doi:10.3772/j.issn.1002-0470.2023.03.003

JavaScript 引擎 JIT 代码的类型混淆缺陷检测器①

孙力立② 张培华 武成岗③ 王 喆

(处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院大学计算机科学与技术学院 北京 100190)

摘 要 类型混淆漏洞是近期在 JavaScript 引擎中集中爆发的一类漏洞。但是,受即时编译(JIT)代码的限制,以往的类型混淆缺陷的检测方法,无法用于检测 JavaScript 引擎 JIT 代码的类型混淆缺陷。本文提出了一种针对该类型缺陷的检测方法,并实现了检测器名为 TC-JIT-San 的 JIT 代码类型混淆缺陷检测器。该方法利用 JIT 代码执行流和数据类型之间关联性,将从 JIT 代码中识别数据类型的难题转为观察执行流的变化。TC-JIT-San 通过观察执行流的变化情况是否符合正常执行逻辑,从而检测出类型混淆缺陷。实验结果表明,TC-JIT-San 具有低开销、低漏报和误报的特点。其运行时开销是正常执行的 1.84 倍,平均漏报率和误报率为 0%和0.11%。

关键词 漏洞挖掘;动态检测器;软件缺陷检测

0 引 言

JavaScript 执行引擎是浏览器、PDF 阅读器、Flash 播放器等软件的核心组件,其中的漏洞具有巨大的安全影响。主流的 JavaScript 执行引擎包含即时编译(just-in-time,JIT)器,其会对频繁执行的 JavaScript 代码进行编译优化,生成 JIT 代码。而这些动态生成的 JIT 代码中可能包含漏洞。其中,类型混淆就是新兴的一种。

类型混淆缺陷[1]是指使用不兼容的类型对已分配的资源(指针、对象、变量)进行访问。研究人员设计各种动态检测器[2](sanitizer)来提升模糊测试[3]发现软件缺陷的能力。但是,现有的类型混淆检测器如 CaVer[4]、TypeSan[5]和 HexType[6]都无法检测 JIT 代码中的类型混淆。这是因为,上述工作所检测的缺陷成因和引擎中 JIT 代码的类型混淆缺陷的成因不同,先前的检测方法无法应用于后者。

设计针对引擎 JIT 代码中类型混淆缺陷的动态检测器的最大难点在于:JIT 代码中缺少类型信息。因此,无法直接基于数据对象的类型信息,实现对类型混淆缺陷的检测。本文对 JavaScript 执行引擎中 JIT 编译器的设计实现进行了剖析,分析其导致类型混淆缺陷的成因,总结出一种 JavaScript 引擎 JIT 代码中类型混淆缺陷的触发模式,并且发现可以利用 JIT 代码的执行路径和其所处理的数据类型间的关联关系,将从 JIT 代码中识别数据类型的难题转为观察 JIT 代码执行流的变化。由此,本文提出了一种基于 JIT 代码执行流的类型混淆缺陷检测方法。该方法使用二进制插桩技术向 JIT 代码中插入检测逻辑,通过观察在 JavaScript 对象的数据类型发生了改变后,JIT 代码的执行流是否存在有违正常执行逻辑的情况,从而发现其中潜藏的类型混淆缺陷。由于 ChakraCore[7]引擎中 JIT 代码的类型混淆漏洞最为突出[8],本文为其实现了一个 JIT 代码的类型混淆缺陷检测器:TC-JIT-San。将该工具应用于测试

① 国家自然科学基金(U1736208,61902374)资助项目。

② 女,1992 年生,博士生;研究方向:系统与软件安全;E-mail:sunlili@ict.ac.cn。

③ 通信作者,E-mail:wucg@ict.ac.cn。

(收稿日期:2022-01-12)

集和模糊测试中,都获得了很好的检测效果。

本文的主要贡献包含如下几点。

(1)基于对 JavaScript 引擎 JIT 编译器设计机制的剖析,总结出一种 JavaScript 引擎 JIT 代码中类型混淆缺陷的触发模式。

(2)提出了一种针对上述模式的类型混淆缺陷的检测方法。针对难以从 JIT 代码中提取数据类型信息这一难题,该方法利用 JIT 代码的执行流和数据类型的关联性,通过观察数据类型改变后执行流的变化情况,推断是否存在类型混淆缺陷。

(3)提出了一种利用引擎提供的转储数据识别 JIT 代码基本块语义的方法。

(4)实现了一个 ChakraCore 引擎的 JIT 代码类型混淆缺陷检测器——TC-JIT-San。实验展示了 TC-JIT-San 检测器的有效性和实用性:其具有低漏报、低误报和低开销的特点。该检测器在测试集上的平均漏报率为 0%,误报率为 0.11%,平均运行时开销为 1.84 倍。将该检测器应用于模糊测试中,成功检测出 ChakraCore 1.10.0 中的 1 个未公开 PoC 的类型混淆漏洞。

1 研究背景

本节将对本文涉及到的背景知识与相关研究技术进行介绍,主要包括类型混淆缺陷的动态检测器的相关研究和 JavaScript 引擎中 JIT 代码的类型混淆漏洞的原理分析。

1.1 类型混淆缺陷的动态检测器相关研究

类型混淆缺陷可以分为两类:一类是 C++语言层类实例化对象的类型混淆,另一类是应用逻辑层面的类型混淆。当前针对类型混淆缺陷的动态检测器,如 CaVer、TypeSan 和 HexType,都是面向前者提出解决方案。这些工作借助编译器,静态分析源码中的类层级结构,构建类型层级表(Type Hierarchy Table);而后,在数据对象的创建点进行插桩,记录数据对象的类型信息;在数据对象的类型转换点进行插桩,从而检测数据对象的目标转换类型是否符合类型约束。

