基于数据结构特征发现的脚本引擎内置对象别名关系识别

张羿伟, 游　伟, 梁　彬, 万欣宇, 郭苏越

中国人民大学信息学院 北京 中国 100872

摘要　越来越多的软件提供内置脚本引擎, 支持通过脚本语言可编程式地调用各项程序功能。内置脚本引擎除了支持标准的脚本语言规范, 还提供了一系列扩展的应用程序编程接口(Application Programming Interface, API)和内置对象。脚本引擎在丰富软件功能的同时, 也引入了额外的攻击面。近年来曝出的内置脚本引擎安全漏洞多数与扩展 API 和内置对象相关。针对脚本引擎中的安全漏洞, 目前的检测技术仅能检测出脚本引擎浅层解析代码中的漏洞, 难以有效检测出涉及内置对象别名关系的深层次漏洞, 例如释放后使用漏洞(use-after-free, UAF)。检测对象别名关系导致的 UAF 漏洞, 需要解决两个关键的技术挑战。其一, 如何高效地识别内置对象别名关系。其二, 如何利用识别出的对象别名关系检测脚本引擎的 UAF 漏洞。为此, 本文设计了一种基于数据结构特征发现的脚本引擎内置对象别名关系识别方法, 并提出了一种利用别名关系构造式触发 UAF 漏洞的检测方案。我们利用内置对象数据结构特征, 提高了别名关系识别效率。同时, 引入了内置对象内存共享区域以辅助触发 UAF 漏洞。为了验证方案的有效性, 我们在 Adobe Reader 的内置 JavaScript 脚本引擎中进行了实验。我们提出的方案成功地识别出了 284 对内置对象的别名关系, 并检测出了 4 个未知的 UAF 漏洞, 获得了厂商的确认和修复。实验结果表明, 本文方法能有效识别内置对象别名关系并能成功应用于脚本引擎 UAF 漏洞的检测。

关键词　脚本引擎; 别名关系; 数据结构特征; 漏洞检测

中图法分类号　TP311　DOI 号 10.19363/J.cnki.cn10-1380/tn.2022.07.06

Identifying Alias Relationship between Built-in Objects of Script Engine Based on the Discovery of Data Structure Signatures

ZHANG Yiwei, YOU Wei, LIANG Bing, WAN Xinyu, GUO Suyue

School of Information, Remmin Vniversity of China, Beijing 100872, China

Abstract　More and more software products provide embedded script engine to enable the users to programmatically invoke various program functions. The embedded script engine not only supports the standard script language specification, but also extends a set of application programming interface (API) and built-in objects. While enriching the software functionality, the embedded script engine also introduces additional attack surfaces. Recent years have witnessed large number of security vulnerabilities found in the embedded script engine, most of which are related to extended API and build-in objects. For the vulnerability detection in the script engine, the existing works can only detect vulnerabilities in the shallow part of the script engine, and fails to effectively detect deep vulnerabilities involving alias relationships between built-in objects, such as use-after-free (UAF) vulnerabilities. Two technical challenges need to be addressed for detecting UAF vulnerabilities caused by the alias relationship between built-in objects. The first one is how to efficiently identify the alias relationship between built-in objects. The second one is how to use the identified object alias relationship to detect UAF vulnerabilities in the script engine. To this end, we propose a method to identify the alias relationship of built-in objects in script engine based on the discovery of data structure signatures, and use the alias relationship to constructively trigger UAF vulnerabilities. Using data structure signatures of built-in objects, we greatly improve the efficiency of alias relationship identification. At the same time, we introduce shared memory areas within built-in objects, helping trigger UAF vulnerabilities. In order to assess the effectiveness of the proposed method, we conducted experiments in Adobe Reader's embedded JavaScript engine. Our method successfully identified 284 pairs of alias relationships between built-in objects, which in turn exposed four unknown UAF vulnerabilities. These vulnerabilities are confirmed by the vendor and got fixed. The experimental results show that our method can effectively identify alias relationship between built-in objects and helps the de-

通讯作者: 游伟, 博士, 副教授, Email: youwei@ruc.edu.cn

本课题得到国家自然科学基金(No. 62002361, No.U1836209)资助。

收稿日期: 2021-05-13; 修改日期: 2021-08-10; 定稿日期: 2022-05-11

tection of UAF vulnerabilities in the embedded script engine.

Key words script engine; alias relationship; data structure signatures; vulnerability detection

1 引言

脚本语言具有简单易用的特性, 越来越多的软件支持通过脚本语言可编程式地调用各项程序功能。这些软件通过内置的脚本引擎对脚本代码进行解析并执行。内置脚本引擎除了支持标准的脚本语言规范, 还提供了一系列扩展的应用程序编程接口(Application programming interface, API)和内置对象。用户可以通过调用扩展 API 来执行由本地代码(Native code)实现的程序功能。

脚本引擎在丰富软件功能的同时, 也引入了额外的攻击面。攻击者可以利用脚本引擎漏洞, 编写特定攻击代码, 达到控制用户设备、窃取用户隐私等目的。根据漏洞库中的信息, 近年来频繁出现脚本引擎相关的程序安全漏洞, 已造成大量经济损失与安全危害[1]。通过对已公开的漏洞利用样本进行分析[2], 我们发现绝大多数的脚本引擎漏洞是由于扩展 API 和内置对象使用不当产生的。

