览器的收益极高，这促使攻击者愿意投入更高成本（如购买 0-day 漏洞）来针对性地突破防线。

1.2 防御态势的变化

从防御的角度考虑，这种角色变化也造成了防御策略的错位。服务端安全的核心诉求在于最小权限原则，即组件应仅拥有完成特定任务所需的最小能力集合；但现有的浏览器设计为了兼容万维网，默认开启了 WebGL、WebRTC 及各种功能接口，这种“默认开放”的策略与服务端严苛的权限管控背道而驰，导致了攻击面的放大。
此外，现代服务端安全通常需要具备高度的可配置性与可运营性，安全策略应当像基础设施代码一样可定义、可审计。遗憾的是，浏览器往往作为一个不透明的黑盒运行，缺乏标准化的服务端配置接口与结构化的安全审计日志，导致运维人员既难以像配置 Nginx 那样对其行为边界进行有效收敛，也无法在攻击发生时获得足够的观测视野。
因此，我们必须重新评估服务端浏览器的风险，并建立一套全新的、符合服务端安全运营需求的浏览器安全防御实践。

二、AI 服务端浏览器安全风险评估

本节将系统性的重新评估服务端浏览器安全的风险

2.1 风险点 1：补丁更新滞后与错误的沙箱配置

客户端浏览器依赖成熟的安全机制运作：自动更新与快速补丁推送、原生沙箱、多进程隔离。Chrome 在桌面端通常能在漏洞披露后数天内向大部分用户推送更新。
服务端浏览器的运行方式有所不同，我们在对多个厂商的 AI 服务端浏览器组件进行分析后，

观察到以下两个普遍存在的问题，

- 沙箱配置问题：Chrome 的用户态沙箱依赖于 Linux 的命名空间（namespace）和 seccomp 机制。在容器环境中，若运行时配置不当（如未授予必要的 Linux Capabilities 或未正确设置 seccomp 策略），Chrome 将无法正常启动。为快速解决启动失败的问题，不少团队会选择添加 --no-sandbox 参数来绕过限制，但这实际上关闭了浏览器最关键的安全边界，带来严重风险。此外，部分容器平台出于防范容器逃逸漏洞的考虑，默认不开放 Chrome 沙箱所必需的系统权限。
- 版本更新慢：部分团队担心版本更新导致页面渲染行为变化或兼容性问题，倾向于使用经过长期验证的旧版本镜像。

根据 Google Project Zero 的统计，Chrome 在 2020-2025 年间累计修复了超过 1600 个安全漏洞，其中包含多个在野利用的零日漏洞。CVE 数据库的记录显示，Chrome V8 引擎、Blink 渲染引擎和 WebAssembly 运行时是漏洞的高发区域，这些组件在服务端浏览器中同样被使用。沙箱的关闭和版本更新的不及时，都会将浏览器暴露在 nday 漏洞的威胁之下，从而导致 AI 服务端的浏览器的远程代码执行风险。

2.2 风险点 2：多用户多产品共享架构导致漏洞影响变大

在客户端场景中，一个浏览器实例只服务一个用户，攻击所影响的范围天然受限。
在服务端 AI 系统中，浏览器通常以资源池的形式运行：同时服务多个 AI 产品，被多个任务队列复用，支撑多个用户的查询请求，在长生命周期容器中持续运行。
在这种架构下，如果一个恶意页面触发漏洞并成功利用，攻击者可能干扰正在处理或即将处理的多个任务，污染 LLM 的数据输入并进而控制行为，产生跨用户甚至跨产品的影响。

2.3 风险点 3：内网隔离不当导致的攻击危害变大

在客户端场景中，浏览器运行在用户个人电脑上，与企业生产系统通常存在网络隔离。
在服务端场景下，浏览器作为业务逻辑的一部分，可能与业务服务部署在同一集群或网络段中。如果缺乏专门的网络隔离策略，浏览器与内网资源（数据库、任务调度系统、模型推理节点、内部 API）可能处于可互访的状态。
一旦浏览器被攻破，攻击者将利用其所在节点作为立足点：扫描内网服务、读取环境变量中的凭据、访问元数据服务获取云平台权限。影响范围远超浏览器本身。

