

腾讯安全 | 云鼎实验室

锋刃无影 御见未来

腾讯安全沙龙第3期（成都站）



关于我

About me



张向伟

- 腾讯云鼎实验室安全研究员
- 研究方向为基于模糊测试和大模型的漏洞挖掘
- 主要从事浏览器安全和主机安全工作
- 在 *Apple Safari* 和 *Mozilla FireFox* 中发现数个安全漏洞

WebAssembly新攻击面下的浏览器 漏洞挖掘探索

浏览器在日常生活被高频使用



- 网页浏览
- 社交媒体
- 网上购物
- 网上银行
- 在线协作
-

浏览器始终是安全研究最具价值、也最具挑战性的对象之一

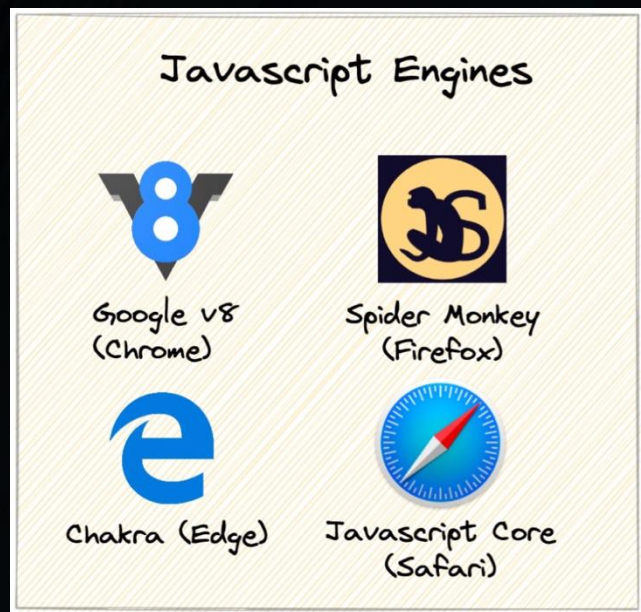
浏览器可能被入侵



- Mozilla Firefox 漏洞利用
 - 在 Pwn2Own 2022 上, 安全研究员 Manfred Paul 成功演示了 2 个漏洞, 获得 \$100,000 奖金。
- Apple Safari 漏洞利用
 - Manfred Paul 再次成功攻破 Safari, 赢得 \$50,000 奖金。

浏览器漏洞不仅客观存在, 而且价值高、危害大

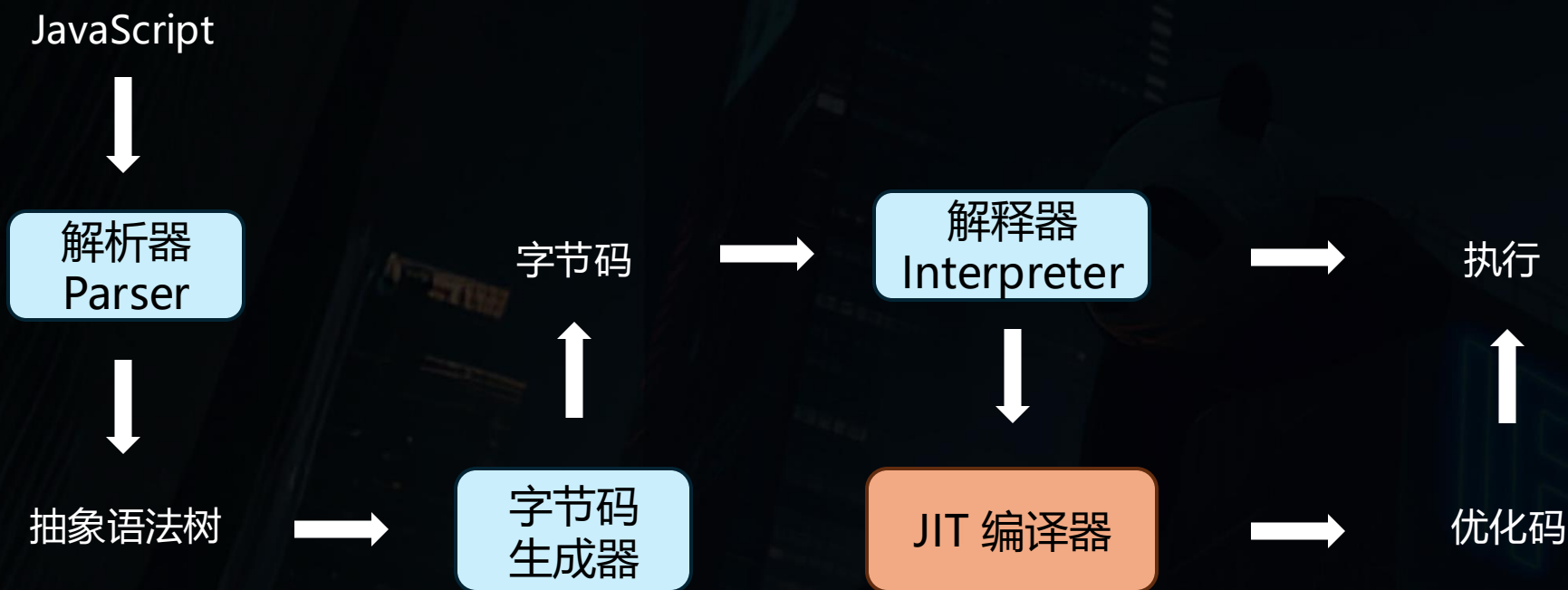
JavaScript 引擎驱动浏览器



- 解析和验证 JavaScript
- 执行 JavaScript
- JIT 编译和优化 JavaScript

正是这一层层动态处理，为漏洞利用提供了丰富而复杂的攻击面。

JavaScript 引擎架构



多阶段、跨线程、带状态地修改代码——这正是过去几年 JIT 部分漏洞频出的原因。

JIT 编译器进行了大量优化

```
var c = a + b;  
var d = a + b;
```



```
var c = a + b;  
var d = c;
```

- 边界检查消除
- 常量折叠
- 死代码消除
- 公共子表达式消除
- 冗余消除
-

JIT 编译器容易出错

```
var c = a + b;  
var d = a + b;
```



```
var c = a + b;  
var d = c;
```

- JavaScript 是一种弱动态类型语言。
- 由于变量类型的潜在模糊性，直接优化并不现实。
- JIT 编译器利用运行时信息对变量类型进行剖析，从而做出优化决定。

JIT 编译器容易出错

JIT 编译器容易出错

CVE-2021-21220 (JIT)

CVE-2020-9805 (JIT)