然而,JavaScript 引擎中的类型混淆漏洞大都属于应用逻辑层面的类型混淆,且 JIT 代码中的类型混淆漏洞占有相当的比重。以 ChakraCore 为例,依据其 2017—2019 年的漏洞数据,JIT 代码的类型混淆占引擎类型混淆漏洞数目的平均占比为72.58%。由于漏洞成因的不同,先前的方法无法用于检测 JavaScript 引擎中的类型混淆漏洞。

1.2 JavaScript 引擎中 JIT 代码的类型混淆漏洞原理

本节首先简要介绍 ChakraCore 引擎类型系统的相关知识,然后从 JavaScript 引擎 JIT 编译器的实现机制说明 JIT 代码的类型混淆漏洞的原理。

1.2.1 ChakraCore 引擎类型系统

出于减少内存开销、提升执行性能的考量,主流的 JavaScript 引擎在实现 JavaScript 内建对象时,会设计多种内部类型。例如 Array 对象,在 ChakraCore 引擎中就对应了 JavascriptArray 类、JavascriptNativeIntArray 类、JavascriptNativeFloatArray 类等多种实现。引擎会依据执行的 JavaScript 代码,选择合适的类进行实例化,并自动地按需调整实例化对象的类型。例如,当引擎执行“arr = [1,2,3];”时,由于 arr 的元素全为整型,其会使用 JavascriptNativeIntArray 类创建对象。该类专用于存储元素为 int 类型的数组,元素值按照 32 位整型格式存储在对应的数据单元中。随后,如果执行“arr[0] = 1.1;”,由于元素类型不再统一为整型,引擎会将 arr 对象实例原地转换为 JavascriptNativeFloatArray 类型。后一类型中的元素值均以 IEEE-754[9] 浮点表示方式进行存储。本文将引擎中实现的这种内部类型间原地转换的操作称为类型迁移操作。之后,如果又执行了“arr[0] = ‘hello’;”,引擎会自动将 JavascriptNativeFloatArray 对象迁移为通用的 JavascriptArray 类型。JavascriptArray 中可以存储任意类型的元素,为了区分其所存储的元素的类型,元素的高位比特会被用作类型标记。JavascriptArray 中的元素在存入/取出前,需要对原本的数值进行装箱(boxing)/拆箱(unboxing)操作,即对数值的高位比特添加/删除类型标记。在上述例子中,arr 对象发生了 2 次类型迁移,但在实际应用中,数组元素的类型在创建时刻通常已经确定了。因此,这种基于存储元素类型的优化设计会减少运行时间和内存开销。

1.2.2 JIT 编译器实现与类型混淆漏洞

JavaScript 是动态类型语言,其数据对象的类型动态可变,因此在运行时才确定具体的数据类型。如图 1 所示,解释器在解释执行“arr[0] =10”时,需要基于 arr 的实际类型,选择恰当的处理方式实现元素赋值。因此,解释器的实现逻辑需要枚举所有可能的数据类型。

为了提升运行效率,对于频繁解释执行的循环和函数体,JavaScript 引擎会对其做 JIT 编译。由 JIT 编译器生成一段二进制代码,即 JIT 代码。JIT 编译器使用类型投机优化策略[10]:其会基于解释执行阶段所收集的数据类型,投机地假设数据的类型保持不变,生成更为简单的处理逻辑(如图 1 右侧所示)。JIT 编译器会生成 bailout[11]代码,来处理投机失败的情况。其通过跳伞机制,确保在执行情况与预设条件不匹配时,控制流可以从 JIT 代码安全地转移到解释器继续执行。

但是,当类型投机优化与 JIT 编译器的其他优化子模块共同生效时,有可能存在逻辑错误,导致生成存在类型混淆问题的 JIT 代码。如图 2 所示,当 JIT 编译器处理前后 2 个对相同对象的元素赋值操作时,依据类型投机优化的处理逻辑,JIT 编译器需要生成 2 份对相同数据对象的类型检查操作。如果存在其他优化子模块判定后一检查冗余,JIT 编译器则会删除此检查,生成如图 2 右侧所示的处理逻辑。然而,在前一检查操作完成后,即便数据指针仍指向原先的对象,该对象的类型也有可能发生改变,例如发生类型迁移。由于后续的操作却依据先前假设的数据类型实现,从而触发类型混淆漏洞。