2.4 风险点 4：对于网页内容的使用方式导致的攻击危害变大

服务端浏览器不仅是数据获取工具，更是 AI 模型的“眼睛”和“耳朵”。攻击者可以通过在网页中植入 Prompt Injection（提示词注入）载荷或恶意指令，当浏览器抓取该页面内容并投喂给下游模型时，可能诱导 AI 输出错误结果、泄露敏感信息，甚至执行非预期的自动化操作。这种攻击跨越了传统的软件漏洞层面，可直接污染 AI 的决策逻辑。

2.5 风险点 5：服务端浏览器的攻击收益高，攻击者使用高价值漏洞的意愿强

在 AI 场景下，服务端浏览器的潜在价值显著提升。客户端攻击通常只能获取个人终端层面的数据与权限，而服务端浏览器的部署位置更接近业务系统。一旦攻破浏览器，攻击者能够进一步接触到内部接口、配置信息、业务数据或其他服务。因此，攻击者很有可能愿意使用投入更高的攻击手法，如使用 0-day 漏洞。
因此，“我们用的是最新版浏览器，所以不会有安全问题”是一种危险的误解。最新版可以缓解 N-day，但无法消除 0-day，这也是我们强调引入“纵深防御”的原因：假设浏览器内部总是可能存在未知漏洞，并提前为其失守设计好“缓冲区”。

三、攻击路径与真实案例

为了有效防御服务端浏览器威胁，我们首先需要理解攻击者视角下的渗透路径。通过对实战案例的复盘，我们梳理出了一套针对服务端浏览器的典型攻击链（Kill Chain），并精选了四个具有代表性的真实案例进行剖析。

3.1 服务端浏览器的攻击链路

针对服务端浏览器的攻击通常遵循以下五个阶段：

- 入口识别 (Reconnaissance)攻击者首先寻找能够触发服务端发起 HTTP 请求的功能点。除了显而易见的“网页抓取”、“URL 预览”功能外，还需要关注隐蔽的入口，如 Markdown 渲染中的图片加载、PDF 生成服务、缓存预热接口或后台的搜索引擎。
- 爬虫防御绕过 (Evasion) 针对系统部署的 URL 白名单或脚本过滤器，攻击者会利用 302 跳转、DNS 重绑定、协议解析差异等手段，或构造特殊的 HTML 结构来规避检查，将请求导向攻击者控制的网站。
- 环境指纹探测 (Fingerprinting)。HTTP 请求头中的 User-Agent 极易被伪造，因此攻击者不会采信。他们通过检测浏览器对特定 API、CSS 属性或 JS 语法特性的支持情况，判断浏览器内核的具体版本。
- 漏洞触发 (Exploitation) 一旦版本被确定，攻击者会从漏洞库中检索该版本存在的 0-day/N-day 漏洞，并部署相应的 Exploit 代码。当服务端浏览器解析恶意页面时，漏洞被触发，攻击者获得代码执行权限。
- 横向渗透与持久化 (Post-Exploitation) 若浏览器开启了沙箱，攻击者需要利用沙箱逃逸漏洞来逃逸沙箱，若浏览器未开启沙箱或沙箱配置不当，攻击者将直接获得宿主机的 Shell 权限，进而探测内网拓扑、读取敏感配置文件（如云服务 AK/SK），甚至以此为跳板攻击内部数据库和其他微服务。

3.2 四个典型案例

我们对市面部分集成服务端浏览器产品开展了测试工作，发现多款产品存在沙箱外的远程代

码执行（RCE）安全风险。相关产品服务端用户规模累计超 10 亿，涵盖多家头部科技企业旗下产品。

接下来，我们挑选了四个典型案例展开说明，以帮助全面理解该攻击面。

案例一：通过 URL 跳转绕过白名单并实现远程代码执行

- 入口识别 (Reconnaissance)：某 AI 搜索产品实施了访问白名单策略，只允许浏览器访问预设的可信网站列表。
- 防御绕过 (Evasion)：我们发现白名单中的某些大型搜索网站对收录的网站制作中间跳转链接如 xx.com/link?url=。我们构造了一个请求：访问白名单内的网站，但 URL 中附带跳转参数指向我们控制的服务器。系统检查初始地址在白名单内，放行请求。浏览器打开可信网站后，读取 URL 参数，自动跳转到了我们的恶意页面。
- 环境指纹探测 (Fingerprinting)：在成功跳转并绕过白名单后，我们的恶意页面在服务端浏览器中运行，探测发现其后台浏览器使用的是一个旧版本 Chrome/120。
- 漏洞触发 (Exploitation)：确认版本后，我们利用了一个该版本已公开的 N-day 漏洞+一个 V8 沙箱绕过的漏洞。由于该服务未开启浏览器沙箱，漏洞触发后我们成功获得了服务器的控制权。