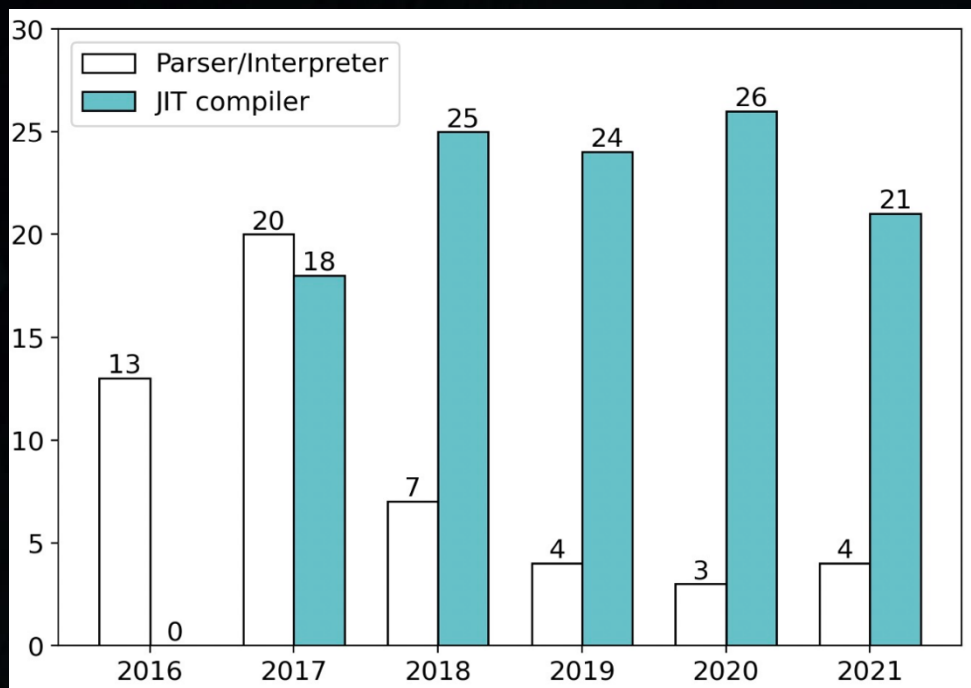
CVE-2019-9813 (JIT)

CVE-2019-6217 (JIT)

CVE-2019-6216 (JIT)

- 在 2019 年至 2021 年 8 次成功的 Pwn2Own 演示中，有 6 次都利用了 JIT 编译器的漏洞。

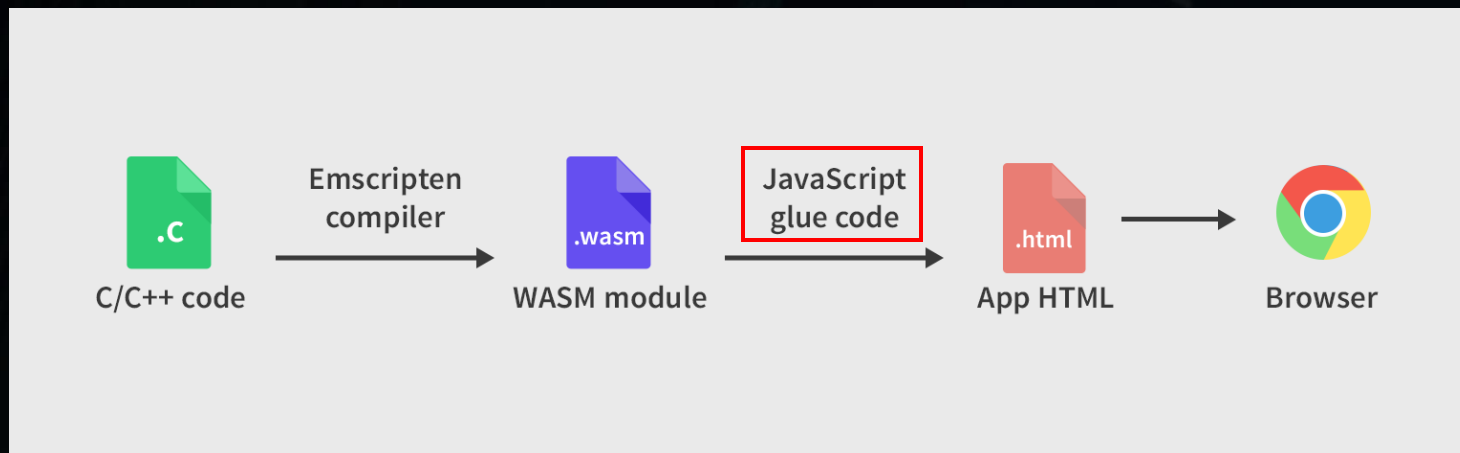
JIT 编译器容易出错



- 在前些年中，JIT 编译器错误的数量大约是解析器/解释器错误数量的四倍。是当时的热点攻击面。

Wasm, 下一波攻击趋势?

Wasm 基本概念



WebAssembly (Wasm) 是一种新型的、可移植的、高效的二进制指令格式，可作为多种高级语言（如 C、C++、Rust 等）的编译目标，从而在 Web 上运行接近原生性能的应用程序。

浏览器利用 JS 引擎 编译执行Wasm

浏览器无法直接识别Wasm文件，需要 JavaScript 胶水代码才能加载它们。

```
int addTwo(int a, int b){  
    return a + b;  
}
```

C

```
const wasm_code = new Uint8Array([0, 97, 115, 109, 1, 0, 0,  
0, 1, 7, 1, 96, 2, 127, 127, 1, 127, 3, 2, 1, 0, 7, 10, 1, 6,  
97, 100, 100, 84, 119, 111, 0, 0, 10, 9, 1, 7, 0, 32, 0, 32,  
1, 106, 11, 0, 10, 4, 110, 97, 109, 101, 2, 3, 1, 0, 0]);  
  
var wasm_module = new WebAssembly.Module(wasm_code);  
const wasm_instance = new WebAssembly.Instance(wasm_module, {});  
const { addTwo } = wasmInstance.exports;  
console.log(addTwo(512, 512)); // Output: 1024
```