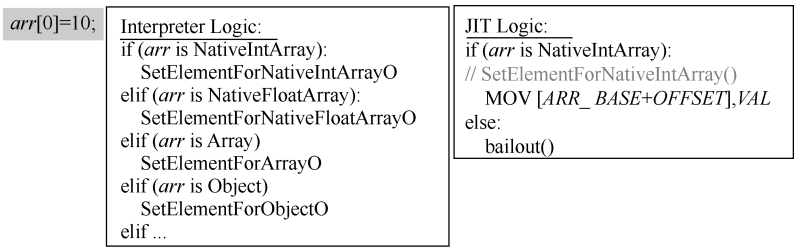


图 1 解释器与 JIT 代码执行“arr[0] =10”操作

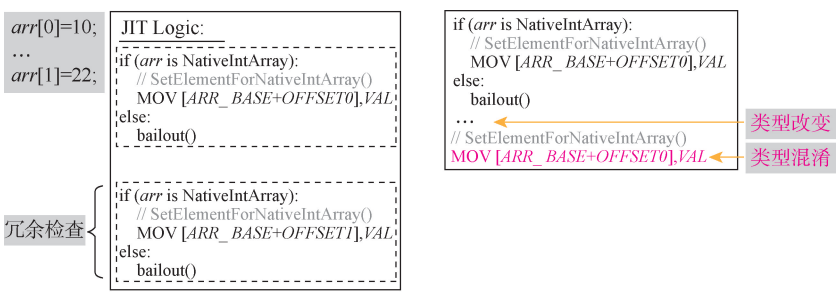


图 2 JIT 代码中类型混淆漏洞触发原理

2 挑战和解决思路

基于上节对 JIT 编译器中类型混淆漏洞触发原理的分析,本文总结出一种 JavaScript 引擎中 JIT 代码的类型混淆缺陷的触发模式,即当 JIT 代码执行过程满足条件:(1)某个 JavaScript 对象发生类型迁移,(2)该对象的后续访问操作仍依据旧的内部类型时,则可判定存在类型混淆缺陷。

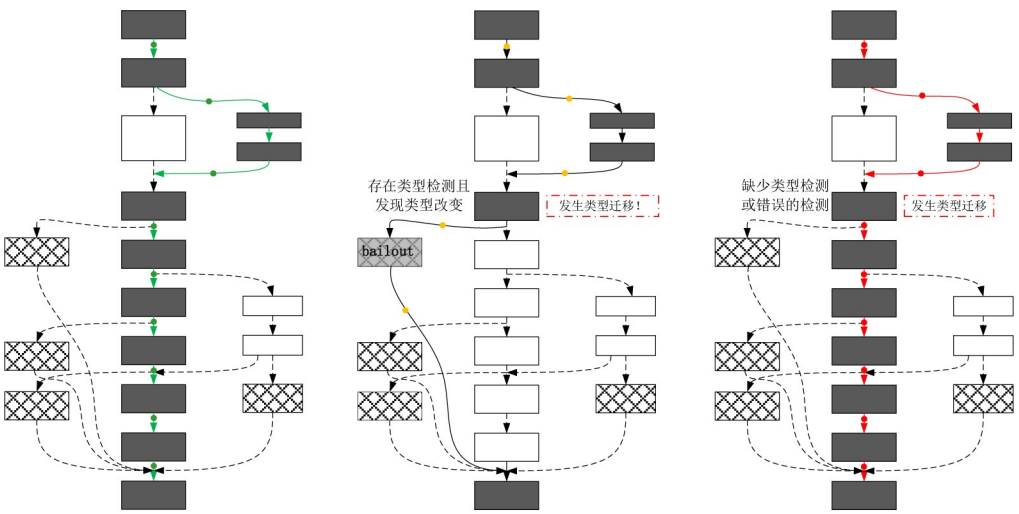
条件(1)中的类型迁移操作在引擎源码中对应 C++实现的方法,其方法名和代码都有明显的特征。例如方法名中存在“Convert”、“Adjust”等关键词,其会修改“type”、“kind”等数据。此外,这些方法会遍历内建对象的每个元素,并对其进行类型转

换。利用上述特征,通过人工分析源码便可以识别出类型迁移方法。通过对这些类型迁移方法进行插桩,一旦 JIT 代码在执行中调用到该方法,桩函数就会记录下具体的类型迁移类别。

条件(2)中对数据对象的访问操作体现在 JIT 代码中是一系列访存指令序列。由于指令层面缺少数据类型信息,因此难以直接提取数据的内部类型。针对这一挑战,本文提出了一种解决方法:利用 JIT 代码的执行路径和其所处理的数据类型间的关联关系,将从 JIT 代码中识别数据类型的难题转为观察 JIT 代码执行流的变化。

2.1　初始检测思路

本研究工作分析了不同情况下的 JIT 代码执行过程(见图 3),发现数据类型和代码执行路径存在关联性。按照正常的执行逻辑,当输入数据满足预设条件时,控制流会执行一条满足其生成初衷的执行路径(见图 3(a)中深色标识的基本块构成的执行流),即预设路径。JIT 代码首次执行的路径通常就是预设路径。后续 JIT 代码执行中,假使输入数据的内部类型发生改变,依据正常处理逻辑,JIT 代码的执行路径也会改变。这是因为 JIT 代码中的比较指令会检测出数据类型的改变,进而触发跳伞机制(见图 3(b)中深色标识的基本块构成的执行流)。但是,如果输入数据的内部类型发生了改变,而 JIT 代码中的执行路径仍然与预设路径相同(见图 3(c)中深色标识的基本块构成的执行流),这会导致类型迁移后的数据访问操作仍按照旧类型进行,也意味着存在类型混淆缺陷。由此,本文提出了一种检测思路:一旦观察到 JIT 代码当前执行路径触发了预设路径不曾发生的类型迁移事件,且当前执行路径仍与预设路径保持一致,则应报出类型混淆缺陷警报。



(a)预设条件下的JIT代码执行路径　(b)触发类型迁移操作的JIT代码执行路径(正常)　(c)触发类型迁移操作的JIT代码执行路径(异常)