针对脚本引擎中的程序漏洞, 目前主流的检测技术为模糊测试(Fuzzing)[3], 使用特定的生成/变异策略产生大量测试样本, 并监测测试样本的执行是否触发程序异常。然而, 现有的模糊测试研究工作主要关注如何生成符合脚本语言语法规范的测试样本。虽然取得了一定的成效, 但是这些研究工作仅能检测出脚本引擎浅层解析代码中的漏洞, 难以有效检测出脚本引擎深层代码中的漏洞。

脚本引擎中的深层次漏洞通常与扩展 API 及内置对象相关。以释放后使用(Use-After-Free, UAF)漏洞为例, 其本质为通过悬挂指针非法访问了被释放的内存区域。在脚本引擎中, 通过调用特定的 API 序列, 可以创建内置对象内部的共享内存区域, 进而产生不同对象在共享内存区域上的别名关系。释放其中一个对象内部的共享内存区域, 将会在与其存在别名关系的另一对象内部产生悬挂指针, 访问该悬挂指针会触发释放后使用漏洞。现有的模糊测试技术没有关注内置对象内部结构, 因而在发现释放后使用漏洞方面效果较差。

为了检测由对象别名关系导致的释放后使用漏洞, 需要解决两个关键的技术挑战。其一, 如何高效地识别内置对象别名关系。脚本引擎中的内置对象由脚本引擎负责维护, 用户可以通过使用 API 的方式来创建、更改和删除内置对象。生成合理的 API 序列对不同对象进行有针对性的操作是首要解决的问题。除此之外, 由于内置对象别名关系与对象结构紧密相关, 需要实现一种机制快速检测 API 序列的执行是否建立了对象间的共享内存区域。其二, 如何利用识别出的对象别名关系检测脚本引擎的释放后使用漏洞。为了触发脚本引擎释放后使用漏洞, 需要构造性地利用对象别名关系以产生悬挂指针。别名关系代表不同内置对象内部存在共享内存区域, 因此需要考虑如何生成 API 序列释放对象内部的共享内存区域, 以产生悬挂指针。进一步, 利用 API 序列针对性地访问悬挂指针, 触发释放后使用漏洞。

为了解决上述挑战, 我们提出了一种基于数据结构特征发现的脚本引擎内置对象别名关系识别方法, 并利用内置对象别名关系辅助检测脚本引擎中的释放后使用漏洞。具体而言, 通过生成特殊的脚本引擎 API 调用序列, 建立内置对象别名关系, 并使用对象数据结构特征, 将内置对象别名关系识别转化为图连通性识别, 加快了内置对象别名关系识别速度。在此基础上, 我们提出了一种利用别名关系构造式触发释放后使用漏洞的检测方案。对于具有别名关系的两个对象, 通过调用特殊的脚本引擎 API 序列, 释放其中一个对象内部的共享内存区域, 从而在另一个对象内部产生悬挂指针, 随后调用特殊的脚本引擎 API 序列访问另悬挂指针, 以触发释放后使用漏洞。

本文的主要贡献有:

(1) 提出了一种基于数据结构特征的脚本引擎内置对象别名关系识别方法。通过利用内置对象数据结构特征, 自动化提取脚本引擎 API 参数信息并生成合理 API 序列, 提高了对象别名关系的产生概率。同时, 利用对象数据结构特征, 将内置对象别名关系识别转化为图连通性识别。在保证识别准确率的前提下, 显著减小了性能开销, 加快了内置对象别名关系识别速度。

(2) 创新性地将内置对象别名关系应用于检测脚本引擎释放后使用漏洞。通过使用内置对象别名关系, 搭配特殊 API 序列, 增加了释放操作产生的悬挂指针数量。相比于传统释放后使用漏洞, 由对象别名关系导致的释放后使用漏洞更容易利用且更难修补。

(3) 设计并实现了别名关系识别方法的系统原型, 成功提取出了 284 组内置对象别名关系, 并设计

评估实验证明了方法的高效性与准确性。进一步, 针对真实软件中的脚本引擎进行了漏洞检测, 成功发现了 4 个未知释放后使用漏洞, 帮助提高了软件安全性。

2 相关工作

针对脚本引擎中的安全漏洞, 目前主流的检测技术为模糊测试(Fuzzing)技术。模糊测试技术根据特定的生成策略产生大量测试样本, 并监测测试样本的执行能否触发程序异常。按照技术细节不同, 具体又可分为生成型模糊测试以及变异型模糊测试[4]。

生成型模糊测试技术基于给定的规范(如语言语法, 样本结构等)生成测试样本, 部分生成型模糊测试框架也会借助语料库信息[5], 常见的测试框架有 SPIKE[6]和 jsfunfuzz[7]等。为了生成语法正确的测试样本, 通常使用上下文无关语法作为规范[8-9]。除此之外, HyungSeok 等人[10]和 Soyeon 等人[11]提出利用现有正确测试样本中的语法信息来完善样本生成规范。进一步, Suyoung 等人[12]提出利用神经网络来训练模型以产生样本生成规范, 旨在通过完善规范来提升测试样本质量。然而, 不同脚本语句之间存在依赖关系, 生成型模糊测试技术无法捕获此类依赖关系, 因而无法避免由依赖关系导致的运行时异常。