JavaScript

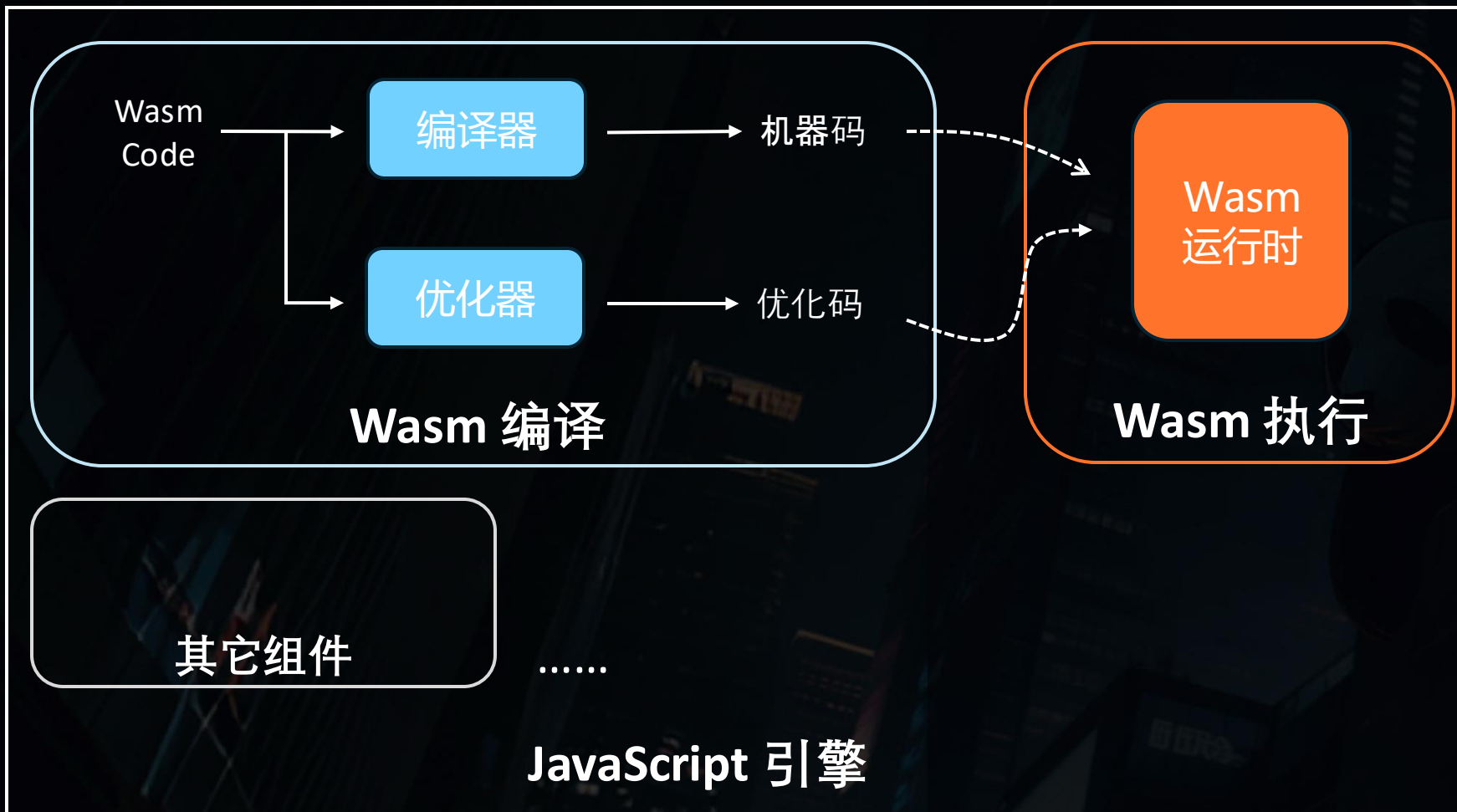
浏览器

Emscripten

```
000000: 0061 736d      ; WASM_BINARY_MAGIC  
000004: 0100 0000      ; WASM_BINARY_VERSION  
; section "Type" (1)  
000008: 01           ; section code  
000009: 00           ; section size (guess)  
00000a: 01           ; num types  
; func type 0  
00000b: 60           ; func  
00000c: 02           ; num params  
00000d: 7f           ; i32  
00000e: 7f           ; i32  
00000f: 01           ; num results  
000010: 7f           ; i32  
000009: 07           ; FIXUP section size  
; section "Function" (3)  
000011: 03           ; section code  
000012: 00           ; section size (guess)  
000013: 01           ; num functions  
000014: 00           ; function 0 signature index  
000012: 02           ; FIXUP section size  
; section "Export" (7)  
000015: 07           ; section code  
000016: 00           ; section size (guess)  
000017: 01           ; num exports  
000018: 06           ; string length  
000019: 6164 6454 776f "addTwo" ; export name  
00001f: 00           ; export kind  
000020: 00           ; export func index  
000016: 0a           ; FIXUP section size  
; section "Code" (10)  
000021: 0a           ; section code  
000022: 00           ; section size (guess)  
000023: 01           ; num functions  
; function body 0  
000024: 00           ; func body size (guess)  
000025: 00           ; local decl count  
000026: 20           ; local.get  
000027: 00           ; local index  
000028: 20           ; local.get  
000029: 01           ; local index  
00002a: 6a           ; i32.add  
00002b: 0b           ; end  
000024: 07           ; FIXUP func body size  
000022: 09           ; FIXUP section size  
; section "name"  
.....
```

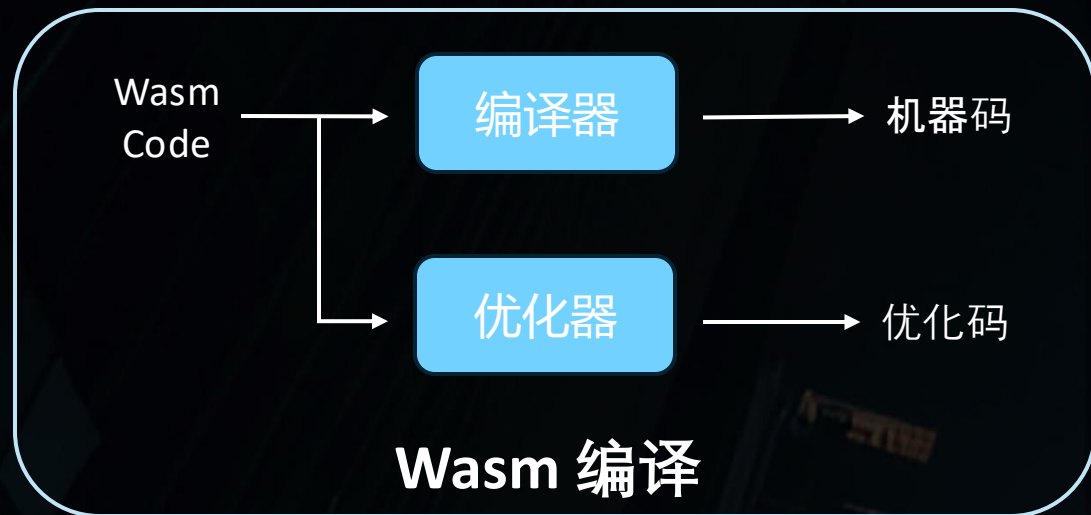
Wasm Code

Wasm workflow



- 在浏览器中，Wasm 的编译和执行由 JS 引擎驱动。
- Wasm 和 JS 在浏览器中共享同一个运行时环境。

Wasm workflow



就像汽车变速箱：

- 低阶编译器 = 低速档（快速起步，但速度有限）
- 高阶编译器 = 高速档（需要预热，但性能更强）
- 运行时监控 = 自动换挡（发现“热”代码时，切换到高阶优化）