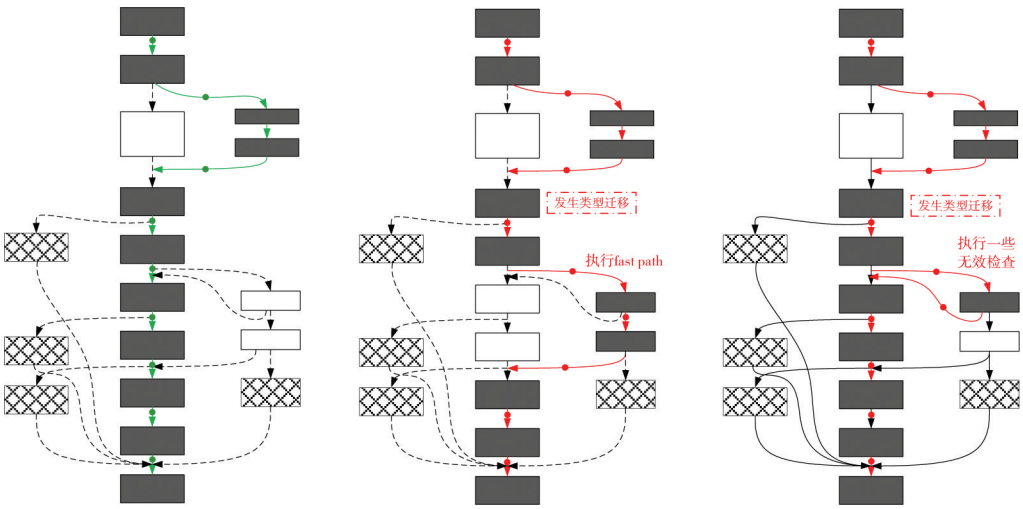
图 3　不同情况下的 JIT 代码执行路径

然而,在现实的类型混淆漏洞触发案例中(如图 4 所示),也存在发生类型迁移事件后,致错路径和预设路径不同的情况。因此,上述方法会导致漏报。为此,本研究对这些案例进行分析,理解其执行路径存在差异的原因。

JIT 代码的执行路径由若干实现不同 JavaScript 语义的指令序列构成。JIT 编译器往往会为同一 JavaScript 语义生成多份指令序列。其中,优化程度最高的指令序列称为快路径(fastpath);优化程度较低的指令序列称为慢路径(slowpath)。如图 4(b)所示,当 JIT 代码使用快路径替换原先的慢路径来实现相同的语义时,会导致执行路径与预设路径不同。慢路径使用调用指令调用引擎中的 C ++ 函数来实现相应的语义功能。C ++ 函数内通常会依据数据

类型选择合适的数据访问方式。但是在快路径中,基于类型投机优化生成的指令序列,会依据旧类型进行数据访问,因此很可能触发类型混淆缺陷。

JIT 编译器为某一 JavaScript 语义生成的指令序列不仅仅会实现该语义功能,也会做一些检查操作。JIT 代码在实际执行时,会按需执行检查操作。在图 4(c)中,JIT 代码多做了一些检查操作,导致执行路径与预设路径不同。但语义功能实现操作的执行并未改变,说明当前路径还是依据旧类型进行数据访问,因此触发类型混淆缺陷。



(a)预设条件下的JIT代码执行路径　(b)触发类型迁移操作的JIT代码执行路径1　(c)触发类型迁移操作的JIT代码执行路径2

图 4　触发类型混淆缺陷的 JIT 代码执行路径

2.2　改进版检测思路

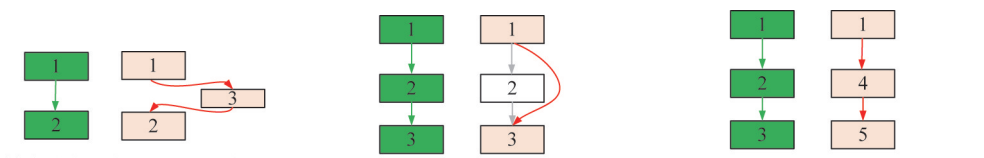
基于上述分析,本文提出了一种改进版检测方法。当检测器发现当前执行路径与预设路径不同时,其需要结合执行路径的语义信息,进一步推断是否应该报出警报。

为了方便引擎开发者调试,引擎通常会提供“-dump”选项,输出 JavaScript 代码和 JIT 代码指令序列之间的映射关系,即转储数据。转储数据可用于辅助识别 JIT 代码中指令序列的语义。图 5 展示了 ChakraCore 引擎转储数据的部分片段。其中,“源码”标识的是 JavaScript 源码;“本地码”标识的是 JIT 代码指令序列。同一段源码,可能对应多段 JIT 代码指令序列。通过解析转储数据,可以按照 JavaScript 源码语义对 JIT 代码指令序列进行分类。例如,通过将图 5 中指令序列的基本块都标记为同一语义编号,表示其标识着“o.a = value;”语义功能。