变异型模糊测试技术基于给定的种子输入, 采用不同的变异策略(如比特翻转, 随机替换等)生成新的测试样本, 常见的测试框架有 AFL[13]和 SymFuzz[14]等。变异型模糊测试技术通常以代码覆盖率为指标评估测试样本的质量, 认为代码覆盖率越高测试样本质量越好。为了提高测试样本的质量, 常使用污染传播[15-16], 符号执行[17]和约束求解[18]等技术。除此之外, Sanjay 等人[19]通过设置程序路径权重, 有针对性地进行变异以生成测试样本, 以及 Caroline 等人[20]提出在变异过程中加入已知漏洞样本(Proof of Concept)信息, 以生成高质量测试样本。然而, 变异型模糊测试容易破坏种子输入的完整性, 导致无法通过完整性校验。

在测试脚本引擎漏洞方面, Sung 等人[21]提出了一种使用内置 API 和内置对象来测试 JavaScript 脚本引擎的方法, 以减少运行时错误的产生。该方法虽然利用脚本引擎 API 和内置对象信息减少了运行时错误的数量, 但测试样本的生成不够自动化, 且需要一些先验知识做引导。除此之外, Nicolas 等人[22]提出利用程序运行时的内存访问信息来改进测试样本, 以测试更复杂的程序模块, 但对于大型脚本引擎效果不佳。

别名分析的主要目的在于确定程序中指针的具体指向, 进而判定指针操作是否存在潜在风险[23]。现有别名分析技术主要分为静态别名分析以及动态别名分析。静态别名分析通过分析程序源代码来识别别名关系, 常结合控制流图或调用图进行分析[24], 即过程间分析[25-26]。通常静态别名分析所需时间与控制流图的复杂程度成正比, 为了精简控制流图, Tobias 等人[27]提出将分支条件与变量状态关联, 并依据变量状态来删除冗余分支。 动态别名分析则通过监控内存中变量的变化来识别别名关系, 常在中间代码层面进行程序插装以判断不同变量之间是否共享同一内存区域[28-29]。Manu 等人[30]提出实施简单的可达性分析, 缩小监控范围提高分析性能。然而, 由于脚本引擎结构复杂, 使用现有技术获得精确分析结果存在性能开销大的问题, 且内置对象结构特殊, 无法使用通用的处理策略。

3 脚本引擎与释放后使用漏洞

脚本引擎作为脚本语言运行的载体, 负责解析执行脚本语言, 以及提供相应的运行环境。脚本引擎提供了一系列定制的应用程序接口(Application programming interface, API)来帮助用户高效使用内部功能, 我们称之为特殊 API, 如 Adobe Reader 内嵌脚本引擎中的 Doc.createIcon 负责在文档中创建按钮元件。与特殊 API 类似, 脚本引擎中的内置对象也具有特殊的结构和语义, 如 Adobe Reader 内嵌引擎中的 Doc 对象代表当前文档。特殊 API 以及内置对象是脚本引擎的重要组成部分, 不同脚本引擎中特殊 API 以及内置对象的数量和功能均有较大差异, 合理使用特殊 API 与内置对象能触及更多、更深层次的引擎代码。

在使用脚本引擎 API 时, 需要遵照特定的使用规范, 即提供正确数量和类型的参数。当脚本引擎 API 参数的数量或类型错误时, 脚本引擎会抛出运行时异常, 并终止脚本代码的执行。脚本引擎中 API 参数的类型包括布尔类型, 整数类型, 浮点类型, 字符串类型以及对象类型五类, 不同类型之间可以动态转化。

进一步, 经过实验发现, 脚本引擎 API 在运行时需要特定结构的内置对象, 因此需要分析内置对象的细粒度类型信息。通常, 对于脚本引擎 API, 软件开发者会提供一份官方的使用规范, 说明不同脚本引擎 API 需要的参数个数以及类型。然而, 以 Adobe Reader 内嵌脚本引擎为例, 官方提供的使用规范并不详细, 未说明脚本引擎 API 对象参数的细粒度类

型信息。我们统计了 Adobe Reader 内嵌脚本引擎中的 386 个 API, 其中使用对象类型参数的 API 有 255 个, 占比约 66%。为了减少运行时错误, 我们需要提取 API 参数的细粒度类型, 进而正确且针对性地使用脚本引擎 API 和内置对象。

内置对象的生命周期由脚本引擎负责维护, 在生命周期结束时被脚本引擎回收。用户可通过调用脚本引擎API来创建, 更改和删除内置对象, 即能够通过调用脚本引擎 API 影响内置对象生命周期。按照创建方式的不同, 具体可以分为静态内置对象以及动态内置对象。静态内置对象由脚本引擎预创建, 可直接使用对象名称访问, 而动态内置对象需要调用脚本引擎 API 创建后才能访问。

对于静态内置对象以及动态内置对象, 脚本引擎采用一个活跃对象集合维护其生命周期。活跃对象集合存储脚本引擎中存活的内置对象, 其状态随着内置对象的创建和删除实时变化。活跃对象集合由脚本引擎负责维护, 用户无法直接更改其状态, 但可以通过影响内置对象生命周期, 间接影响活跃对象集合的状态。活跃对象集合常用于查询内置对象生命周期信息, 在脚本引擎中具有重要意义。