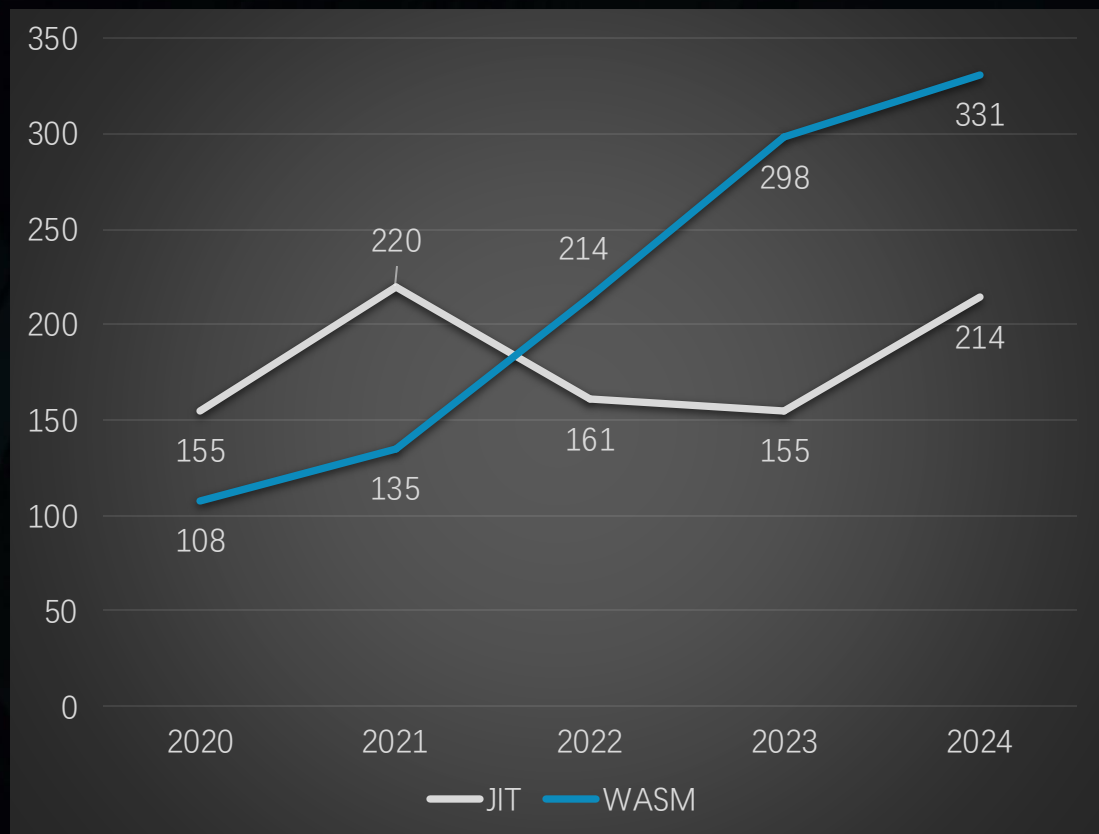
低速档



高速档

- Firefox: *Baseline, IonMonkey, Optimizing*
- Chrome: *Liftoff, Turbofan, Turboshift*
- Safari: *LLInt, BBQ, OMG*











JavaScript 引擎热点的转移



从 WebKit 的数据可以发现 Wasm 正取代 JIT 成为新的热点。

近年来 WebKit 在 JIT 编译器和 Wasm 编译器中的 git 提交次数

Wasm 已成为当前浏览器漏洞挖掘的热点

	Your browser										
Phase 5 - The Feature is Standardized											
JS BigInt to Wasm i64 Integration	✓	85	78	15 ^[i]	15.0	1.1.2	21.3	N/A	N/A	N/A	N/A
Branch Hinting	?	✗ ^[a]	✗ ^[f]	16	✗ ^[n]	✗ ^[v]	✗	✗	✗	✗	✗
Bulk Memory Operations	✓	75	79	15	12.5	0.4	23.0	1.0.0	0.20	1.0	1.0.30
Custom Text Format Annotations	?	N/A	N/A	N/A	N/A	N/A	N/A	N/A	✓	✓	N/A
Extended Constant Expressions	✓	114	112	17.4	21.0	1.33	✗ ^[ac]	N/A	25	✗	✗ ^[ao]
Garbage Collection	✓	119	120	18.2	22.0	1.38	✗	✗	✗ ^[aj]	✗	✗
Multiple Memories	✓	120	125	✗	22.0	1.38	✗ ^[ae]	N/A	15	✗	✗ ^[aq]
Multi-value	✓	85	78	13.1	15.0	1.3.2	22.3	1.0.0	0.17	1.0	1.0.24
Import/Export of Mutable Globals	✓	74	61	12	12.0	0.1	21.3	1.0.0	✓	0.7	1.0.1
Reference Types	✓	96	79	15	17.2	1.16	23.0	1.0.0	0.20	2.0	1.0.31
Relaxed SIMD	✓	114	✗ ^[h]	✗ ^[k]	21.0	1.33	✗	✗	15	✗	✗
Non-trapping float-to-int Conversions	✓	75	64	15	12.5	0.4	22.3	1.0.0	✓	✓	1.0.24
Sign-extension Operators	✓	74	62	14.1 ^[l]	12.0	0.1	22.3	1.0.0	✓	✓	1.0.24
Fixed-width SIMD	✓	91	89	16.4	16.4	1.9	24.1	1.1.0	0.33	2.0	1.0.33
Tail Call	✓	112	121	18.2	20.0	1.32	✗	1.0.0 ^[ag]	22 ^[ak]	✗	✗ ^[ar]
Typed Function References	✓	119	120	18	22.0	1.38	✗	✗	✗ ^[al]	✗	✗
Phase 4 - Standardize the Feature											
Exception Handling with exnref	✗	✗ ^[c]	131	18.4	✗ ^[p]	✗ ^[x]	✗	✗	✗	✗	✗ ^[an]
JS String Builtins	✓	130	134	✗	✗ ^[s]	2.1	✗	✗	N/A	N/A	N/A
Memory64	✓	133	134	✗	✗ ^[t]	✗ ^[aa]	✗ ^[ad]	N/A	✗ ^[aj]	✗	✗ ^[ap]
Threads	✓	74	79	14.1 ^[l]	16.4	1.9	✗ ^[af]	N/A	15	✗	✗

- 自 2017 年 11 月 Wasm 社区组 (CG) 发布最小可行产品 (MVP) 标准以来, 该技术栈始终处于动态演进状态。四大主流浏览器厂商 (Chrome、Edge、Firefox、Safari) 通过标准化进程持续推进特性扩展。

新特性的引入, 势必会带来潜在的漏洞风险!

如何发现 Wasm bugs?



About 2026 Symposium 2025 Symposium Previous Events

FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities



32ND USENIX
SECURITY SYMPOSIUM

ATTEND

PROGRAM

PARTICIPATE

SPONSORS

ABOUT

Conference

FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler



面临的问题

1. 如何让 fuzzer 生成结构正确的 wasm input ?
2. 如何检测 JIT 优化引起的错误 ?