在对指令执行序列按语义分类后,还需推断:在实现相同源码语义时,预设路径和当前路径的执行差异到底是由检查操作导致的,还是由快慢路径导致的。路径的执行差异体现为不同的基本块覆盖情况,具体可细分为增、缺、替换 3 类(见图 6)。如果当前执行路径与预设路径相比,在实现同一源码语义功能时,仅存在增加或缺失基本块的情况(见图 6(a)和(b)),可以推断出:存在差异的基本块是在进行检查操作。这是因为,指令序列中真正完成语义功能的指令仅为少数,而基本块的增加或缺失,并未影响源码语义的实现,说明真正实现语义功能的指令就在 2 条路径共同覆盖的基本块中。而对于存在基本块替换的情况而言(图 6(c)),可以保守地认为所有替换/被替换的基本块都在完成真正的语义功能,其影响着数据的访问方式。检测器需要进一步观察这些基本块的内部特征,将包含调用指令的指令序列判定为慢路径;否则,则判定为快路径。

```
Line 21: o.a = value;                                                                    源码
Col  5: ^
...
26CE5E412D1  s104(rax).i64 = MOV   [s12<s78>(r14)[UninitializedObject].var+8].i64  Func # (#1.2), #3
26CE5E412D5  s106(rcx).i32 = XOR   s106(rcx).i32, s106(rcx).i32          Func # (#1.2), #3
26CE5E412D7  CMP     s104(rax).i64, [s105(rbx).u64 <0x0264E5BD6860 (&GuardValue)>].u64  Func # (#1.2), #3
26CE5E412DA  JNE     $L11                                Func # (#1.2), #3          本地码
26CE5E412E0  s12<s78>(r14)[UninitializedObject].var = CMOVNE  s106(rcx).u64  Func # (#1.2), #3
26CE5E412E4  $L12:                                       Func # (#1.2), #3
26CE5E412E4  [s12<s78>(r14)[UninitializedObject].var+16].i64 = MOV  s34(r15)[UninitializedObject].var  Func # (#1.2), #3
...
Line 21: o.a = value;                                                                    源码
Col  5: ^
...
26CE5E416DD  $L11: [helper]                                  Func # (#1.2), #3
26CE5E416DD  arg2(s110)(rdx).u64 = MOV  0x0264E5BD6860 (TypeCheckGuard).u64  Func # (#1.2), #3
26CE5E416E7  arg1(s111)(rcx).i64 = MOV  s107(rax).i64                Func # (#1.2), #3
26CE5E416EA  s112(rax).u64 = MOV   CheckIfTypeIsEquivalent.u64
26CE5E416F4  s12<s78><-48>.var = MOV  s12<s78>(r14).var
26CE5E416F8  s34<-56>.var = MOV    s34(r15).var                                          本地码
26CE5E416FC  s109(rax).u8 = CALL   s112(rax).u64                     Func # (#1.2), #3
26CE5E416FF        TEST       s108(rax).u8, s108(rax).u8             Func # (#1.2), #3
26CE5E41701        JEQ        $L10                                   Func # (#1.2), #3
26CE5E41703        JMP        $L12
26CE5E41708
```

图 5　转储数据将 JavaScript 源码和 JIT 代码相映射



(a) 当前执行路径覆盖的基本块包含预设执行路径所覆盖的基本块

(b) 预设执行路径覆盖的基本块包含当前执行路径所覆盖的基本块

(c) 当前执行路径覆盖的基本块和预设执行路径所覆盖的基本块相互存在差集

图 6　预设执行路径和当前执行路径完成同一语义操作的不同基本块覆盖情况

如果路径的执行差异仅由检查操作导致，说明执行操作仍依据旧的内部类型，因此需要报出警报。如果路径的执行差异由快慢路径导致，且当前执行的是快路径，也应报出警报。

3　设计实现

3.1　设计架构图

基于第 2 节所述的解决思路，本文实现了一个针对 JavaScript 引擎 JIT 代码的类型混淆缺陷检测器：TC-JIT-San。TC-JIT-San 的核心逻辑和数据定义都封装在一个自定义的运行时库中。其中，最为关键的是 AddTypeConfusionChecker( ) 函数和 UpdateStatus( ) 函数。TC-JIT-San 分别使用 LLVM 插桩和二进制代码插桩，向 JIT 代码嵌入检测逻辑。

在引擎编译阶段，需要使用 LLVM 插桩在 JIT 代码生成模块插入对库函数 AddTypeConfusionChecker( ) 的调用。此外，LLVM 插桩也需要在类型迁移方法中插入对 UpdateStatus ( ) 函数的调用。UpdateStatus( ) 函数会更新状态向量。状态向量长度为 3，分别对应整数数组转为通用数组、浮点数组转为通用数组和对象内联布局转为通用布局这 3 种类型迁移操作。

在引擎运行阶段，一旦 JIT 代码生成完成，引擎的控制流就会转移到 AddTypeConfusionChecker( ) 函数。AddTypeConfusionChecker( ) 函数会对 JIT 代码进行反汇编、构建控制流图 (control flow gragh，CFG)，识别基本块的语义，最后将检测逻辑通过二进制重写的方式注入原先的 JIT 代码 (见图 7 左半部分)。

在 JIT 代码执行阶段 (见图 7 右半部分)，插入的检测逻辑在起始基本块执行前会创建状态向量和基本块覆盖集合；在普通基本块执行前会更新基本块覆盖集合；在终止基本块执行前会检查当前的状

态向量和基本块覆盖情况,判定是否应该报出类型混淆错误。

3.2 反汇编和控制流图构建

TC-JIT-San 使用递归式的反汇编方法,该方法在反汇编 JIT 代码的同时构建控制流图。TC-JIT-San 使用 Capstone[12] 反汇编工具进行线性反汇编。基于所构建的控制流图,可以识别出所有的起始基本块和终止基本块。此外,还可以识别出和跳伞机制相关的基本块。其指令模式具有明显特征:在调用跳伞机制处理函数后,紧接着跳转到出口基本块。