脚本引擎在通过丰富的 API 以及内置对象提供便利的同时, 也引入了更多安全漏洞。攻击者可利用脚本引擎 API 以及内置对象构造特殊的攻击代码, 触发多种类型的安全漏洞, 对用户造成危害。其中, 释放后使用漏洞与非法内存操作紧密相关, 可利用性强, 常成为攻击者的重点关注目标。释放后使用漏洞的本质为通过悬挂指针访问了被释放的内存区域, 而悬挂指针是指向被释放内存区域的指针, 是由于对象释放后缺乏合理的访问检查导致。我们以图 1 为例具体说明释放后使用漏洞的触发模式和修补方式。

(a) 修补前:		(b) 修补后:	
① var *p* = malloc()	//创建	④ var *p* = malloc()	//创建
② free(*p*)	//释放	⑤ ***p*_flag = False**	
③ use(*p*)	//使用	⑥ free(*p*)	//释放
		⑦ ***p*_flag = True**	**//设置标志**
		⑧ **if *p*_flag = False**	**//检查标志**
		⑨ use(*p*)	//使用

图 1 释放后使用漏洞触发模式及修补方式

Figure 1 Trigger mode and patch of UAF vulnerability

图 1 展示了 1 个典型释放后使用漏洞的触发模式。首先执行创建操作(malloc)创建对象 *p*(对应标号为①的语句)。接着调用合适的 API 序列执行释放操作(free), 释放创建出的对象 *p* 并产生了悬挂指针(对应标号为②的语句)。最后, 使用(use)被释放的对象 *p* 并访问其内部的悬挂指针, 触发了释放后使用类型的安全漏洞(对应标号为③的语句)。现有工作针对释放后使用漏洞的修补方式对应语句⑤⑦⑧, 即增加对象释放标志并根据对象状态实时维护释放标志(对应标号为⑦的语句), 同时在每次访问对象前检查释放标志的状态(对应标号为⑧的语句), 当对象释放标志被设置为 True 时拒绝访问对象。

修复释放后使用漏洞的核心在于消除悬挂指针, 现有设置对象释放标志的方式简单有效, 且适用于所有类型的对象, 通用性强。然而, 由于脚本引擎内部存在复杂对象关系, 比如内置对象别名关系, 导致不同内置对象内部存在共享内存区域, 进而在释放时出现多个悬挂指针。相比于原有的释放后使用漏洞, 使用对象别名关系可以在不同对象中引入悬挂指针, 增加了漏洞的不确定性, 利用释放标志进行修补也更加困难。我们将由对象别名关系导致的释放后使用漏洞触发模式总结为图 2 所示。

如图 2 所示, 首先执行创建操作产生了对象 *p* 和对象 *q*(对应标号为①②的语句)。接着, 通过执行特定脚本引擎 API 序列, 建立了对象 *p* 和对象 *q* 之间的别名关系(对应标号为③的语句), 具体对应对象 *p*, *q* 内部的共享内存区域。之后, 执行释放操作释放对象 *p*(对应标号为④的语句), 产生了悬挂指针。不同于传统释放后使用漏洞触发模式, 在最后访问悬挂指针时, 并未使用上一步中释放的对象 *p*, 而是使用了与对象 *p* 存在别名关系的对象 *q*(对应标号为⑤的语句)。

```
① var p = malloc()  //创建
② var q = malloc()  //创建
③ relation(p, q)    //关联
④ free(p)           //释放
⑤ use(q)            //使用
```

图 2 别名关系释放后使用漏洞触发模式

Figure 2 Trigger mode of UAF vulnerability with alias relationship

相较于传统的释放后使用漏洞, 通过建立内置对象别名关系, 引入了更多的悬挂指针, 增加了释放后使用漏洞的可利用性。传统的释放后使用漏洞触发模式与对象别名关系导致的释放后使用漏洞触发模式有如下异同点: 二者均通过访问悬挂指针触发安全漏洞, 但在传统释放后使用漏洞触发模式中, 释放以及使用的为同一个对象, 仅产生了一个悬挂指针。在别名关系导致的释放后使用漏洞中, 释放以及使用操作涉及不同对象, 产生了多个悬挂指针。相

较于传统的释放后使用漏洞, 别名关系触发的释放后使用漏洞更为复杂, 涉及不同内置对象的交互, 并且现有的防御措施无法有效抵御别名关系导致的释放后使用漏洞。

4 别名关系识别

本研究提出了一种基于数据结构特征脚本引擎内置对象别名关系识别方法, 能高效准确地建立并识别脚本引擎内置对象别名关系, 如图 3 所示, 整个内置对象别名关系识别方法共分为三个阶段。

第一阶段对应文章 3.2 节, 负责提取脚本引擎内置对象的数据结构特征。该阶段主要采用动态插装的方式, 使用脚本引擎 API 创建并访问内置对象并监控程序执行, 获取程序运行时内置对象的内存地址。进一步, 通过识别与对象起始地址相关的内存单元类型, 提取脚本引擎内置对象的数据结构特征。