“输入可执行”

“结果可验证”

如何生成有效的 Wasm 输入

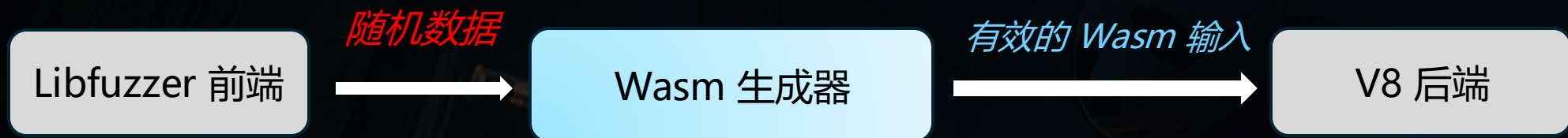
TITLE ▾	STATUS ▾	↓ LAST MODIFIED
Reconsider g_thread_in_wasm_code flag	Assigned	Mar 27, 2025 09:47
v8_wasm_compile_fuzzer: Abrt in heap::base::Worklist<v8::internal::Tagged<v8::internal::HeapObject>,	Duplicate	Mar 14, 2025 09:14
v8_wasm_compile_fuzzer: Abrt in v8::internal::MutablePageMetadata::ContainsAnySlots	Duplicate	Mar 12, 2025 08:24
v8_wasm_compile_fuzzer: Abrt in v8::internal::MutablePageMetadata::AllocateSlotSet	Duplicate	Feb 27, 2025 03:05
v8_wasm_compile_fuzzer: Crash in Builtins_JSToWasmWrapperAsm	Fixed	Feb 25, 2025 08:42
v8_wasm_compile_fuzzer: Abrt in v8::base::OS::SetPermissions	Duplicate	Feb 24, 2025 05:23
v8_wasm_compile_fuzzer: Abrt in heap::base::BasicSlotSet<4ul>::RemoveRange	Duplicate	Feb 20, 2025 12:57
v8_wasm_compile_fuzzer: Abrt in v8::internal::RootMarkingVisitor::VisitRootPointer	Duplicate	Feb 18, 2025 09:07
v8_wasm_compile_fuzzer: Abrt in v8::internal::Builtins::CallInterfaceDescriptorFor	Duplicate	Feb 17, 2025 08:56
v8_wasm_compile_fuzzer: Abrt in unsigned int v8::internal::ExternalEntityTable<v8::internal::TrustedPointerTable	Duplicate	Feb 17, 2025 08:52
v8_wasm_compile_fuzzer: Abrt in v8::internal::ExternalEntityTable<v8::internal::ExternalPointerTableEntry, 53687	Duplicate	Feb 17, 2025 08:49
v8_wasm_compile_fuzzer: Abrt in v8::internal::MarkingBitmap::IsClean	Duplicate	Feb 17, 2025 08:48
v8_wasm_compile_fuzzer: Abrt in v8::internal::Sweeper::RawSweep	Duplicate	Feb 17, 2025 08:45
v8_wasm_compile_fuzzer: Abrt in v8::internal::MemoryAllocator::Pool::ReleasePooledChunks	Duplicate	Feb 17, 2025 08:17
v8_wasm_compile_fuzzer: Abrt in heap::base::ActiveSystemPages::Add	Verified	Feb 15, 2025 05:08
v8_wasm_compile_fuzzer: Abrt in v8::internal::StubCache::Clear	Verified	Feb 14, 2025 05:17
Bump resource allocation of certain known-to-be-important fuzzers	Duplicate	Feb 14, 2025 01:27
v8_wasm_compile_fuzzer: CHECK failure: !code_space.is_empty()	Verified	Feb 13, 2025 08:21
v8_wasm_compile_fuzzer: DCHECK failure in src.is_byte_register() in assembler-ia32.cc	Verified	Feb 13, 2025 08:20

Chromium Issue Tracker

v8_wasm_compile_fuzzer 的产出惊人

"Attacking WebAssembly Compiler of WebKit." Black Hat Asia 2023

如何生成有效的 Wasm 输入



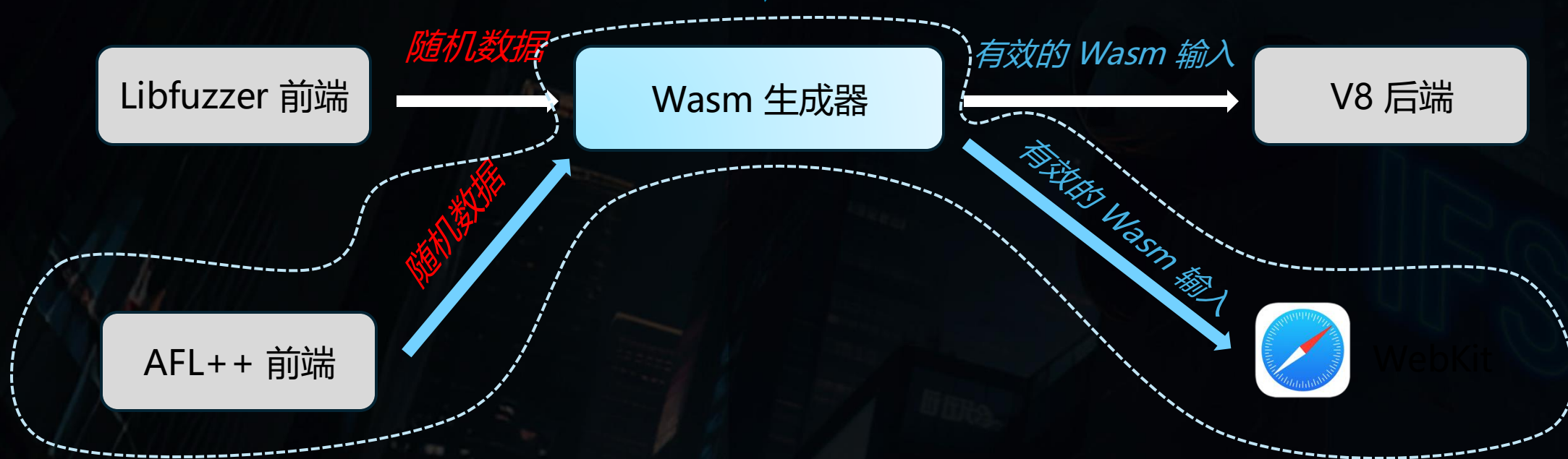
`v8_wasm_compile_fuzzer` 的工作流程

如何生成有效的 Wasm 输入



复用 `v8_wasm_compile_fuzzer`

将 `v8_wasm_compile_fuzzer` 的生成器编译成共享库
作为 AFL++ 的 Custom Mutator Library



如何检测 JIT 优化引起的错误

```
function foo(){  
  return Object.is(Math.expm1(-0),-0);  
}  
console.log(foo()); // true  
%OptimizeFunctionOnNextCall(foo);  
console.log(foo()); // false
```

现有的 Wasm Fuzzers:

- 主要使用 **崩溃** 作为指标。
- 这足够吗?

如何检测 JIT 优化引起的错误

```
function foo(){  
  return Object.is(Math.expm1(-0), -0);  
}  
console.log(foo()); // true  
%OptimizeFunctionOnNextCall(foo);  
console.log(foo()); // false
```

- $\text{Math.expm1}(x) = e^x - 1$
- `Object.is` 用于判断两个值是否是相同。
- 0 和 -0之间的细微差别会造成什么危害?
- 该漏洞已被证明可利用，且在网上公开。

JIT 优化错误可能不触发崩溃，但可能被利用!

