V8 引擎 WebAssembly 类型混淆漏洞实证研究

谢远峰[1)]

天津大学智能与计算学部，天津市，中国

摘　要　WebAssembly（Wasm）作为一种专为现代开放网络设计的高性能二进制指令格式，正逐渐重塑 Web 开发的格局。它以提供接近原生的执行速度、跨平台兼容性及语言无关性，促进了游戏、计算密集型应用乃至桌面级软件在 Web 上的无缝运行。本文研究聚焦于 V8 引擎——Chrome 浏览器核心的 JavaScript 执行环境——在集成 WebAssembly 过程中遇到的安全挑战，尤其是类型混淆漏洞。V8 引擎在 WebAssembly 特性上支持领先，包括对 JS API、最低可行性产品(MVP)特性及后续提案的完整实现，甚至对处于探索阶段的特性亦有实验性支持。然而，伴随 WebAssembly 快速发展的同时，类型混淆漏洞等安全问题成为不容忽视的技术障碍，这些漏洞可能源于 WebAssembly 与 JavaScript 对象混淆、指令类型不匹配或编译器优化不一致，进而导致代码执行错误，甚至为远程代码执行攻击敞开了大门。基于此背景，本文通过实证研究，深入剖析了 V8 引擎中的三个典型 WebAssembly 类型混淆漏洞案例，从 JS 类型混淆、指令冲突到编译优化不一致性，全方位揭示了漏洞的成因、表现及潜在危害。研究不仅为理解 WebAssembly 的安全机制提供了深度见解，更为加固 V8 引擎及提升 WebAssembly 执行环境的整体安全性指明了方向，强调了未来研究在优化编译策略、增强类型系统健壮性及开发高效安全检测工具等方面的重要性。

关键词　WebAssembly, V8 引擎, 类型混淆漏洞, 安全性, 实证研究

Empirical Study of WebAssembly Type Confusion Flaw in V8 Engine

Yuanfeng Xie[1)]

Tianjin University College of Intelligence and Computing, Tianjin

Abstract　WebAssembly (Wasm), a high-performance binary instruction format tailored for modern web development, is reshaping the landscape by enabling near-native execution speeds, cross-platform compatibility, and language-agnostic deployment of applications ranging from games to computationally intensive software on the web. This study centers on the V8 engine, the JavaScript execution environment within Chrome browsers, and the security challenges it encounters in integrating WebAssembly, particularly type confusion vulnerabilities. Renowned for its leading support of WebAssembly features, including JS APIs, Minimum Viable Product (MVP) features, and experimental implementations of subsequent proposals, V8 nonetheless faces significant obstacles like type confusion vulnerabilities amidst WebAssembly's rapid evolution. These vulnerabilities can stem from various sources, including JavaScript object confusion, mismatched instruction types, or inconsistencies in compiler optimizations, leading to erroneous code execution or exposure to remote code execution attacks. Through empirical analysis, this research delves into three representative cases of type confusion vulnerabilities within the V8 engine related to WebAssembly, encompassing JavaScript type confusion, instruction conflicts, and inconsistencies in code optimization. By thoroughly examining their causes and potential hazards, the study not only enhances our comprehension of WebAssembly's security mechanisms but also charts a course for reinforcing the V8 engine and enhancing the overall security of the WebAssembly execution environment. It underscores the necessity for future research to concentrate on optimizing compiler strategies, bolstering the robustness of type systems, and developing efficient security inspection tools.

Key words　WebAssembly, V8 Engine, Type Confusion Vulnerabilities, Security Analysis, Empirical Study

---

收稿日期：年-月-日；最终修改稿收到日期：年-月-日 投稿时不填写此项. 本课题得到……基金中文完整名称(No.项目号)、… …基金中文完整名称(No.项目号)、…… 基金中文完整名称(No.项目号)资助.作者名 1(通信作者)，性别，xxxx 年生，学位(或目前学历)，职称，是/否计算机学会(CCF)会员（提供会员号）,主要研究领域为*****、****.E-mail: **************.作者名 2（通信作者），性别，xxxx 年生，学位(或目前学历)，职称，是/否计算机学会(CCF)会员（提供会员号）,主要研究领域为*****、****.E-mail: **************.

1 引言

WebAssembly，简称 Wasm，是一种为现代开放网络设计的二进制指令格式。它的目标是在网络上提供接近原生的性能，同时也为各种不同的平台提供一种通用的编译目标。这使得在 Web 上运行高性能应用程序成为可能，包括游戏、计算密集型任务，甚至是完整的桌面应用程序 [1]。

作为 Chrome 浏览器中编译执行 JavaScript 的主体，V8 引擎具有跨平台、可独立运行、可调试、可嵌入等特性。同时，V8 团队作为 Wasm 标准制定的主要参与者之一，V8 引擎对 WebAssembly 的 JS API、MVP 以及 Post-MVP 等核心提案都完整支持，并且实验性地支持各类处在探索阶段的提案[1][2]。因此，V8 引擎对于 WebAssembly 的支持程度相较于其它浏览器 JS 引擎更高。