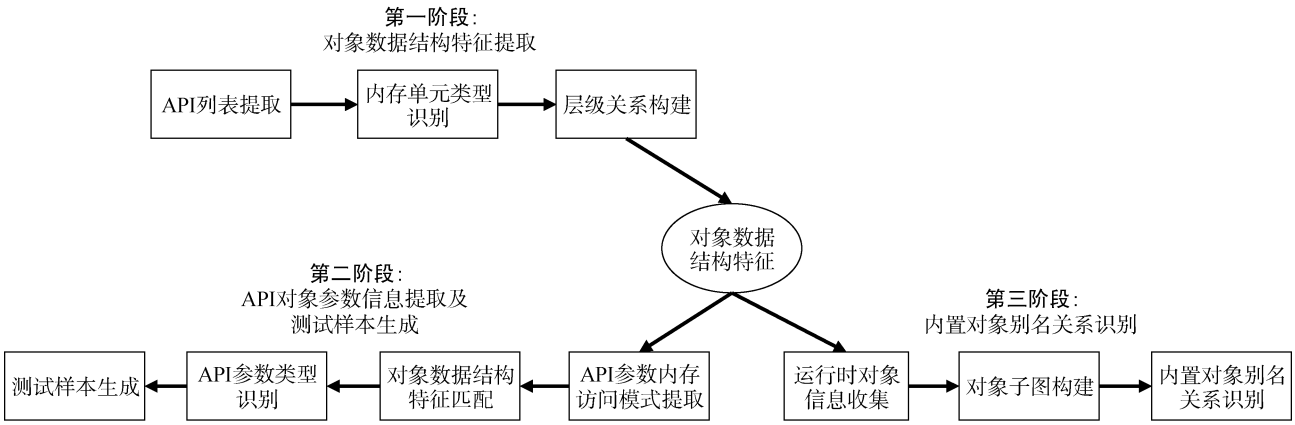

图 3 系统流程图

Figure 3 System flow chart

第二阶段对应文章 3.3 节, 负责生成参数类型正确的脚本引擎 API 调用实例。此阶段我们采用静态分析技术来提取脚本引擎 API 对象参数的内存访问模式, 并与第一阶段中提取的对象数据结构特征匹配, 获得 API 对象参数候选集。进一步, 使用获得的 API 对象参数列表, 生成参数类型正确的脚本引擎 API 调用语句, 并组合不同 API 调用语句产生测试样本。

第三阶段对应文章 3.4 节, 负责监测共享内存区域来识别内置对象别名关系。该阶段主要使用动态测试的技术, 获取程序运行时对象的内存地址, 提取对应的内置对象数据结构特征, 并转化为数据结构特征图存储。通过在 API 执行前后插入检查点的方式, 分析数据结构特征图的变化, 并根据数据结构特征图的变化识别内置对象别名关系。

在接下来的部分, 我们首先介绍内置对象数据结构特征的定义, 之后依次介绍流程每个阶段的具体实现。

4.1 内置对象数据结构特征

内置对象数据结构特征 T 定义为由不同类型内存单元组成的, 具有明显分层与指向关系的一种内存特征。我们以字长为单位划分内置对象所在的内存区域, 并将划分后的字长大小的内存区域称为内存单元。字长的大小与操作系统相关, 在 64 位操作系统中字长大小为 8 字节, 32 位操作系统中字长大小为 4 字节。对于内置对象数据结构特征 T, 我们采用有向图的方式表示如下:

$$T = (V, E)$$

其中, V 代表有向图中顶点的集合, E 代表有向图中边的集合, 具体如下:

$$V = (v_1, v_2, v_3, \ldots)$$

$$E = (e_1, e_2, e_3, \ldots)$$

顶点集合 V 中每个元素 v_i 均对应一个内存单元, v_i 的具体定义如下:

$$vi = \text{<Value, Type, Ancient>}$$

v_i 定义为由 Value, Type 和 Ancient 组成的三元组, 其中 Value 代表运行时内存单元中存储的数值, 随着脚本引擎 API 的执行发生变化。Type 代表内存单元数值的类型, 具体包括可变数值类型(Number), 固定数值类型(Padding)以及指针类型(Pointer)3 种, 在具体实现时用不同数值代表 3 种类型(如指针类型对应数值 0x1)。固定数值类型代表内存单元中数据的数值固定, 即相同类型的内置对象拥有相同的数值。同时, 固定数值类型对应的数值与运行状态无关, 即多次创建出相同类型的内置对象, 其固定数值类型对应的数值均相同。可变数值类型的定义与固定数值类型相反。对于相同类型对象, 可变数值类型内存

单元中数据的数值不固定, 且与运行状态紧密相关, 在多次运行中会得到不同的数值。指针类型的内存单元代表该内存单元中的数据指向其他合法内存单元, 能够通过指针解引用操作进行层级展开。Ancient 为内存单元所属标识, 代表该内存单元从属于特定内置对象。我们为同一内置对象下属所有内存单元设置统一的 Ancient, 并使用唯一 Ancient 值以区分不同内置对象。通常, Ancient 值被设置为内置对象运行时内存起始地址, 方便查询内置对象信息。