主要挑战



1. 直接复用 `v8_wasm_compile_fuzzer` 效果差。

- 其他研究者包括厂商也在测试，难以挖到 Unique Bug

2. 现有 Fuzzer 主要以 *Crash* 作为 Bug 指标

- 忽略非崩溃 Bug

解决方案

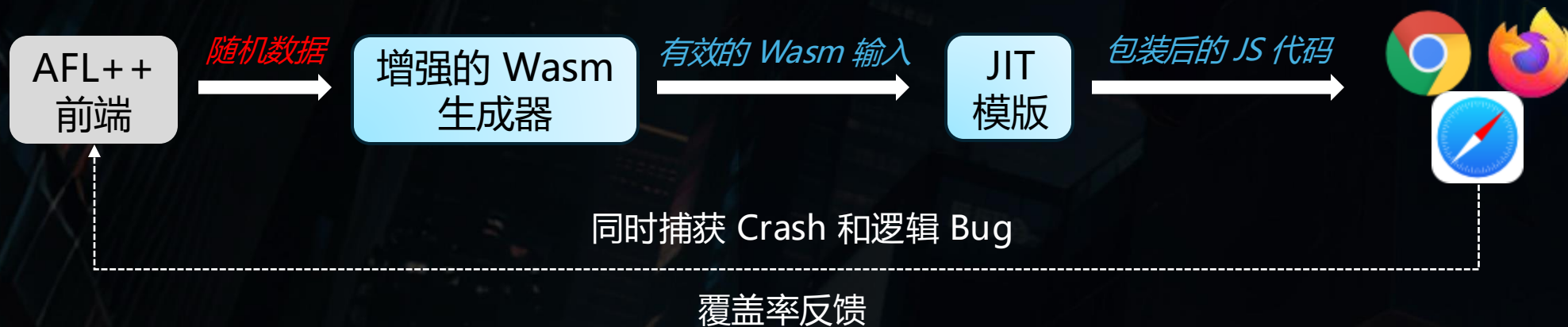


1. 利用 `v8::internal::wasm::fuzzing` APIs 构建差异化的 Fuzzer

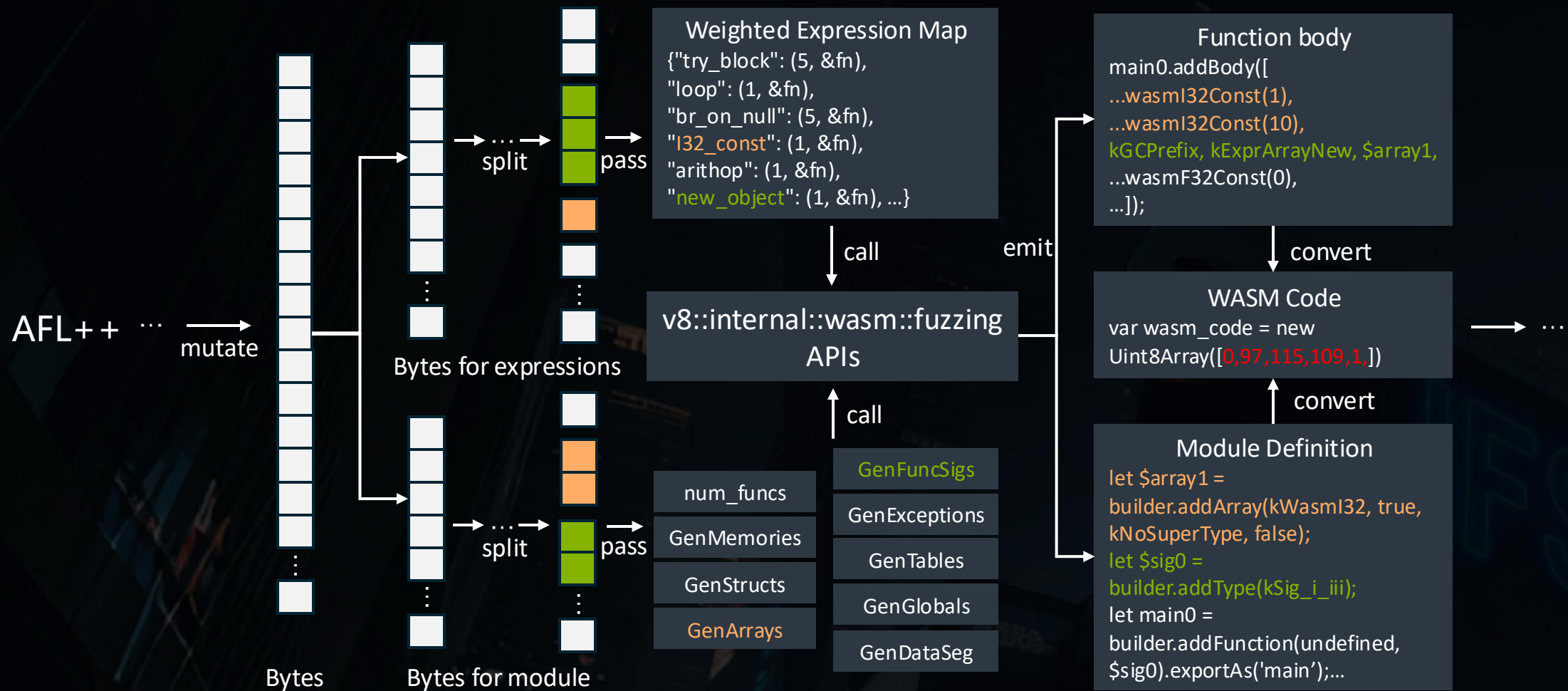
- 降低开发成本
- 自动继承 V8 对 Wasm 新特性的支持与更新

2. 设计 JIT 模版，来触发 JIT 并捕获非崩溃 Bug

- 同时捕获崩溃和非崩溃 Bug
- 灵活度高



增强的 Wasm 生成器



生成器将输入的字节与 Wasm 结构进行了细致的映射，确保有效利用输入字节。

用 JavaScript 模板封装 Wasm Code

... →
Generate

```
// JS 胶水代码
var wasmCode = new
  Uint8Array([0,97,115,109,1, ...]);
var wasmModule = new
  WebAssembly.Module(wasmCode);
var wasmInstance = new
  WebAssembly.Instance(wasmModule);
var opt = wasmInstance.exports.main;
function deepEquals(r1, r2){...}
...
```

```
r1 = opt(p); r2 = opt(p);
// 检查内在随机性
if( !deepEquals(r1, r2) ) return; ...
// 触发 JIT 优化
for(...){...; opt(p); ...}
// 捕捉崩溃和非崩溃错误
r3 = opt(p);
if( !deepEquals(r1, r3) ) quit(111);
```

→
Execute



JavaScript

左侧的胶水代码保证浏览器正确加载 Wasm，右侧的代码用于触发 JIT 并捕获差异。

用 JavaScript 模板封装 Wasm Code

// 胶水代码

```
var wasmCode = new Uint8Array([0,97,115,109,1,...]);  
var wasmModule = new WebAssembly.Module(wasmCode);  
var wasmInstance = new WebAssembly.Instance(wasmModule);  
var opt = wasmInstance.exports.main;
```