案例二：组合多个浏览器功能并实现远程代码执行

- 入口识别 (Reconnaissance)：某 AI 产品有三个独立功能，分别是使用最新版浏览器的 AI 阅读功能、可生成同域公开链接的分享功能以及有 URL 白名单过滤的截图功能。
- 防御绕过 (Evasion)：我们构建了一条穿越这三个功能的攻击路径：先用“AI 阅读”功能分析一个包含恶意代码的网页，系统将恶意内容嵌入回答页面；接着将回答“分享”为公开链接；最后请求“截图功能”渲染该分享页面。由于分享链接属于本站域名，满足截图服务的白名单要求，恶意代码得以成功加载。
- 环境指纹探测 (Fingerprinting)：截图服务的浏览器打开分享页面时，嵌入的攻击者代码被执行。我们探测发现，尽管 AI 阅读功能使用的是新版浏览器，但截图服务的浏览器版本较旧
- 漏洞触发 (Exploitation)：针对该旧版本，我们测试了对应的 Exploit 代码，在能够执行 shellcode 之后，尝试读取 /proc/self/maps 文件，发现文件无法打开，由此间接推测截图服务的浏览器开启了沙箱。最终我们利用 SSRF 访问了云的 metadata，证明了其危害性。

案例三：绕过脚本执行限制并实现远程代码执行

- 入口识别 (Reconnaissance)：某 AI 产品具有访问对话中指定的 URL 功能，该功能过滤了网页中的所有 <script> 标签，期望阻止 JavaScript 执行。
- 防御绕过 (Evasion)：JavaScript 的执行方式不限于 <script> 标签，iframe 嵌入，事件注册都可以触发，我们构造了不包含任何 <script> 标签的页面，通过 <img src=x onerror="恶意代码"> 的方式触发代码执行。图片加载失败时的 onerror 事件处理器成功绕过了脚本过滤。

- 环境指纹探测 (Fingerprinting)：恶意代码执行后，我们探测发现它后台的浏览器所使用的版本是一个旧版本 Chrome/121。
- 漏洞触发 (Exploitation)：依据探测到的版本信息，我们检索并利用了一个已公开的 N-day 漏洞以及两个 V8 沙箱绕过漏洞。在使用第一个沙箱漏洞时，发现目标系统开启了 pkey 对 wasm 的运行时写保护，导致 shellcode 无法写入 wasm 的可读可写可执行代码区。因此，我们先用第一个沙箱绕过漏洞泄漏地址，再利用第二个沙箱绕过漏洞进行 JIT Spray 劫持程序执行流以实现利用。最终结合这三个漏洞，我们成功获取了服务器的控制权。

案例四：发现隐藏的后台浏览器入口并实现原创代码执行

- 入口识别 (Reconnaissance)：某 AI 搜索产品的前台功能拒绝访问我们的测试 URL。但我们保持测试服务器运行，三天后在日志中发现了来自该产品的访问记录。调查发现该产品有一个隐藏的后台索引系统，会在空闲时段批量抓取用户查询过的网址。
- 防御绕过 (Evasion)：该后台爬虫对用户不可见，不在任何产品文档中说明。攻击者无需直接绕过前台的实时检查，而是通过留下历史记录等待后台异步抓取，从而进入系统。
- 环境指纹探测 (Fingerprinting)：当后台爬虫访问我们的服务器时，我们发现它运行的浏览器版本确认为一个旧版本内核 Chrome/122。
- 漏洞触发 (Exploitation)：针对这个旧版本内核，我们部署了相应的 N-day 漏洞利用代码。由于该后台浏览器未开启沙箱，最终我们成功执行 shellcode 获得了部署该浏览器的服务器。