正如许多新兴技术一样，WebAssembly 的快速发展也带来了一些挑战。尽管 WebAssembly 的设计目标是安全和高效，但由于当前 WebAssembly 核心提案仍在不断地开发以增强功能，编译器对于 WebAssembly 特性的适配仍旧处于开发测试阶段。作为 WebAssembly 的众多应用中，以浏览器 JS 引擎对于 WebAssembly 的支持最早；而在 JS 引擎中，以谷歌浏览器的 V8 引擎对于 WebAssembly 的支持最为完善。因此，本文选取 V8 引擎作为研究对象，确保研究的公平性。

本文针对 V8 引擎 WebAssembly 类型混淆漏洞实例进行实证研究。第二章介绍了 WebAssembly 的设计目标、表达形式、详细结构和支持特性。第三章介绍了 V8 引擎的架构、框架支持和代码支持。第四章分析类型混淆漏洞的原理，并对 V8 引擎中 WebAssembly 类型混淆漏洞进行了实证研究。第五章总结了本文工作，并对未来研究方向进行了展望。

2 WebAssembly

本章节将介绍 WebAssembly 的设计目标、表达形式、详细结构和支持特性。设计目标包括语义特性目标和表示特性目标。表达形式通过展示 C 语言到 WebAssembly 的二进制格式和文本格式的映射进行说明。详细结构介绍 WebAssembly 二进制模块结构各个组成段以及各个分段之间的联系。支持特性包括 WebAssembly 的支持浏览器、编译器、工具和库等方面的特性。

2.1 设计目标

WebAssembly 的最初的设计目标是创建一个快速、安全、可移植的编程语言，具有快速、安全、可移植的语义以及高效、可移植的表示，可以在网络上实现高性能应用，但不依赖于任何网络基础，也不提供特定的网络功能，因此也可以应用于任何其它环境[2][3]。WebAssembly 设计目标特性可概括为两点，分别是语义特性和表示特性。

(1) **快速、安全、可移植的语义**[2][3]。WebAssembly 旨在执行接近本地代码性能，利用所有现代硬件的共同功能。所有的 WebAssembly 代码都会在一个内存安全的沙盒环境中运行，防止数据损坏或安全漏洞。WebAssembly 全面且精确地定义了有效程序及其行为，使得人们可以非正式和正式地推理。WebAssembly 可以在所有现代架构（包括桌面、移动设备还是嵌入式系统）上编译。WebAssembly 不限定任何特定的语言、编程模型或对象模型。WebAssembly 具有平台独立性，可以嵌入浏览器，作为独立的虚拟机运行，或集成在其他环境中，以简单和通用的方式与其环境互操作。

(2) **高效、可移植的表达形式**[2][3]。WebAssembly 具有二进制格式，由于比典型文本或本地代码格式小，可以快速传输。同时，WebAssembly 程序可以分割成较小的部分，这些部分可以单独传输、缓存和使用。针对 WebAssembly 的编译，无论是即时（JIT）还是提前（AOT）编译，都可以在快速的单通道中进行解码、验证和编译。

2.2 表达形式

表 1 WebAssembly 表达形式[3][4]

二进制文本	可读文本	C 语言文本

02 40
03 40
20 00
45
0d 01
	block
	  loop
	    get_local 0
	    i32.eqz
	    br_if 1
		while(x != 0){

20 00 21 02	get_local 0 set_local 2	z = x;
20 01 20 00 6f 21 00	get_local 1 get_local 0 i32.rem_s set_local 0	x = y % z;
20 02 21 01	get_local 2 set_local 1	y = z;
0c 00 0b 0b	br 0 end end	}
20 01	get_local 1	return y;

表 1 所示为最大公因子函数求解在 C 语言、WebAssembly 二进制格式及 WebAssembly 文本格式下的表达形式对照。WebAssembly 最初作为二进制格式出现，但由于其可读性较差，后续推出了文

1 https://zhuanlan.zhihu.com/p/624211362

2 https://webassembly.github.io/spec/core/intro/introduction.html

3 https://v8.dev/blog/webassembly-browser-preview

本格式，以便于阅读和理解。WebAssembly 文本格式与低级汇编语言类似，但是更加简洁，不包含冗余信息，更加直观，且严格映射于二进制格式，二进制格式和文本形式可以通过开源工具 wabt[1][5] 进行相互转换。当前高级程序语言，如 C++[2][6]、Rust[3][7] 等，都可以编译为 WebAssembly 二进制格式，以便在浏览器中实例化运行调用。

2.3 详细结构

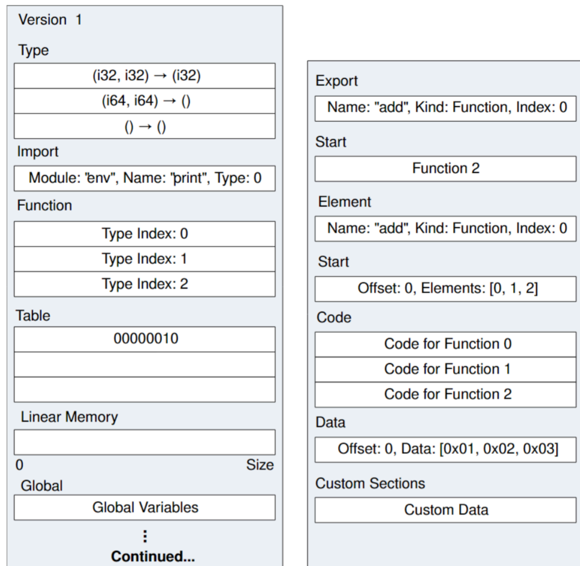

图 1　WebAssembly 详细结构二进制格式[8]