实例化 WASM 模块，导出主函数。

用 JavaScript 模板封装 Wasm Code

```
.....  
// 触发 JIT  
var opt = wasmInstance.exports.main;  
for (var i = 0; i < 10000; i++) {  
  opt(p);  
}
```



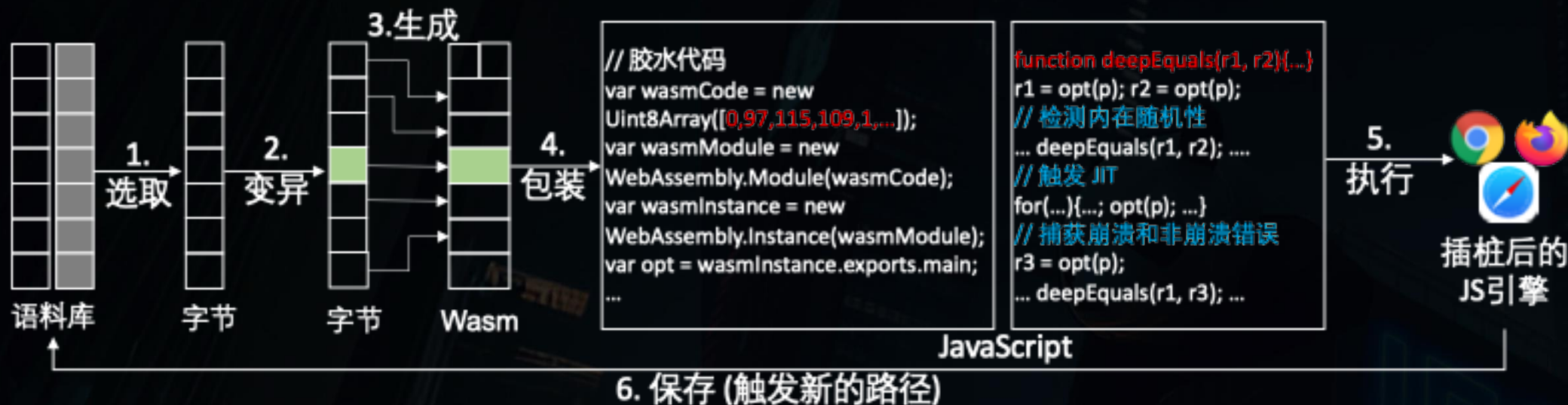
- 当某些 JavaScript 代码变得 **Hot**（即被执行的次数足够多）时，JIT 编译器就会被激活。
- JS 胶水代码将 Wasm Code 中的函数导出（opt），并在 for 循环中调用。
- for 循环的次数和时间由每个 JavaScript 引擎的优化条件决定。

用 JavaScript 模板封装 Wasm Code

```
.....  
var opt = wasmlInstance.exports.main;  
var beforeJIT = opt(p);  
for (var i = 0; i < 10000; i++) {  
  opt(p);  
}  
var afterJIT = opt();  
r3 = opt(p);  
// 对比结果, 捕获逻辑错误  
if( !deepEquals(beforeJIT, afterJIT) )  
quit(111);
```

- 比较 JIT 前和 JIT 后优化函数的返回值是否深度相等。
- 为了不把“合法但非确定性行为”误判成 Bug, 我们禁止生成某些 API:
 - grow_memory
 - table_grow
 -

Wasm Fuzzer 工作流程



1. 种子生成：从语料库中提取种子，并进行变异得到随机输入。
2. 生成器：将随机输入传递给生成器，映射成结构正确的 Wasm 模块。
3. JS 模版嵌入：将生成的 Wasm 模块嵌入到预先准备好的 JS 模版中。
4. 执行与反馈：使用浏览器的 JS 引擎解释执行生成的 JS 样本，并回传覆盖率信息，形成反馈循环。

Bugs Found

Apple Security Research

New Report

Search

Apple

Open

[WASM] IPInt lea

Fall 2025

Out-of-l JavaScr

Multiple WebAss

Improper Exception Handling Leads to Null Pointer Dereferenc...

UAF in JavaScriptCore's WebAssembly Compiler During...

OE

Addressed

In progress

XZ

Xiangwei Zhang

3 credited reports

Apple Product Security

发送至 我

OE

' - please include this ID in replies to this thread.

Hello Xiangwei,

Thanks for r

CVE-2024-5

Reporter

Xiangwei Zhang of Tencent Security YUNDING LAB

CVE-2025-1933: JIT corruption of WASM i32 return values on 64-bit CPUs

Impact high

Description

On 64-bit CPUs, when the JIT compiles WASM i32 return values they can pick up bits from left over memory. This can potentially cause them to be treated as a different type.

References

[Bug 1946004](#)

m Bugzilla

Search Bugs

0

e

ative

案例分析：CVE-2025-1933

```
0x5574bbc1052:    mov    $0x3,%eax
0x5574bbc1057:    mov    %eax,(%rdx)
0x5574bbc1059:    mov    $0x2,%eax
0x5574bbc105e:    pop    %rbp
0x5574bbc105f:    ret

IonMonkey
```

在一次函数调用过程中，一个经过 Wasm-Ion 编译的函数返回整数值 3，该值被存储在栈中。对应的汇编代码如左图

```
0x5574bb9106a:    lea    -0x28(%rbp),%rsp
0x5574bb9106e:    mov    %eax,%eax
0x5574bb91070:    mov    %eax,0x24(%rsp)
0x5574bb91074:    pop    %rax
0x5574bb91075:    jmp    0x5574bb9107b
0x5574bb9107a:    int3
0x5574bb9107b:    add    $0x20,%rsp
0x5574bb9107f:    pop    %rbp
0x5574bb91080:    ret

Baseline
```

函数返回后，在一个经过 Wasm-Baseline 编译的函数中，将这个值从栈中弹出到 rax 寄存器。



Excepted **0x3** !
Why **0x7ffd00000003** ?