3.3 JIT 代码重写

为了植入检测逻辑,TC-JIT-San 将 JIT 代码基本块的起始指令改写为一条 jmp 指令(见图 8),使得控制流先转移到一段跳板指令序列(trampoline)。在跳板指令序列中,首先会保存当前上下文,然后准备参数,再调用实现检测逻辑的库函数。函数返回后,跳板指令序列还需要恢复上下文,并对先前因被改写为 jmp 指令而影响到的指令(被改写指令)进行补偿执行。最后,再跳转回被改写指令的后续指令继续执行。

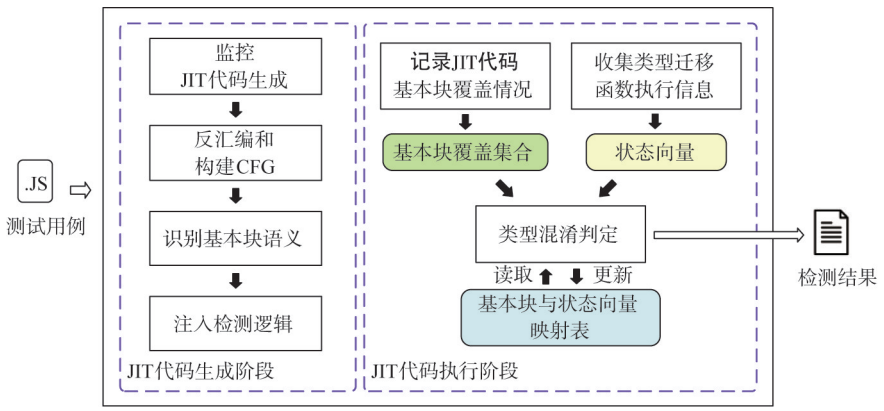


图 7 TC-JIT-San 检测器的设计架构图

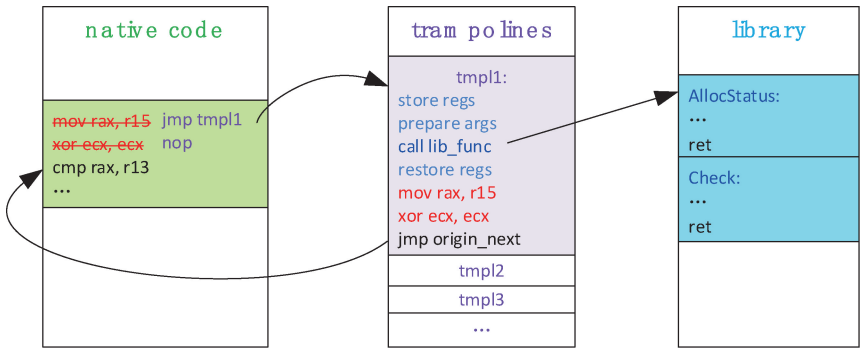


图 8 修改 JIT 代码指令

当基本块的大小不足以写一条 jmp 指令时,TC-JIT-San 使用 1 byte 长的 int3 软中断指令改写原先的指令。通过预先注册好的 sigtrap 信号处理函数,实现对检测逻辑库函数的调用。这种做法需要陷入内核,会引入较大开销。所以,仅在基本块大小受限时,TC-JIT-San 才采用该策略。

4 实验设计和结果分析

4.1 实验设定

研究工作收集了近年来已公开的 ChakraCore 的 JIT 代码类型混淆漏洞 PoC[8,13] 作为测试集 T1,用于进行漏报实验。

研究工作从 ChakraCore 回归测试集中随机抽取测试用例,共计 3 组,每组 300 个,作为测试集 T2;从 ECMA-262 测试集中随机抽取测试用例,共计 3 组,每组 300 个,作为测试集 T3。T2 和 T3 用于进行误报实验。

研究工作从 ChakraCore 回归测试集中随机抽取测试用例,共计 3 组,每组 500 个,作为测试集 T4;从访问热度最高的前 50 网站[14]中爬取 JavaScript 代码,获得 222 个测试用例,作为测试集 T5;T4 和 T5 用于进行运行时开销实验。

此外,实验将 TC-JIT-San 改进版应用于 JavaScript 引擎的模糊测试中,对自动生成的测试用例进行检测。

4.2 检测器有效性分析

4.2.1 对已知漏洞的检测(漏报实验)

实验使用 T1 评估 TC-JIT-San 初始版和改进版对已知漏洞的检测能力,以此评估 TC-JIT-San 的漏报情况。由于漏洞对应的 ChakraCore 版本不同,TC-JIT-San 会对存在漏洞的不同版本的 ChakraCore 进行插桩,然后检测。实验数据如表 1 所示。

表 1 TC-JIT-San 对已知漏洞的检测情况

CVE	ChakraCore 版本	初始版	改进版
CVE-2018-0776	1.7.5	√	√
CVE-2018-0834	1.8.0	√	√
CVE-2018-0835	1.8.0	√	√
CVE-2018-0837	1.8.0	×	√
CVE-2018-0838	1.8.0	√	√
CVE-2018-0840	1.8.0	√	√
CVE-2018-0933	1.8.1	√	√
CVE-2018-0934	1.8.1	√	√
CVE-2018-0953	1.8.3	×	√
CVE-2018-8372	1.10.1	×	√
CVE-2018-8617	1.11.3	×	√
CVE-2019-0539	1.11.4	√	√
CVE-2019-0567	1.11.4	√	√
CVE-2019-0650	1.11.4	×	√