如图 1 所示，WebAssembly 模块构建于一个精密且层次清晰的架构之上，旨在通过一系列精心规划的组成部分来保障跨平台执行的高效性和安全性。此结构循序渐进，首先由版本声明(Version)奠定基石，明确了模块遵守的 Wasm 规范版本，为后续兼容性铺设了基础。

版本声明紧接类型段(Type)，类型段详尽地勾画了模块内部函数的签名蓝图，严谨地界定了参数与返回值的规格，从而确保了模块内外以及与宿主环境之间的无缝链接。导入段(Import)则扮演了桥梁角色，罗列并概述了对外部模块功能及全局变量的引用需求，深化了模块间的互动协作能力。

在此基础上，函数段(function)沿袭类型段设定的框架，系统性地组织模块内部的函数，确保调用接口的一致性与执行的精确性。为了促进执行效率与动态调度，表格段(Table)界定了存储结构，明确了类型、容量及元素属性；而内存段(Memory)则框定了模块可直接操作的内存范围，为资源的高效管理与访问制定了规则。此外，全局段(Global)文档化了模块的全局变量，详尽记录了它们的属性与初始状态，维护了状态的一致性与可预见性。

为了提升模块的互操作性，导出段(Exprot)暴露了模块的公开接口，列明了可供外界访问的功能入口，如函数和全局变量，增强了模块的可复用价值。启动段(Start)指定了加载初期需执行的初始化函数，确保了模块配置的恰当性与环境的预准备状态。元素段(Element)作为配置的补充，针对基于表格的间接调用进行了初始化设定。

核心的代码段(Code)承载了所有函数的编译代码，直接映射了模块的逻辑执行流程与功能实现场景。数据段(Data)专注于非指令数据的初始化，确保内存中静态数据的准确加载。自定义段(Custom)作为灵活的扩展机制，支持特定应用场景或环境数据的嵌入，极大提升了模块的灵活性与环境适应力。

2.4 支持特性

自 2015 年首次公布以来，WebAssembly 迅速发展，成为了现代 Web 开发不可或缺的一部分。Chrome、Firefox、Safari 等主流浏览器已全面支持 WebAssembly，确保了开发者编写的 WASM 模块能够在大多数用户的设备上无缝运行。这些浏览器支持 WASM 的即时编译（JIT），使得 WASM 代码几乎可以达到与原生代码相近的执行速度[4][9]。丰富的开发工具和库生态正在快速成长，例如 Emscripten[2][6]、Wasmer[5][10]、WasmPack[3][7] 等，简化了从源代码到 WASM 代码的编译流程。同时，WebAssembly 社区活跃，不断有新的案例、教程和最佳实践分享，帮助开发者更好地利用 WebAssembly[6]。

表 2　主流 JavaScript 引擎 Wasm 特性支持统计

JavaScript Engine	阶段 5[6]	阶段 4[6]	阶段 3[6]
V8	8	7	6
JavaScriptCore	8	3	3
SpiderMonkey	8	6	5

表 2 所示为针对主流 JavaScript 引擎对 WebAssembly 特性数量支持的统计。阶段 5 特性[6] [11] 为稳定特性，已在各引擎实现中完全支持；阶段 4 特性[6] [11] 为实验性特性，规范达成一致，已在各引擎实现中部分支持，等待充分测试合入正式规范；阶段 3 特性[6] [11] 为提案特性，规范文法确定，已在各引擎实现中部分支持，但仍旧存在实现不一致问题。从表中可以看出，V8 引擎对 WebAssembly 的支持程度最高，支持最新的 WebAssembly 特性，如阶段 4。V8 引擎支持的 WebAssembly 特性版本最新，且支持的特性数量最多，这使得 V8 引擎成为了 WebAssembly 的主要运行环境之一。

1 https://github.com/WebAssembly/wabt
2 https://github.com/emscripten-core/emscripten
3 https://github.com/rustwasm/wasm-pack
4 https://webassembly.org/features/
5 https://github.com/wasmerio/wasmer
6 https://github.com/WebAssembly/proposals

3 V8 引擎

本章节将介绍 V8 引擎对于 WebAssembly 的编译支持和代码支持。编译支持包括 v8 引擎的编译流水线、WebAssembly 编译器模块及编译策略。代码支持包括特性支持及测试工具支持。

3.1 编译支持

本节划分为三部分，编译流水线介绍 V8 引擎针对 JS 和 Wasm 的编译流程，WebAssembly 编译器介绍 WebAssembly 涉及的编译模块，编译策略说明编译模块的组合运用策略。