边集 E 中每个元素 e_i 代表有向图中顶点的指向关系, 其具体定义如下:

$$e_i = <v_s, v_e, \text{Offset}>$$

其中, v_s 和 v_e 分别代表该有向边的起始顶点和终止顶点, 对应顶点集合 V 中的元素。Offset 为顶点 v_s 代表内存单元到顶点 v_e 代表内存单元的内存偏移。在 e_i 中, 顶点 v_s 对应指针类型的内存单元, 我们根据其中包含的指向信息划分对象数据结构特征内部的层级。以 Stream 对象为例, Stream 对象起始地址对应的内存区域称为第一层级, 第一层级中指针类型指向的内存区域为第二层级, 以此类推。

我们将存储对象数据结构特征的有向图称为数据结构特征图。数据结构特征图中的根节点代表具体内置对象, 每个子节点对应数据结构特征中的每个内存单元。以 Stream 类型对象为例, 最终转化成的数据结构特征图如图 4 所示。以图 4 右下角阴影标注的顶点为例, 三元组中 Value 的值为 0x1340, 代表运行时内存单元中的数值为 0x1340。Type 值为 0x1 代表该顶点的类型为指针类型。Ancient 值为 0x1220 对应对象根节点的 Value 值(即图 4 中 Stream 标注的顶点), 标明该顶点属于 Stream 对象。

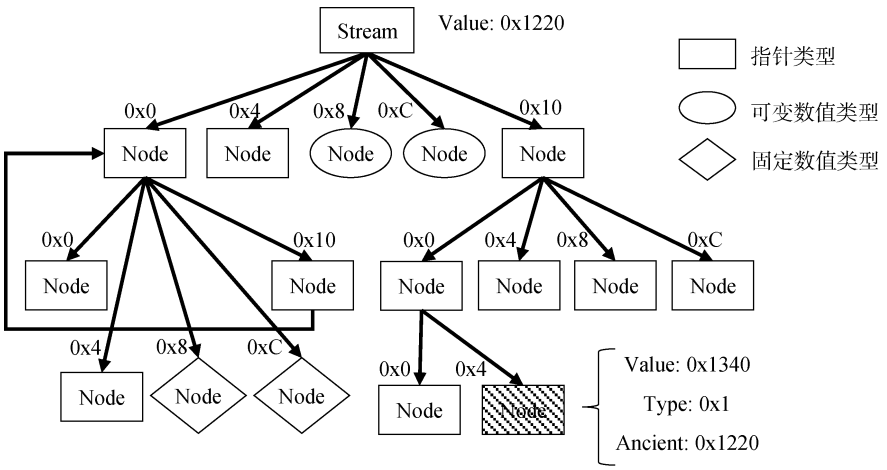

图 4 Stream 类型对象数据结构特征图

Figure 4 Data structure signature graph of type Stream object

内置对象数据结构特征中的层级个数以及内存单元个数可自主配置。通常而言, 使用的层级个数和内存单元个数越多, 特征的分类效果越好, 而性能开销则越大。为了确定合适的配置, 我们进行了相关探索, 结果如表 1 所示。当层级个数为 1, 每个层级中内存单元的个数为 4 时, 所有内置对象被归为一个类, 没有任何区分度。随着层级个数与内存单元个数的增加, 内置对象种类的数量也随之上升, 上升幅度呈递减趋势。最终, 出于平衡性能开销与分类效果的考虑, 确定层级个数为 3, 层级中内存单元个数为 8。第 6 章的实验评估结果也表明, 这样的配置可以保证较高的测试用例生成效率。

表 1 内置对象种类统计表

Table 1 Statistics of built-in object type

层级个数	内存单元个数	内置对象类别数	运行耗时(s)
1	4	1	0.2
1	8	31	0.2
2	8	69	0.6
3	8	123	1.1
3	16	135	2.9

4.2 提取内置对象数据结构特征

在该节中我们具体介绍如何提取内置对象数据结构特征。该阶段主要采用动态插装的方式, 获取程序运行时内置对象的内存地址, 具体又分为获取对象内存地址以及内存单元类型识别两部分。进一步, 通过识别内置对象内存地址相关的内存单元类型, 提取出内置对象的数据结构特征。

4.2.1 提取内置对象列表

首先需要提取脚本引擎中所有静态、动态内置对象。为了提取脚本引擎所有的静态内置对象, 我们

采用深度优先遍历的方式, 从文档根节点出发依次访问文档中所有节点。具体实现时我们使用 for-in 的方式进行遍历, 对于访问到的每一个节点, 调用内置操作符 typeof 判断其节点类型, 并记录对象类型节点的信息, 即静态内置对象信息。同时, 为了避免遍历过程陷入死循环, 我们标记了所有已访问的节点, 并以集合的方式存储。当某次访问的节点位于标记集合中时, 不再从该节点出发进行更深层次的遍历。

脚本引擎中的动态内置对象需要调用脚本引擎 API 创建。由于官方文档的说明信息不全, 如针对 Collab 这一类型的内置对象, 官方文档中仅包含三个子方法说明, 而实际存在 23 个子方法。因此, 需要提取不在官方文档中的脚本引擎 API, 以获得完整的脚本引擎 API 列表。我们同样采用从文档根节点出发深度优先遍历的方式, 记录类型为函数的节点信息, 将其整理为最终的脚本引擎 API 列表, 使用提取出的 API 列表可以获得所有动态内置对象。接下来, 针对提取出的所有内置对象, 提取对应的数据结构特征。