初始版 TC-JIT-San 成功检测出 9 个漏洞,其检测成功率为 64.3%,漏报(FN)为 35.7%。改进版 TC-JIT-San 成功检测出 14 个漏洞,其检测成功率为 100%,漏报(FN)为 0%。实验显示,TC-JIT-San 在已知漏洞测试集上具有有效的检测能力。改进版 TC-JIT-San 因考虑到由检查操作或快慢路径导致的语义操作相同但路径不同的情况,因此可以降低漏报。虽然改进版 TC-JIT-San 在测试集 T1 上的漏报率为 0%,但这并不意味着其没有漏报,本文在 4.2.5节进行进一步讨论。

4.2.2 对正常测试用例的检测(误报实验)

实验使用 T2 和 T3 评估 TC-JIT-San 初始版和改进版对正常测试用例的检测能力,从而评估该检测器的误报情况。这些测试用例在引擎发布前会被反复测试,因此本文默认其不会触发引擎缺陷。实验使用 TC-JIT-San 对最新版的 ChakraCore(v1.11.24)进行插桩,实验结果如表 2 所示。

在 6 组实验中,初始版 TC-JIT-San 的平均误报率(FP)为 0%,改进版 TC-JIT-San 对 2 个测试用例报出虚警,平均误报率(FP)为 0.11%。误报原因主要是由于缺少准确的数据流分析。本文也在 4.2.5 节进行进一步讨论。

表 2 TC-JIT-San 对正常测试用例的检测情况

组	测试集	测试用例数目	检测存在类型混淆数目	
			初始版	改进版
1	T2-1	300	0	0
2	T2-2	300	0	2
3	T2-3	300	0	0
4	T3-1	300	0	0
5	T3-2	300	0	0
6	T3-3	300	0	0

4.2.3 运行时开销

实验使用测试集 T4 和 T5 评估 TC-JIT-San 初始版和改进版的运行时开销。用于实验的服务器配置为:Intel Xeon CPU E7-4807(4 CPU,1.87GHz),16 GB 内存。实验测试最新版的 ChakraCore(v1.11.24)引擎,并为其编译了原始版、插桩 TC-JIT-San 初始版和插桩 TC-JIT-San 改进版 3 个可执行程序。对比 3 个版本引擎的运行时间,评估 TC-JIT-San 的运行时开销。实验结果如表 3 所示。

插桩了 TC-JIT-San 初始版的引擎的平均运行时

开销为未插桩版的 1.65 倍,而插桩了 TC-JIT-San 改进版的引擎的平均运行时开销为未插桩版的 1.84 倍。相关研究表明[2],实用性较高的动态检测器的运行时开销一般控制在 3 倍之内,TC-JIT-San 的运行时开销也满足这一限制。该实验展示了 TC-JIT-San 具有良好的实用价值。

4.2.4 对未知漏洞的检测

实验将 TC-JIT-San 改进版加入自行开发的 JavaScript 引擎的模糊测试系统中,对自动生成的测试用例进行动态检测。模糊测试所使用的的机器配置是 Intel Xeon Gold 6143(14 cores),500 GB 内存。

实验在 ChakraCore 1.10.0 版本上发现了 1 个 JIT 代码的类型混淆漏洞。测试用例如图 9 所示。通过人工分析该漏洞的致错原因,并追踪 ChakraCore 的漏洞修复记录,确认该漏洞为 CVE-2019-0810[15]。

表 3 TC-JIT-San 的运行时开销

组	测试集	Ch 版本	测试用例数目	运行时间/s	运行速度/个/s	运行时开销/倍
		原始版		110.8	4.5	–
1	T4 – 1	TC-JIT-San 初始版	500	215.3	2.3	1.94x
		TC-JIT-San 改进版		247.1	2.0	2.23x
		原始版		170.9	2.9	–
2	T4 – 2	TC-JIT-San 初始版	500	303.6	1.6	1.78x
		TC-JIT-San 改进版		329.5	1.5	1.93x
		原始版		179.7	2.8	–
3	T4 – 3	TC-JIT-San 初始版	500	321.9	1.6	1.79x
		TC-JIT-San 改进版		370.7	1.3	2.06x
		原始版		58.7	3.8	–
4	T5	TC-JIT-San 初始版	222	62.8	3.5	1.07x
		TC-JIT-San 改进版		64.7	3.4	1.12x

```
1  function opt(obj, victim) {
2    obj.b = 0x3210;
3    for (let i = -1; i < 0; i++) {
4      delete victim[i];
5    }
6    obj.a = 0x1234;
7  }
8  for (let i = 0; i < 0x1000; i++) {
9    let obj = {'-1': -1, a: 0x01234, b:0x4567,
                c:0x89ab, d:0xcdef};
10   opt(obj, {});
11 }
12 let victim = {'-1': -1, a: 0x01234, b:0x4567,
                c:0x89ab, d:0xcdef};
13 opt(victim, victim);
```

图 9 可触发 JIT 代码类型混淆漏洞的测试用例

该漏洞在 ChakraCore 1.11.8 版本上已被修复[16],但尚未公开任何漏洞相关的概念证明(proof of concept, PoC)。上述实验展示了,TC-JIT-San 也能够检测出未知的 JIT 代码类型混淆漏洞。