3.1.1 编译流水线

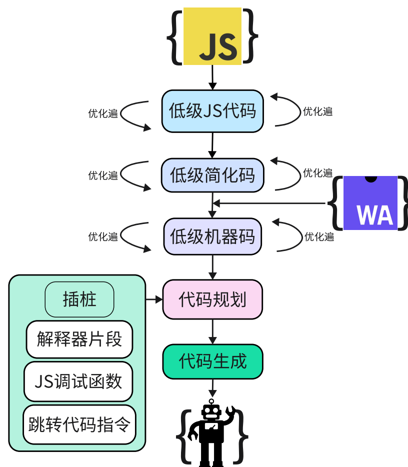

图 2 v8 引擎编译框架[1][12]

如图 2 所示为 V8 引擎的编译流水线框架。针对 JavaScript 和 WebAssembly，V8 利用统一框架对二者进行编译。JavaScript 作为动态语言，首先需要转换为 JS 低级代码，将代码中的动态类型转换为静态类型，内部可以执行多次优化遍利用代码遍历信息可以实现类型静态化的目标。之后，低级 JS 代码转换成不依赖于硬件平台的机器无关代码，在此期间可以进行无关硬件平台相关的优化遍，例如死代码删除，常量优化，控制流结构简化等。在经历两次中间代码转换后，JS 代码最终转换为静态类型的伪机器码。WebAssembly 作为静态类型低层次字节码，无需动态类型转换，具有可预测数量的参数，无函数重载操作，能够迅速进行验证和编译，因此 WebAssembly 代码可以直接转换为机器无关代码。

在获取到低级机器码后，V8 引擎会首先进行代码规划，将代码分为热代码和冷代码，热代码是指频繁执行的代码，冷代码是指不频繁执行的代码。通过插桩的形式，V8 引擎可以在代码中插入计数器，统计代码执行次数，当代码执行次数达到一定阈值时，V8 引擎会将代码标记为热代码，进行优化编译。冷代码则不会进行优化编译，以节省编译时间。最后，V8 引擎将规划后的代码传递到代码生成器中，生成对应于硬件平台的机器码。

3.1.2 WebAssembly 编译器

V8 引擎针对 WebAssembly 有着完整的架构支持，包括编译器，垃圾回收，运行时等。V8 引擎的 WebAssembly 编译器支持二进制格式，通过解码、验证、编译等步骤将 WebAssembly 代码编译为本地代码。V8 引擎针对 WebAssembly 编译的流水线架构包含两个编译器模块：LiftOff，TurboFan。两个编译器模块都可以将 WebAssembly 代码编译为本地代码，但是 LiftOff 编译器更加快速，TurboFan 编译器更加优化。下面将介绍 V8 引擎的两个 WebAssembly 编译器：LiftOff，TurboFan。

(1) **LiftOff 基线编译器**[2] [13]。作为一种基于懒加载机制的一次性遍历编译器，Liftoff 仅在函数首次调用时进行编译，结合其高效的单遍编译特性，实现了 WebAssembly 代码到机器码的快速转换，编译速率高达每秒数十兆字节。此机制不仅优化了资源占用，还通过将编译结果注册至模块内，确保后续函数调用的即时执行，提升整体运行性能。

(2) **TurboFan 优化编译器**[2] [13]。Liftoff 编译器虽能迅速生成可执行代码，但因独立处理每条 WebAssembly 指令而优化空间有限，TurboFan 作为多遍编译器的引入显得尤为重要。TurboFan 内置优化框架 Turboshaft 通过构建代码中间表示的多遍优化生成最优机器码，在面对频繁执行的"热函数"时，TurboFan 显著提升代码执行效率。

3.1.3 编译策略

```
async function instantiateWasm(imports) {
  const response = await fetch('module.wasm');
  const buffer = await response.arrayBuffer();
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(\
                          module, imports);
  return instance;
}
```

代码 1 WebAssembly 实例化代码[3][14]

如代码 1 所示，WebAssembly 二进制文件在 JavaScript 中通过 `fetch` 指令获取，并通过 `arrayBuffer` 加载到数组缓存中，通过 `compile` 指令执行静态编译，而后在实例化阶段导入 WebAssembly 内部声明的外部导入函数，生成可供 JavaScript 调用的 WebAssembly 实例，在后续使用中，WebAssembly 代码在运行时会进行第二次优化编译，从而提高运行速度。在编译过程中，为平衡 WebAssembly 编译速度和优化效果，V8 引擎会利用两种编译器模块，针对不同硬件平台和运行需求进行

1 https://github.com/hannespayer/v8-tutorial-pldi2019/blob/master/v8-compiler.pdf

2 https://v8.dev/docs/wasm-compilation-pipeline

3 https://pliss2019.github.io/ben_titzer_webassembly_slides.pdf

编译策略选择。以下列举出谷歌官方披露的四种编译策略，并给出编译策略的优缺点分析 [14]。

(1) **LiftOff 提前编译,TurboFan 后台完全编译**。该种策略静态编译采用 LiftOff，运行时间短。热身阶段完整，能够充分实现类型信息收集，但由于后期运行采用 TurboFan 进行代码全文优化编译，会对代码二次编译，但在实际运行过程中，大量函数尚未被调用运行，构成冷代码，耗费巨大内存空间。

(2) **LiftOff 提前编译,动态分层编译**。该策略静态编译采用 LiftOff，时间短。热身阶段完整，能够实现足够的类型信息收集。TurboFan 编译阶段会根据热身阶段搜集的信息定位热函数进行针对性的优化，节省了内存，降低了编译工作量，但也在一定程度上增加了热身阶段的时间。