这些案例暴露了一个共性问题：防御思维的局限性。
企业往往认为添加了 URL 白名单就能高枕无忧，却忽视了白名单易被跳转绕过、黑名单易被混淆绕过的事实。更严重的是，安全视野往往遗漏了截图服务、后台爬虫等“隐形”浏览器入口。而一旦攻击者绕过第一道防线，在“旧版本浏览器”与“未开启沙箱”的组合往往导致系统直接被 nday 攻破。值得注意的是，即便是保持更新和开启沙箱，对于高价值的系统，攻击者仍可以选择使用 0day 漏洞进行攻击。
因此，应对服务端浏览器威胁，不能仅靠开发人员自行设计的防护方案，而需要一套系统化的纵深防御思路。

四、AI 服务端浏览器防御思路：从被动防护到主动运营

服务端浏览器需要一套不同于客户端的防御思路。针对服务端环境的特殊性，我们将防御策略拆解为以下四个关键维度：

1. 更加主动的防御策略。客户端浏览器通常依赖厂商的快速安全更新和内建的沙箱机制来抵御威胁。然而，服务端环境的补丁更新周期往往较长。此外，且受限于容器或虚拟化环境，浏览器沙箱经常因兼容性问题被迫关闭。这意味着我们不能单纯依赖客户端的“自动防御”机制，必须构建更主动的防线。
2. 攻击面收敛与最小权限原则。客户端浏览器为了兼容万维网的丰富生态，默认开启了 WebGL、WebRTC 及各类传感器接口。而在服务端场景中，我们应遵循最小权限原则 (Least Privilege)：组件应仅拥有完成特定任务所需的最小能力集合。必须坚决关闭不需要的浏览器功能，防止非必要的功能组件被攻击者利用。

3. 可配置可运营。在客户端场景下，浏览器的防护机制对用户而言通常是一个不可配置的“黑盒”。但服务端安全通常需要具备高度的可配置性与可运营性。服务端解决方案必须允许安全团队根据业务需求定制策略，而不是被动接受预设的安全配置，从而实现对安全态势的精准控制。“

4. 假设漏洞必然存在”的防御哲学。由于服务端浏览器承载着极高的攻击价值，我们不能排除攻击者使用 0-day 漏洞进行突防的可能性。因此，我们需要建立一套“假设漏洞必然存在”的防御体系。防御的重点不仅仅是防止漏洞触发，更在于假设防线被突破后，如何将损失控制在最小范围。

综合考虑以上所有因素，我们设计了一个针对 AI 服务端浏览器的防护方案：通过精简内核收敛可被利用的攻击面，并通过可配置的文件访问控制、新进程启动的白名单机制，严格限制漏洞被触发后的影响范围。

4.1 攻击面收敛

服务端浏览器的任务通常高度单一（如截图、文本提取）。许多为丰富人类体验而设计的复杂组件，在服务端不仅冗余，更是高危的攻击入口。

关闭无用的功能模块

- WebGL 和 GPU 加速渲染在服务端通常不需要，可通过--disable-gpu 和--disable-webgl 参数关闭。
- WebRTC 实时通信功能在服务端场景通常无用，且可能泄露内网 IP 地址，可通过--disable-webrtc 参数关闭。
- PDF 插件如无明确需求也应当禁用。
- V8 引擎的 JIT 编译器是漏洞高发区域，如果性能要求允许，可通过--jitless 参数关闭 JIT，代价是 JavaScript 执行性能下降，需要根据业务场景权衡。服务端浏览器不应加载任何扩展和插件。

据 CVE 数据分析，2023-2025 年间 Chrome 的高危漏洞中，约 16% 与 WebGL/GPU 相关，约 23% 与 V8 JIT 相关。通过启动参数直接禁用这些模块，可从源头切断近 40% 的漏洞。

配置建议

以下是一个面向安全的 Chrome 启动参数示例，安全团队可根据业务需求调整：

```
# 基础模式设置
--headless=new
--no-first-run
--disable-crashpad
--disable-crash-reporter

# 攻击面收敛（核心）
--disable-gpu                    # 禁用 GPU 硬件加速（高危区）
--disable-webgl                  # 禁用 WebGL（高危区）
--disable-webrtc                 # 禁用 WebRTC（防止 IP 泄露与 P2P 攻击）
--disable-extensions             # 禁止加载任何扩展
--disable-plugins                # 禁用 PDF 等插件

# V8 引擎加固（可选，视性能需求而定）
# 关闭 JIT 编译器，虽然降低 JS 性能，但能免疫绝大多数 V8 漏洞
--js-flags="--jitless"
# 或者仅关闭部分优化管道
--js-flags="--no-turbofan,--no-maglev"
```