4.2.2 获取内置对象内存地址

为了获得内置对象内存地址, 需要使用脚本引擎 API 创建和访问内置对象。通过在脚本引擎 API 执行前后插入检查点, 结合动态插装技术来获得内置对象内存地址。我们以图 5 中伪代码为例, 说明如何获得内置对象内存地址。

为了获取对象内存地址, 我们生成脚本引擎 API 调用语句创建和访问内置对象, 对应图 5 中②④, 同时在每条脚本引擎 API 调用语句前后设置检查点, 对应图 5 中①③⑤行。使用微软提供的动态调试工具 Windbg 可以在检查点判断上一条 API 调用语句返回值的类型, 根据返回值的类型可以获取部分由 API 创建的动态内置对象内存地址。除此之外, 还可以依据脚本引擎活跃对象集合状态的变化获取内置对象内存地址。当创建新内置对象时, 活跃对象集合将相应地增加新元素, 通过分析活跃对象集合的变化可以获取内置对象内存地址。

```
① CheckPoint                                  //检查点
② console.println(app.fs);                    //使用静态对象
③ CheckPoint                                  //检查点
④ this.Collab.newWrStreamToCosObj(); //创建动态对象
⑤ CheckPoint                                  //检查点
```

图 5 脚本引擎 API 伪代码

Figure 5 Pseudocode of script engine API

4.2.3 识别内存单元类型

在获取了内置对象内存地址后, 需要识别内存单元类型, 具体为从对象内存地址出发, 识别其对应内存区域中内存单元数据的类型。接下来, 我们将结合图 6 阐述识别内存单元数据类型的具体流程。

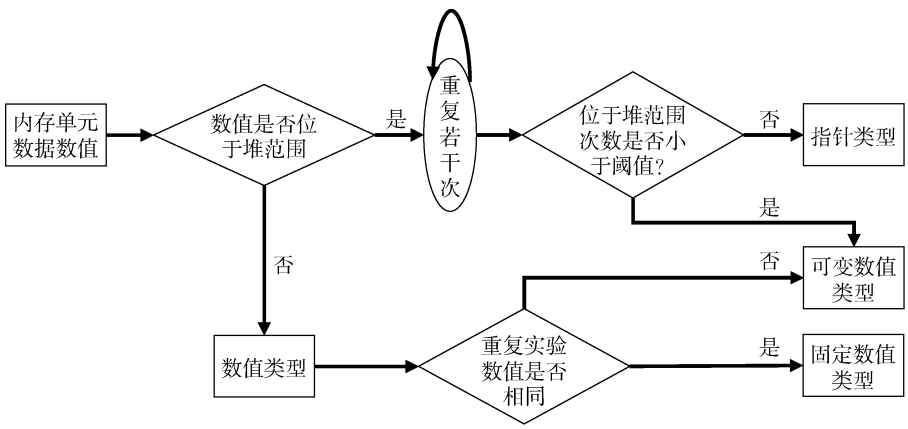

图 6 内存单元类型识别流程图

Figure 6 Flow chart of identifying memory unit type

为了提取对象数据结构特征, 需要将内存单元识别为指针类型, 可变数值类型或固定数值类型。进一步, 依据指针类型内存单元的指向关系, 识别出层级结构特征, 并组合各个层级形成对象数据结构特征。我们使用进程的内存布局信息作为先验知识来初步区分指针类型或数值类型。在 Windows 系统中程序的内存布局可大致分为代码段, 数据段, 栈以及堆。对于指针类型的内存单元, 其存储数值对应的内存地址位于堆中。通过获取进程运行时内存布局信息, 得到堆起始和终止内存地址, 并依据内存单元中数据数值是否位于堆范围来初步区分指针类型与数值类型(超出堆范围的数据可判定为数值类

型)。使用进程的内存布局信息能识别大约 90%的内存单元类型, 但仍有 10%的数据因数值特殊而无法区分, 如数值类型中存在位于堆的范围的特殊值。此类数据对应的内存单元(后续称为待定数值单元)将进一步进行内存单元的类型识别。

我们发现开启地址随机化之后, 在多次测试中, 待定数值单元中的数值每次均位于堆范围内的概率较小。因此, 可以进行多次独立测试, 并统计待定数值单元中数值位于堆范围的次数, 来判断该待定数值单元的类型。同时, 由于固定数值类型在多次测试中将具有相同的数值, 也可用该方法来区分可变数值类型与固定数值类型。从内置对象内存起始地址出发进行层级展开时, 我们通过配置参数的方式设置层级的个数以及层级的大小, 并通过不断实验调整配置参数, 以达到最好的区分效果。最终, 结合各层级中的内存单元类型可以提取出准确的内置对象数据结构特征。

在提取出内置对象数据结构特征后, 需要将数据结构特征转换为数据结构特征图进行存储。在转换的过程中, 我们重点关注指针类型对应的内存单元, 并将其指向关系对应的内存偏移映射为数据结构特征图中的边。