(3) **LiftOff 懒加载编译,动态分层编译**。该策略静态编译时最短，未采用任何一种编译器模块，编译工作量最小，但与此同时也带来了启动阶段时间短，不做任何编译，后期热身时间需要交叉运用两种编译器模块进行编译，时间大幅延长的问题。

(4) **LiftOff 后台完全编译,动态分层编译**。该策略静态编译时间最短，未采用任何一种编译器模块，热身阶段完整，能够实现足够的优化信息收集,编译工作量减小，但需要长时间的热身，以便实现两个模块交叉使用，完成优化信息搜集，同时，静态编译阶段资源闲置，编译器未得到充分利用。

针对以上编译策略，谷歌官方提出了基于启发式懒加载的预先编译思想，使得静态编译时间短，热身阶段完整，可以实现足够优化信息收集，两阶段充分运用两个编译器模块，使代码在运行时能够得到最佳的执行效果。然而，编译阶段静态启发算法尚不完善，如何将静态编译与动态编译结合，实现最佳的编译策略，仍是一个值得研究的问题[14]。

3.2 代码支持

V8 引擎已实现的 WebAssembly 的代码[1]，支持当前 WebAssembly3.0 版本的所有特性[2]，并且跟随 WebAssembly 核心提案持续更新。为了增强 V8 引擎对于 WebAssembly 代码编译的鲁棒性，V8 引擎实现了基于 Libfuzzer 的生成式模糊工具[3]。

V8 引擎针对 WebAssembly 测试用例生成包含两种接口，两者都基于 WebAssembly 二进制格式进行生成，不同的是：一种是基于 C++的接口[4]，利于进行内部测试和编译解析，另一种是基于 JavaScript 的接口[5]，利于进行外部测试和漏洞复现。V8 引擎通过生成式模糊工具生成的测试用例，可以有效补充传统测试样例，发现常规测试难以捕捉的漏洞，以确保 V8 引擎对于 WebAssembly 代码的编译支持的鲁棒性，但也存在生成测试样例碎片化，文件过小，代码覆盖率低的情况。当前 WebAssembly 仍旧处于早期开发阶段，对于 WebAssebmly 的格式化验证和编译优化仍有较多的工作需要完善。

1 https://github.com/v8/v8/tree/main/src/wasm
2 https://github.com/WebAssembly/spec/tree/wasm-3.0
3 https://github.com/v8/v8/tree/main/test/fuzzer
4 https://github.com/v8/v8/tree/main/src/wasm/fuzzing
5 https://github.com/v8/v8/blob/main/test/mjsunit/wasm/wasm-module-builder.js

4 漏洞分析

本章节进行 V8 引擎上类型混淆漏洞的实证研究。首先介绍类型混淆漏洞的原理，并通过 V8 引擎中三个典型 WebAssembly 类型混淆漏洞（JS 类型混淆[6][15]，指令类型混淆[7][16]，优化错误[8][17]）实例分析，给出当前 WebAssembly 在 V8 引擎仍旧存在的问题分析，进行实例问题归类，说明当前 V8 引擎在 WebAssembly 支持上仍旧存在的问题。

4.1 类型混淆错误

```c
struct MessageBuffer {
  union {
    char *name;
    int nameID;
  }; // share common memory
};
int main(int argc, char **argv) {
  struct MessageBuffer buf;
  char *defaultMessage = "Hello World";
  buf.name = defaultMessage;
  printf("pointer:%p\n", buf.name); // 0x***004
  buf.nameID = (int)(defaultMessage + 1);
  printf("pointer:%p\n", buf.name); // 0x***005
  printf("Message: %s\n", buf.name);
} // Message: ello World
```

代码 2 类型混淆错误 C 语言示例

在 C 语言或 C++语言代码中，类型混淆错误可以定义为程序使用一种类型分配或初始化诸如指针，对象或变量等资源，但在之后使用与原始类型不兼容的类型访问该资源的情况。

代码 2 所示为针对 C 中 `union` 结构构造的简单类型混淆错误代码片段，字符指针类型变量 `name` 与整型变量 `nameID` 共用一段内存地址空间。程序首先为变量 `name` 赋值为默认字符串 `"Hello World"`，之后程序 `nameID` 被赋值成字符串 `"Hello World"` 的位置空间，由于 `name` 和 `nameID` 共用一个内存空间，导致对 `nameID` 赋值时，`name` 的赋值被错误覆盖，导致类型混淆错误产生。

在 V8 引擎的 WebAssembly 模块当中，类型混淆错误同样存在。V8 引擎对 WebAssembly 的支持程度较高，但在 WebAssembly 的类型混淆错误检测上仍存在一定的问题。下面将介绍 V8 引擎中三个典型 WebAssembly 类型混淆漏洞实例，分别为 JS 类型混淆[6]，指令类型混淆[7]，编译错误类型混淆[8]。

6 https://issues.chromium.org/issues/40067712
7 https://issues.chromium.org/issues/40067050
8 https://issues.chromium.org/issues/41484431