如果性能允许，可以添加--jitless 进一步减少攻击面。

关于沙箱的说明

尽量不要使用 --no-sandbox 参数。
部分团队在容器中启动 Chrome 时，常为了解决沙箱虚权限报错而直接关闭沙箱。这是极其危险的做法。正确的路径是修复容器配置（如添加必要的 seccomp profile 或使用 --cap-add SYS_ADMIN），而非拆除最后一道防线。鉴于沙箱逃逸（Sandbox Escape）漏洞的稀缺性与高昂成本，保留原生沙箱具有极高的防御性价比。

4.2. 限制漏洞被触发后的影响

即便收敛了攻击面，考虑到浏览器是高危漏洞的重灾区，我们必须假设浏览器会被攻破。纵深防御的目标是：即使浏览器沦陷，攻击者也无法造成实际的危害，无法访问内网，无法窃取数据。
我们将纵深防御分为"基础设施层"和"浏览器运行时层"两道防线。基础设施层隔离，即利用容器、网络和操作系统提供的隔离机制实现的解决方案。浏览器运行时层隔离，则是通过对

浏览器进程的行为进行监控与审计，从而限制了浏览器被攻破后能执行的行为。

第一道防线：基础设施层隔离（粗粒度边界）

网络隔离：浏览器容器应当部署在独立的网络区域，只允许访问互联网，禁止访问内网资源。具体实施方式包括使用独立的 VPC 或子网，通过网络策略（Kubernetes NetworkPolicy 或云平台安全组）限制出站流量。验证方法是从浏览器容器内尝试访问内部服务（如元数据服务、内部数据库、其他业务 API），确认无法连通。
文件系统隔离：浏览器应当以只读方式运行，只允许写入特定的临时目录，避免攻击者通过写入定时任务、SSH 密钥或其他配置文件实现持久化。具体措施包括使用只读根文件系统（readOnlyRootFilesystem: true），将临时目录挂载为 tmpfs（内存文件系统，容器重启后清除），确保浏览器进程无法访问敏感配置文件。
实例隔离：理想情况下，每个任务使用独立的浏览器实例，任务完成后销毁实例。这可以防止一个恶意页面影响后续任务，也能避免信息在任务间泄露。如果性能要求不允许完全的实例隔离，至少应当在不同用户或不同安全等级的任务间使用独立实例。

第二道防线：浏览器进程行为管控（细粒度管控）

基础设施层提供了外部围墙，而运行时隔离则是在浏览器进程内部植入“监控探头”。通过监控并审查浏览器进程的系统调用（Syscall），限制其行为能力。即便是攻击者成功的对浏览器发起了 RCE 攻击，系统调用层面的行为管控仍能将漏洞影响限制在可控的范围之内。在第五章，我们将详细介绍基于该思想设计的防护方案 SeChrome。

两层隔离的协同

基础设施层隔离和浏览器层隔离相互补充。基础设施层提供粗粒度的网络和资源边界，作为纵深防御的外层；浏览器层提供细粒度的行为控制，在攻击发生的第一现场进行拦截。
即使浏览器层隔离被突破（例如通过内核漏洞），基础设施层隔离仍然有效；即使基础设施层配置存在疏漏，浏览器层隔离也能阻断大部分攻击。两者叠加，攻击者需要同时突破两道防线才能造成实质性危害。

4.3 其他需要关注的风险点

除了浏览器漏洞利用，服务端浏览器场景还有以下风险值得关注：
凭据和会话管理：如果浏览器需要登录某些网站，登录凭据和 Cookie 需做好跨用户的隔离。
资源耗尽攻击：恶意页面可能通过无限循环或大量内存分配消耗资源，导致服务拒绝。应当为浏览器容器设置 CPU 和内存限制，并设置页面加载超时。
供应链安全：Puppeteer、Playwright、Selenium 等工具本身也需要安全更新。应当将这些依赖纳入漏洞扫描和更新流程。

五、SEChrome，玄武浏览器防护方案

我们设计了 SEChrome，一种低成本、高收益的浏览器运行时隔离防护方案。该方案通过系统调用监控，对浏览器的文件访问、进程创建及网络请求等行为进行持续审查，从而实现对浏览器攻击（包括 0day 漏洞利用）的实时检测与阻断。评估结果表明，SEChrome 对多种浏览器攻击场景具有广泛的防御有效性。我们已在 GitHub 上开源该方案（https://github.com/XuanwuLab/SEChrome），期望为 AI 生态的安全建设贡献力量。