案例分析: CVE-2025-1933

```
static bool GenerateJitEntry(MacroAssembler& masm, size_t funcExportIndex,
                           const FuncExport& fe, const FuncType& funcType,
                           const Maybe<ImmPtr>& funcPtr,
                           CallableOffsets* offsets) {
    // [...]
    // Store the return value in the JSReturnOperand.
    Label exception;
    const ValTypeVector& results = funcType.results();
    if (results.length() == 0) {
        GenPrintf(DebugChannel::Function, masm, "void");
        masm.moveValue(UndefinedValue(), JSReturnOperand);
    } else {
        MOZ_ASSERT(results.length() == 1, "multi-value return to JS unimplemented");
        switch (results[0].kind()) {
            case ValType::I32:
                GenPrintIsize(DebugChannel::Function, masm, ReturnReg);
                // No widening is required, as the value is boxed.
                masm.boxNonDouble(JSVAL_TYPE_INT32, ReturnReg, JSReturnOperand);
                break;
            case ValType::F32: {
                // [...]
            }
        }
    }
    // [...]
}
```

[0]

[0]: JIT 入口代码, 这里处理了函数返回值。

[1]: 这里在处理 I32 类型的返回值时, 默认 **ReturnReg** 中的高位已清零, 直接将 **ReturnReg** 传给了 **boxNonDouble** 函数

[2]: **boxNonDouble** 函数用于将非双精度类型的原始值装箱

“装箱”是指将原始值封装为引擎内部统一的 Value 格式, 使其能够参与动态类型操作。

src/js/src/wasm/WasmStubs.cpp

案例分析: CVE-2025-1933

```
// 将非 double 类型的值装箱为 JS Value
void boxNonDouble(JSValueType type, Register src, const ValueOperand& dest) {
    MOZ_ASSERT(src != dest.valueReg());
    boxValue(type, src, dest.valueReg());
}

void MacroAssemblerX64::boxValue(JSValueType type, Register src,
                                  Register dest) {
    MOZ_ASSERT(src != dest);
    // 调试模式下验证 32 位值的高位是否为零 (仅限 INT32/BOOLEAN)
    #ifdef DEBUG
        if (type == JSVAL_TYPE_INT32 || type == JSVAL_TYPE_BOOLEAN) {
            Label upper32BitsZeroed;
            movePtr(ImmWord(UINT32_MAX), dest);
            asm().branchPtr(Assembler::BelowOrEqual, src, dest, &upper32BitsZeroed);
            breakpoint();
            bind(&upper32BitsZeroed);
        }
    #endif
    mov(ImmShiftedTag(type), dest); // 将类型标签左移 47 位后存入 dest
    orq(src, dest); // 使用按位或操作将 src 与标签合并成一个值
}
```

举个例子, 现要将 int32 值 3 装箱

1. ReturnReg: rax 存储原始值 $0x3$
2. 类型标签: **JSVAL_TYPE_INT32** ($0x01$) 。
3. 装箱操作: 结果为 **$0x80000000000003$** , 表示一个合法的 int32 类型 Value。

$0x01 \ll 47 = 0x8000000000000000$

Value: $0x3 \mid 0x8000000000000000 = 0x8000000000000003$

那么当 ReturnReg 不符合预期时会发生什么?

案例分析: CVE-2025-1933

```
// 将非 double 类型的值装箱为 JS Value
void boxNonDouble(JSValueType type, Register src, const ValueOperand& dest) {
    MOZ_ASSERT(src != dest.valueReg());
    boxValue(type, src, dest.valueReg());
}

void MacroAssemblerX64::boxValue(JSValueType type, Register src,
                                  Register dest) {
    MOZ_ASSERT(src != dest);
    // 调试模式下验证 32 位值的高位是否为零 (仅限 INT32/BOOLEAN)
#ifdef DEBUG
    if (type == JSVAL_TYPE_INT32 || type == JSVAL_TYPE_BOOLEAN) {
        Label upper32BitsZeroed;
        movePtr(ImmWord(UINT32_MAX), dest);
        asm().branchPtr(Assembler::BelowOrEqual, src, dest, &upper32BitsZeroed);
        breakpoint();
        bind(&upper32BitsZeroed);
    }
#endif
    mov(ImmShiftedTag(type), dest); // 将类型标签左移 47 位后存入 dest
    orq(src, dest); // 使用按位或操作将 src 与标签合并成一个值
}
```

举个例子, 现要将 int32 值 3 装箱

1. ReturnReg: rax 存储原始值 $0x3$ 。
2. 类型标签: **JSVAL_TYPE_INT32** ($0x01$) 。
3. 装箱操作: 结果为 **$0x8000000000000003$** , 表示一个合法的 int32 类型 Value。

当 ReturnReg 中的值为预期外的 **$0x7ffd000000000003$**

$0x7FFD000000000003 \mid 0x0000800000000000 = 0xffffd00000000003$

高位未置零, 导致类型标签变成非法值。

不合法的 Value 表示!

案例分析: CVE-2025-1933

🤔 如何利用?

```
enum JSValueType : uint8_t {
    JSVAL_TYPE_DOUBLE = 0x00,
    JSVAL_TYPE_INT32 = 0x01,
    JSVAL_TYPE_BOOLEAN = 0x02,
    JSVAL_TYPE_UNDEFINED = 0x03,
    JSVAL_TYPE_NULL = 0x04,
    JSVAL_TYPE_MAGIC = 0x05,
    JSVAL_TYPE_STRING = 0x06,
    JSVAL_TYPE_SYMBOL = 0x07,
    JSVAL_TYPE_PRIVATE_GCTHING = 0x08,
    JSVAL_TYPE_BIGINT = 0x09,
#ifdef ENABLE_RECORD_TUPLE
    JSVAL_TYPE_EXTENDED_PRIMITIVE = 0x0b,
#endif
    JSVAL_TYPE_OBJECT = 0x0c,

    // This type never appears in a Value; it's only an out-of-band value.
    JSVAL_TYPE_UNKNOWN = 0x20
};
```

合法的 BIGINT Value

装箱过程: $0xdeadbeef \mid 0x48000000000000 = 0x48000deadbeef$

拆箱过程: $0x48000000000000 \gg 47 = 0x09$ $0x00000deadbeef$

类型

值



利用高位垃圾数据破坏类型标签, 使引擎误判为其他类型。

当 ReturnReg 中的值为 Int32 **0xdeadbeef**
但是高位未清零, 导致原始值为 **0x48000deadbeef**

会发生什么?

装箱

$0x48000deadbeef \mid 0x800000000000 = 0x48000deadbeef$

JSVAL_TYPE_INT32 << 47

JSVAL_TYPE_BIGINT

类型混淆!

拆箱

$0x48000000000000 \gg 47 = 0x09$

$0x00000deadbeef$

案例分析: CVE-2025-1933

Patch:

```
--- a/js/src/wasm/WasmStubs.cpp
+++ b/js/src/wasm/WasmStubs.cpp
@@ -1230,17 +1230,19 @@ static bool GenerateJitEntry(MacroAssemb
 if (results.length() == 0) {
     GenPrintf(DebugChannel::Function, masm, "void");
     masm.moveValue(UndefinedValue(), JSReturnOperand);
 } else {
     MOZ_ASSERT(results.length() == 1, "multi-value return to JS unimplemented");
     switch (results[0].kind()) {
     case ValType::I32:
         GenPrintIsize(DebugChannel::Function, masm, ReturnReg);
         // No widening is required, as the value is boxed.
-        #ifdef JS_64BIT
+        masm.widenInt32(ReturnReg);
+    #endif
         masm.boxNonDouble(JSVAL_TYPE_INT32, ReturnReg, JSReturnOperand);
         break;
     case ValType::F32: {
```

装箱前高位清零

64 位系统上, 对于 int32 类型的值, 没有正确处理高 32 位。

如果高位留下可控的垃圾数据, 就能控制 Value 类型进行 **类型混淆**, 用于构造后续利用原语, 进而实现任意代码执行。

堆喷

未来展望



谢谢



IFS