4.2 JS 类型混淆

如表 3 所示为 V8 引擎中 WebAssembly 类型混淆漏洞 CVE-2023-4068 信息，漏洞影响平台包括 Windows,Android,Linux,Mac 等，影响版本为 V8-115.0.5790.170 及以前版本，漏洞奖金为 23000 美金。漏洞报告时间为 2023-07-20，漏洞评级为 S1 High，属于严重漏洞。该漏洞导致 wasm 空值与 JavaScript 的 null 值混淆，攻击者能够利用这种类型混淆绕过预期的隔离限制，直接操作 V8 堆栈空间地址内容，为远程代码执行打开门户。

表 3　CVE-2023-4068 漏洞信息[1]

谷歌威胁评级	S1 High
CVE 编号	2023-4068
报告时间	2023-07-20
影响平台	Windows,Android,Linux,Mac 等
影响版本	V8-115.0.5790.170 以前版本
奖金	23000 美金

V8 引擎引入 wasm 空值产生了两种类型的空值。一种称为 `wasm-null`，用于 wasm 运行时(隔离)，另一种称为 `null-value`，应用于 JS 层。因此，当对对象引用执行使用、赋值或非空判断等操作时，必须首先确定该对象是内部对象还是外部对象。从而决定是加载 `wasm-null` 还是 `null-value`。

```
(module
  (type (;0;) (struct (field (mut i32))))
  (type (;1;) (struct (field (mut ref null 0) )))
  (type (;2;) (struct (field (ref null 1))))
  (type (;3;) (struct (field (ref null 2))))
  (type (;4;) (struct (field (ref null 3))))
  (type (;5;) (func))
  (table (;0;) 1 2 (ref null struct)) ;;structref
  (elem (;0;) (table 0) (offset 0)
    (struct.new_default 4)
  )
  (func (;0;) $main (type 5)
    i32.const 0,
    table.get 0,
    struct.get 4 0,
    struct.get 3 0,
    struct.get 2 0,
    struct.get 1 0,
    struct.get 0 0,
    drop,
    end
  )
  (export "main" (func $main))
)
```

代码 3　CVE-2023-4068 漏洞 WebAssembly 文本格式

然而在代码[2]中，对于结构体和数组的默认值设置并不区分对象类型，而是直接将所有 `null` 引用赋值为 JavaScript 类型 `null-value`，而不是 `wasm-null`。(本该隔离的 null 值却暴露给外部 JS)。结构成员的空指针(null)以后可能会被视为 `wasm-null`，这可能会与此处分配的“空值“相混淆。(可以对 null 值进行操作，读写 V8 堆栈空间地址内容)

代码 3 所示为 WebAssembly 文本格式漏洞代码，类型段中定义了 5 种类型，索引 1 类型为 32 位整型常量，索引 2-4 为结构体类型索引，指向上一索引类型，索引 5 为函数类型。表格段中定义了一个表格，表格中存储了一个结构体类型的空值。元素段中将表格索引 0 位置的元素赋值为一个新的结构体类型的空值。函数段中定义了一个函数，函数中依次获取表格索引 0 位置的元素的结构体类型的空值的各个字段，最后将栈顶元素弹出，获取 V8 堆栈空间地址内容。

4.3 指令类型混淆

表 4 所示为 V8 引擎中 WebAssembly 类型混淆漏洞 CVE-2023-4069，漏洞影响平台包括 Windows,Android,Linux,Mac 等，影响版本为 V8-115.0.5790.170 及以前版本，漏洞奖金为 20000 美金。漏洞报告时间为 2023-07-07，漏洞评级为 S1 High，属于严重漏洞。该漏洞导致 `extern.convert_any` 指令对 `ref null 0` 指令在 rax 寄存器存储的内容进行修改，绕过了 WebAssembly 沙箱隔离，可以读取到 V8 堆栈空间内容，实现远程代码执行操作。

表 4　CVE-2023-4070 漏洞信息[3]

谷歌威胁评级	S1 High
CVE 编号	2023-4070
报告时间	2023-07-07
影响平台	Windows,Android,Linux,Mac 等
影响版本	V8-115.0.5790.170 以前版本
奖金	20000 美金

如代码 4[4]所示，V8 引擎在 WebAssembly 处理函数调用参数的方式上存在缺陷。函数 main 调用时，参数既存于栈中，也可能直接置入寄存器内。例如，调用 `main` 函数时，`ref.null 0` 创建的空指针作为参数入缓存栈，同时存储在 rax 寄器中。这虽看似无害，实则为漏洞埋下隐患。

栈帧的平衡是 WebAssembly 实现调用过程中的关键。栈帧必须保持完整，确保调用的每个元素在函数结束时能够恢复到初始状态。进入 `callee` 函数后，因 rax 寄存器直接存储 ref 指针参数，为确保后续调用无法更改修改寄存器值，寄存器被锁定为

1 https://issues.chromium.org/issues/40067712
2 https://chromium.googlesource.com/v8/v8/+/b947905d27518b7764607708ec9f74ac3ea94b6b%5E%21/

3 https://issues.chromium.org/issues/40067050
4 https://chromium-review.googlesource.com/c/v8/v8/+/5319506/3/test/mjsunit/regress/wasm/regress-1462951.js