5.1 方案特点

双引擎隔离

SEChrome 采用 seccomp 与 ptrace 双引擎架构，以平衡安全性、性能与环境兼容性。
Seccomp 利用 Linux 内核的 BPF 功能在内核态直接过滤系统调用，性能损耗极低（<1%），适合高并发生产环境；但需要容器拥有 CAP_SYS_ADMIN 权限，在部分托管容器服务（如 AWS Fargate）中可能受限。
Ptrace 利用 Linux 进程跟踪机制在用户态拦截并审计系统调用，无需特殊权限即可运行，兼容几乎所有 Linux 环境，并支持更细粒度的参数级审计与日志记录；但因用户态与内核态之间的上下文切换，存在一定的性能开销（20%左右）。

开箱即用

方案作为浏览器运行时的封装层提供，部署不依赖外部网络配置、容器运行时版本或云平台特性，安全策略随浏览器一起分发，在任何环境中行为一致。
我们也适配了 Puppeteer、Playwright、cypress 和 Selenium 中，并将适配好的配置文件放到了 github 中(demos 目录)，方便直接克隆使用。

精细化权限管控

基于最小权限原则，提供文件系统访问、程序执行、网络请求的三层立体管控，非必要权限默认全阻断，通过系统调用精准过滤，定向拦截高风险操作。路径规则采用白名单（默认拒绝）来约束文件读写与程序执行，天然避免“漏配=放行”；相比只靠黑名单或环境约定，更容易通过审计证明权限边界。方案保证及时浏览器内部出现 RCE 或逻辑漏洞，攻击者也无法获得读写系统敏感文件和执行恶意程序的关键能力，从而显著降低影响面。

灵活配置扩展

方案将浏览器隔离策略产品化为一套可组合、可复用、可演进的策略体系，而不是一次性“调通即止”的手工沙箱。它用稳定的策略原语（路径级访问控制覆盖读/写/执行，syscall 规则用于补齐内核能力边界）来表达权限需求，使策略能够以“基线策略 + 场景增量”的方式扩展：新增业务通常只需追加少量白名单/例外项即可完成适配，变更范围小、回归成本可控。同时，引擎提供审计日志，支持在开发阶段对策略缺口进行定量定位与闭环收敛，最终将收敛后的策略迁移到执行路径用于生产。这种“同语义跨引擎执行 + 基线复用 + 审计驱动迭

代"的设计，才构成方案层面真正可持续的扩展能力。

此外，我们基于对 Chrome 浏览器正常运行所需系统调用的深入分析，预置了适用于主流服务端场景的安全策略。策略遵循最小权限原则：允许页面加载、JavaScript 执行、DOM 操作、屏幕截图等正常功能所需的系统调用；阻断任意文件读写、进程派生、命令执行等高危操作。企业可以直接使用预置策略，也可以根据业务需求进行调整。

5.2 防护效果分析

为了评测方案的防护效果，我们使用不同模块的多个高危漏洞在 Chrome 旧版本上进行测试，我们的防护方案均能成功阻止攻击者的攻击，测试的部分漏洞防护效果如下：

漏洞编号	影响组件	漏洞危害	防护效果
CVE-2021-30551	V8 JIT	远程命令执行	使用 --no-turbofan 参数启动，漏洞模块未启用，漏洞失效
CVE-2021-38003	V8 Runtime	远程命令执行	只能执行白名单内的无害命令，当执行白名单外的命令时立即阻断并上报
CVE-2023-4357	Libxslt	任意文件读	只能读取白名单内的无害文件，当尝试读取白名单外的文件时立即阻断并上报
CVE-2023-4863	Libwebp	远程命令执行	只能执行白名单内的无害命令，当执行白名单外的命令时立即阻断并上报
CVE-2024-10230	WebAssembly	远程命令执行	只能执行白名单内的无害命令，当执行白名单外的命令时立即阻断并上报