4.2.5 实验结论及分析

检测器的设计需要在漏报、误报以及运行时开销之间取得平衡。因此,本小节对 TC-JIT-San 的一些设计决策以及其对漏报、误报和运行时开销的影响进行讨论。

TC-JIT-San 的设计依赖于一些假设条件的成立。其中包括:慢路径的访存方式是安全的,以及引擎中跳伞机制的处理是正确的。一旦上述条件不成立,则可能导致漏报。

TC-JIT-San 的漏报和误报主要来源于缺少准确的数据流信息。一种可以导致误报的情况:预设路径和“致错路径”完全相同,但“致错路径”上发生类型迁移的数据对象和后续访问的数据对象并非同一对象。当前 TC-JIT-San 的检测机制仅依赖于控制流信息,通过增加数据流分析,可以减少上述漏报和误报。但是,这也会引入较大的运行时开销。因此,最终 TC-JIT-San 还是仅采用控制流信息来指导类型混淆漏洞的检测。

5 结 论

本文针对 JavaScript 引擎 JIT 代码的类型混淆缺陷,提出了一种动态检测方法。该方法基于类型混淆漏洞的触发条件,通过判断 JIT 代码执行是否违背了正常执行逻辑,从而发现引擎中的类型混淆缺陷。本文从漏报、误报和运行时开销 3 个角度对该检测方法进行评估。实验表明,该检测方法对于检测 JavaScript 引擎 JIT 代码的类型混淆缺陷,具有有效性和实用性。

参考文献

[1] CWECONTENT TEAM. CWE-843: access of resource using incompatibletype ('Type Confusion') [EB/OL]. (2011-07-20) [2022-03-23]. https://cwe.mitre.org/data/definitions/843.html.

[2] SONG D, LETTNER J, RAJASEKARAN P, et al. SoK: sanitizing for security[C]//2019 IEEE Symposium on Security and Privacy. San Francisco: IEEE, 2019: 1275-1295.

[3] 任泽众,郑晗,张嘉元,等. 模糊测试技术综述[J]. 计算机研究与发展,2021,58(5): 944-963.

[4] LEE B, SONG C, KIM T, et al. Type casting verification: stopping an emerging attack vector[C]//The 24th USENIX Security Symposium. Washington: USENIX Association, 2015: 81-96.

[5] ISTVAN H, YUSEOK J, HUI P, et al. TypeSan: practical type confusion detection[C]//Proceedings of 2016 ACM SIGSAC Conference on Computer and Communications Security. New York: Association for Computing Machinery, 2016: 517-528.

[6] JEON Y, BISWAS P, CARR S, et al. Hextype: efficient detection of type confusion errors for C++[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas: Association for Computing Machinery, 2017: 2373-2387.

[7] CHAKRACORE. ChakraCore is an open source Javascript engine with a C API[EB/OL]. (2020-12-09)[2022-03-23]. https://github.com/chakra-core/ChakraCore.

[8] TUNZ, Case study of JavaScript engine vulnerabilities[EB/OL]. (2019-09-04)[2022-03-23]. https://github.com/tunz/js-vuln-db.

[9] IEEE-754. IEEE 754-2019 IEEE standard for floating-point arithmetic[EB/OL]. (2019-07-22)[2022-03-23]. https://standards.ieee.org/standard/754-2019.html.

[10] FILIP PIZLO. Speculation in JavaScript core[EB/OL]. (2020-07-29)[2022-03-23] https://webkit.org/blog/10308/speculation-in-javascriptcore/:webkit.

[11] 黎遇军,张昱. JavaScript 引擎中 Deoptimization 模式分析与改善[J]. 电子技术, 2017, 46(7):17-23.

[12] CAPSTONE. The ultimate disassembler [EB/OL]. (2020-05-08)[2022-03-23]. https://www.capstone-engine.org/:capstone.

[13] ZENHUMANY, CYRILIU. Using the JIT vulnerability to Pwning Microsoft edge[EB/OL]. (2019-03-29)[2022-03-23]. https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Li-Using-the-JIT-Vulnerability-to-Pwning-Microsoft-Edge.pdf.

[14] NICK R. Ranking the top 100 websites in the world[EB/OL]. (2019-08-07)[2022-03-23]. https://www.visualcapitalist.com/ranking-the-top-100-websites-in-the-world/.

[15] CVE. CVE-2019-0810[EB/OL]. (2018-11-26)[2022-03-23]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0810.

[16] CHAKRACORE. April servicing update for ChakraCore [EB/OL]. (2019-04-10)[2022-03-23]. https://github.com/chakra-core/ChakraCore/pull/6087.

Type confusion vulnerability sanitizer in JavaScript engine JIT code

SUN Lili, ZHANG Peihua, WU Chenggang, WANG Zhe

(State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100190)

Abstract

Type confusion is a type of vulnerability that has recently emerged in JavaScript engines. However, due to JIT (just-in-time) code limitations, prior type confusion sanitizers are unable to detect type confusion bugs in JavaScript engine JIT code. For the first time, this paper proposes a detection method for these bugs, as well as a sanitizer, TC-JIT-San. The method takes advantage of the correlation between the JIT code's execution flow and data types, converting the issue of recognizing data types in JIT code into observing changes in the execution flow. TC-JIT-San detects type confusion bugs by observing whether changes in the execution flow comply with normal execution logic. Experiments show that TC-JIT-San has a low overhead, as well as a low number of false negatives and false positives. Its runtime overhead is 1.84 times that of normal execution, and its average false negative and false positive rates are 0% and 0.11%, respectively.

Key words: vulnerability finding, sanitizer, software flaw detection