只读状态。故需通过 local.get 操作从寄存器拷贝值到缓存栈中，以确保缓存栈的正确性。

```
(module
  (type (;0;) (struct (field (mut ref null struct))))
  (type (;1;) (func))
  (type (;2;) (func (param ref null 0)))
  (table (;0;) 2 2 funcref)
  (func (;0;) $main (type 1)
    ref.null 0,
    call $callee,
    end
  )
  (func (;1;) $callee (type 2)
    local.get 0,
    extern.convert_any, ;;extern.externalize
    local.get 0,
    struct.get 0 0,
    ref.cast 0,
    drop,
    drop,
    end
  )
  (export "main" (func $main))
)
```

代码 4　CVE-2023-4070 漏洞 WebAssembly 文本格式

extern.convert_any 指令[1]是本次漏洞操作的关键点，它直接修改寄存器内容。外部调用时，攻击者可通过 extern.convert_any 指令改变寄存器如 rax 值[2]，利用 ref.null 与 extern.covert_any 的类型混淆问题修改了 ref 参数，构成了可以访问外部资源的非法指针，这使攻击者操控函数输入，为漏洞利用奠定基础。

4.4 优化错误类型混淆

表 5　issues-41484431 漏洞信息[3]

谷歌威胁评级	S1 High
CVE 编号	—
报告时间	2023-12-15
影响平台	Windows,Android,Linux,Mac 等
影响版本	V8-120.0.6099.212 以前版本
奖金	11000 美金

表 5 所示为 V8 引擎漏洞编号 41484431 的信息汇总。漏洞报告时间为 2023-12-15，漏洞影响平台包括 Windows,Android,Linux,Mac 等，影响版本为 V8-120.0.6099.212 以前版本，漏洞奖金为 11000 美金，漏洞评级为 S1 High，属于严重漏洞。LiftOff 和 TurboFan 编译器协同工作时，由于编译阶段信息的不一致传递，可能导致对某些函数的优化决策失误，进而引发类型混淆错误，如对参数和返回值处理不当，这类问题直接关联到编译器优化策略的匹配与执行效率，可被利用进行远程代码执行攻击。

V8 引擎针对 WebAssembly 的编译包含两个编译器，在 TurboFan 编译器模块中，IR 层架次架构为 TurboShaft。该漏洞问题为 LiftOff 编译器模块与 TurboShaft 编译器模块架构 Turboshaft 优化不一致。

TurboShaft 针对 WebAssembly 中 CallRef 指令和 CallDirect 指令进行优化时会进行图生成，图生成过程中会将 CallRef 指令和 CallDirect 指令等调用指令进行可达性分析，判断调用指令是否可达。而 TurboShaft 在图生成过程中，TurboShaft 假定调用指令指向的代码块是不可达的。但针对调用指令执行内联优化时，内联代码可达。优化前后信息冲突，造成调用指令优化错误，涉及待内联代码块传入参数和返回值优化错误，导致类型混淆。

```
d8.file.execute('../wasm-module-builder.js');
(function Simple() {
  const builder = new WasmModuleBuilder();
  let void_function = builder.addFunction(
    "void", makeSig([], [])
  ).addBody([]);
  let main_function = builder.addFunction(
    "main", makeSig([kWasmI32], [kWasmI32])
  ).addBody([
      kExprTry, kWasmI32,
        kExprCallFunction, void_function.index,
        kExprI32Const, 42,
      kExprCatchAll,
        kExprCallFunction, void_function.index,
        kExprI32Const, 0,
      kExprEnd
    ]).exportFunc();
  let main = builder.instantiate().exports.main;
  main();
  // main function TurboFan optimize compile
  %WasmTierUpFunction(main);
  main();
})();
```

代码 5　issues-41484431 漏洞 JavaScript 格式[4]

如代码 5 所示，漏洞代码中定义了两个函数，void_function 和 main_function。void_function 函数逻辑为空，为无返回值函数，可看成为死代码。main_function 传入 32 位参数，返回 32 位参数，函数逻辑为采用异常捕获方法，调用 void_function 函数，压栈常量 42，出现异常时调用 void_function 函数，压栈常量 0。最后，实例化 main_function 函数，调用 main 函数，执行 TurboFan 优化编译，再次调用 main 函数。在 TurboFan 优化编译过程中，由于 void_function 函数为死代码，TurboFan 优化编译时会将 void_function 函数优化掉，导致 main_function 函数调用 void_function 函数时出现异常，进而导致类型混淆错误。

1 漏洞披露时指令名称为 extern.externalize

2 https://chromium.googlesource.com/v8/v8/+/65a913ada7c46baaab65580e31b4d3cc2114973f%5E%21/

3 https://issues.chromium.org/issues/41484431

4 https://chromium.googlesource.com/v8/v8/+/acbfa775ae2a3a0e7e6b7528f33d545940f28ea4/test/mjsunit/regress/wasm/regress-1511849.js