5.3 性能测试

为量化两种隔离方案的性能损耗，我们基于最新稳定版 chrome/143.0.7499.146，在 Chrome 官方推荐的三大性能基准测试平台开展多轮对比测试：

测试项目	seccomp 方案损耗	ptrace 方案损耗
Speedometer（综合性能表现）	0.69%	33.10%
MotionMark（图像渲染能力）	0.05%	23.72%

JetStream（JS/WASM 脚本引擎性能）	1.82%	11.31%

测试结果表明，seccomp 方案在所有核心性能维度均保持近原生的运行效率，完全适配服务端高并发、低延迟的业务需求；ptrace 方案虽在渲染和综合性能上存在一定损耗，但仍能满足中低负载场景的安全隔离需求，且无需依赖特殊系统权限，具备更强的环境兼容性。

六、评估清单与实施建议

6.1 风险评估清单

安全负责人可以使用以下清单快速评估本企业服务端浏览器的风险状态：

资产信息

系统中有多少处使用服务端浏览器？是否包括后台服务和非显式入口？
当前使用的浏览器版本是什么？与最新稳定版相比，版本滞后情况如何？
浏览器沙箱是否正常启用？是否存在使用 --no-sandbox 参数的情况？

隔离状态

浏览器容器是否能访问内网资源？是否存在未授权的横向访问可能？
浏览器容器的文件系统是否为只读？浏览器进程是否运行在非特权模式下？
是否为浏览器任务实现了实例隔离，避免任务间的数据泄露和影响？

运维流程

浏览器版本的更新周期是多久？是否存在延迟更新的风险？
是否有明确的配置变更安全审查流程？是否有针对浏览器行为的监控和审计机制？

6.2 实施建议

服务端浏览器的安全防护需要从多个维度入手，各项措施应协同发挥作用，形成完整的防御体系：
强化关键安全边界：确保沙箱机制正常启用，避免因配置不当而使浏览器暴露于高风险环境。
收敛攻击面：通过关闭不必要的功能模块和定期更新版本，减少潜在漏洞利用的可能性。
运行时行为隔离：限制高危操作，监控异常行为，做到漏洞利用后仍能控制攻击影响范围。
基础设施隔离：通过网络、文件系统和实例隔离构建纵深防御，进一步增强安全保障。
防范非传统攻击：关注资源耗尽、凭据保护和供应链安全等隐性风险，避免被忽视的漏洞成为突破口。
每项措施并非独立孤立，而是以相辅相成的方式共同构建服务端浏览器的安全基石。企业应根据实际情况，结合以上措施逐步完善防护体系，确保安全防护与系统运行需求平衡。

七、总结

服务端浏览器作为 AI 系统的重要基础设施，其被攻破可能导致更严重的后果，例如 AI 输出内容被控制，Agent 决策链被污染、企业内网被渗透。然而，由于服务端浏览器从客户端迁移到服务端，其角色发生了显著变化：浏览器不再是独立用户的工具，而是服务于多用户、多任务的共享组件，运行在高权限环境中，且往往直连企业内网和关键业务系统。这种变化导致服务端浏览器被利用的危害和影响均被放大。

我们的案例研究发现，企业自行设计的防护方案（如简单的白名单过滤）通常存在绕过风险，无法有效抵御复杂攻击。因此，我们需要一套系统化的防护措施，重点包括：

静态攻击面收敛：遵循最小化权限原则关闭无用功能模块，减少潜在的攻击入口。

动态行为审计与限制：通过运行时行为控制和基础设施多层隔离，将漏洞利用后的影响范围控制到最小。

这套防护体系不仅逻辑清晰且可操作性强，能够帮助企业安全团队有效减小服务端浏览器的攻击面，同时限制漏洞利用后的影响范围，为构建全面的服务端浏览器防御能力奠定基础。

附录：术语说明

RCE（Remote Code Execution）：远程代码执行漏洞，攻击者可在目标系统上运行任意代码。

沙箱（Sandbox）：隔离机制，将程序运行限制在受控环境中。

0-day 漏洞：尚未被厂商修复的漏洞。

N-day 漏洞：已修复但系统未更新的漏洞。

JIT（Just-In-Time Compilation）：即时编译技术，提升 JavaScript 性能，但易成为攻击目标。

Prompt Injection：通过恶意提示词操控 AI 行为的攻击方式。

seccomp：Linux 内核系统调用限制机制。

ptrace：Linux 进程跟踪工具，用于监控和控制进程行为。