4.5 漏洞总结

通过分析 3 个典型 V8 引擎 WebAssembly 类型混淆漏洞，可以将 WebAssembly 漏洞划分为 3 类：WebAssembly 隔离问题，WebAssembly 指令冲突，WebAssembly 代码优化。WebAssembly 隔离问题主要是由于 WebAssembly 与 JavaScript 对象混淆导致的类型混淆错误，WebAssembly 指令冲突主要是由于 WebAssembly 指令对底层寄存器操作误操作导致的类型混淆错误，WebAssembly 代码优化主要是由于 WebAssembly 编译优化不一致导致的类型混淆错误。这些类型混淆错误可能导致 WebAssembly 代码执行错误，进而被攻击者利用实现远程代码执行攻击。因此，V8 引擎对 WebAssembly 代码支持仍旧存在一定的问题，需要进一步完善。

5 总结

本文首先回顾了 WebAssembly（Wasm）的核心设计理念与技术特征。WebAssembly 旨在提供快速、安全且可移植的编程环境，能够在不同平台上实现接近本地代码的性能表现，支持跨语言编译与执行。其二进制表达促进了高效传输和编译，而文本格式则增加了可读性与调试便利性。WebAssembly 的模块结构严谨，从类型定义、导入导出到内存分配等方面，都确保了模块间的高效交互与安全。

在第三章中，本文深入探讨了 V8 引擎的架构与对 WebAssembly 的支持细节。V8 作为 Chrome 浏览器的 JavaScript 执行引擎，不仅具备跨平台运行的能力，还对 WebAssembly 提供了广泛的支持，包括 JS API、MVP 特性及后续提案。这表明 V8 在 WebAssembly 的实现上具有前沿性与完整性。

在第四章中，本文深入探讨 V8 引擎中 WebAssembly 类型混淆漏洞的三个方面，通过实例分析，清晰展现了类型混淆错误的多样性和复杂性。

综上所述，本文通过理论阐述与实证分析相结合的方式，揭示了 V8 引擎在 WebAssembly 支持中的技术优势与存在的安全隐患，特别是类型混淆漏洞的产生原因与影响。这些发现不仅加深了对 WebAssembly 安全机制的理解，也为未来优化 V8 引擎及提升 WebAssembly 执行环境的安全性提供了重要参考。未来的研究方向可能包括进一步优化编译器的代码优化策略，加强类型系统的鲁棒性，以及开发更有效的安全检测工具以预防和修复潜在的类型混淆漏洞。

参考文献

[1] A. Haas *et al.*, “Bringing the web up to speed with WebAssembly,” *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, Jun. 2017, doi: 10.1145/3140587.3062363.

[2] 杨文明, “WebAssembly 常见引擎简介.” Accessed: Apr. 16, 2023. [Online]. Available: https://zhuanlan.zhihu.com/p/624211362

[3] WebAssembly Community Group, “WebAssembly 2.0 (Draft 2024-04-28).” Accessed: Apr. 16, 2023. [Online]. Available: https://webassembly.github.io/spec/core/intro/introduction.html

[4] V8 Project Authors, “WebAssembly Browser Preview.” Accessed: May 08, 2024. [Online]. Available: https://v8.dev/blog/webassembly-browser-preview

[5] WebAssembly Community, “WebAssembly Binary Toolkit (WABT).” Accessed: May 08, 2024. [Online]. Available: https://github.com/WebAssembly/wabt

[6] Emscripten Core Team, “Emscripten: An LLVM-to-WebAssembly Compiler.” Accessed: May 08, 2024. [Online]. Available: https://github.com/emscripten-core/emscripten

[7] Rust and WebAssembly Working Group, “wasm-pack: Rust to WebAssembly Package Manager.” Accessed: May 08, 2024. [Online]. Available: https://github.com/rustwasm/wasm-pack

[8] P. P. Ray, “An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions,” *Future Internet*, vol. 15, no. 8, 2023, doi: 10.3390/fi15080275.

[9] “WebAssembly Features.” Accessed: May 08, 2024. [Online]. Available: https://webassembly.org/features/

[10] Wasm Developers, “Wasm: High-performance WebAssembly runtime.” Accessed: May 08, 2024. [Online]. Available: https://github.com/wasmer

[11] “WebAssembly Proposals.” Accessed: May 08, 2024. [Online]. Available: https://github.com/WebAssembly/proposals

[12] H. P. et al., “V8 Compiler Tutorial,” 2019. Accessed: May 08, 2024. [Online]. Available: https://github.com/hannespayer/v8-tutorial-pldi2019/blob/master/v8-compiler.pdf

[13] V. D. Team, “WebAssembly Compilation Pipeline in.” [Online]. Available: https://v8.dev/docs/wasm-compilation

[14] B. Titzer, “WebAssembly Slides,” 2019. [Online]. Available: https://pliss2019.github.io/ben_titzer_webassembly_slides.pdf

[15] Jerrry, “Security: Memory corrupt in v8, leading to RCE.” Accessed: May 08, 2024. [Online]. Available: https://issues.chromium.org/issues/40067712

[16] Jerrry, “Security: Type Confusion in V8 WebAssembly, leading to RCE.” Accessed: May 08, 2024. [Online]. Available: https://issues.chromium.org/issues/40067050

[17] Jerrry, “Security: Debug check failed: inlinee.sig->return_count() == sig->return_count() . in v8.” Accessed: May 08, 2024. [Online]. Available: https://issues.chromium.org/issues/41484431