4.3 生成参数类型正确的 API 序列

官方文档中对于脚本引擎 API 的描述通常较为粗略, 如针对脚本引擎 API 参数的描述中仅说明了参数的基础类型(整数类型, 布尔类型, 对象类型等), 而没有说明参数的细粒度类型。根据统计, 在 Adobe Reader 内置脚本引擎中总共包含 163 种不同类型的内置对象。当一个脚本引擎 API 的参数为对象类型时, 能接收脚本引擎中所有类型的内置对象, 且不会提示类型错误。然而, 当输入参数的细粒度类型与脚本引擎 API 参数需求的细粒度类型不符时, API 无法正确处理所提供的参数, 从而无法产生对象别名关系。我们发现当为脚本引擎 API 提供正确类型的对象参数时, 涉及的参数内存访问模式能够匹配对应的对象数据结构特征。

我们按照脚本引擎内置对象具体数据结构的不同, 划分内置对象的细粒度类型。相较于粗粒度类型, 即基础类型(如对象类型, 字符串类型, 整型等), 细粒度类型能够更全面地展现不同脚本引擎内置对象的差异。使用内置对象细粒度类型, 我们可以在调用脚本引擎 API 时针对性地搭配内置对象, 在最大程度上保证 API 调用具有合法语义, 同时也减少了运行时错误的产生, 间接增加了代码覆盖率。

在获得脚本引擎 API 对象参数信息之后, 将生成脚本引擎 API 调用语句。对于 API 需求的对象类型参数, 我们根据参数候选集, 选取类型符合的内置对象作为输入。当参数包含多个候选对象时, 需分别使用不同候选对象生成对应的 API 调用语句。同时, 在单个测试样本的生成过程中, 我们以列表的形式记录已生成的内置对象, 方便后续使用。当引用新的静态内置对象或由于 API 调用产生了新的动态内置对象时, 需要相应地更改对象列表的内容。对于其他基础类型的 API 参数(整数类型, 字符串类型等), 我们通过构建常量表的方式, 每次采用随机策略从常量表中选择类型合适的数据填充。

4.3.1 API 参数内存访问模式

对于 API 参数内存访问模式 *M*, 我们采用如下定义:

M = <函数名, 参数位次, 内存解引用序列>

其中函数名为具体脚本引擎 API 的名称, 参数位次代表参数在 API 调用时对应的序号, 而内存解引用序列为该参数被使用时, 按照层级的访问顺序, 依次对应层级跳转时的内存偏移。

以 Collab 内置对象下属 getInitiatorSource 方法为例, 该方法在使用时需要两个 Stream 类型或类 Stream 类型的参数, 而官方文档中仅说明了两个参数为基础对象类型。对于 Collab.getInitiatorSource 方法中参数的内存访问模式和对应的对象数据结构特征, 我们总结如图 7 所示。以 Collab.getInitiatorSource 的第二个参数为例, 提取出的内存访问模式可表示为<Collab.getInitiatorSource, 2, [0x0, 0x4]>。

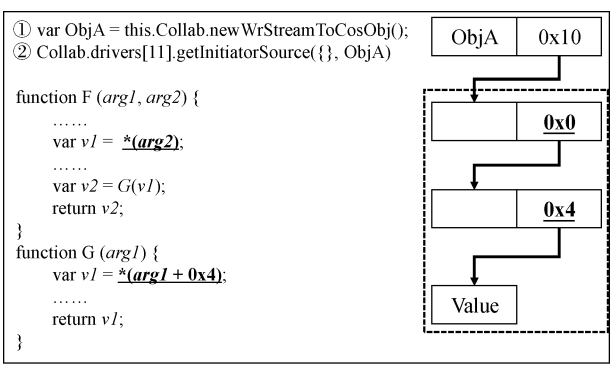

图 7 API 参数内存访问模式

Figure 7 Memory access pattern of API parameter

在图7左侧标号①②的语句为具体脚本引擎 API, 在语句①中通过 API Collab.newWrStream ToCosObj 创建了 FileStream 类型的内置对象 *ObjA*, 并作为语句②中 API 的第二个参数使用。图7左侧下半部分为 API Collab.getInitiatorSource 对应函数体的伪代码, 函数体中的*号代表指针解引用操作, 即读取指针指

向内存单元中的数据。在脚本引擎 API Collab.getInitiatorSource 对应的函数体中, 首先对第二个参数进行了指针解引用操作, 并将读取的值存储于变量 *v1*中, 其内存访问模式偏移为0x0(下划线部分)。之后, 通过将变量*v1*传给函数G, 并将函数的返回值赋值给变量 *v2*。函数 G 针对传入的参数再次进行了指针解引用操作, 此时的内存访问模式偏移为0x4(下划线部分)。据此, 可以发现 Collab. getInitiatorSource 对应的函数 F 中对第二个参数进行了两次指针解引用操作, 对应的内存访问模式偏移分别为0x0以及0x4。

在图7右侧为FileStream类型内置对象的部分数据结构特征, 其中存在 0x0 和 0x4 的特征信息(虚线框部分, 分别为第二层级到第三层级, 以及第三层级到第四层级对应的内存偏移), 与 Collab. getInitiatorSource 函数第二个参数的内存访问模式相匹配。同时, 不同内置对象的数据结构特征拥有不同的内存偏移信息, 而不同脚本引擎 API 中对象参数的内存访问模式也不同。因此, 可以通过匹配脚本引擎 API 对象参数的内存访问模式与内置对象内存结构特征, 推断脚本引擎 API 对象参数的候选列表。

对于两个不同类型的内置对象, 其数据结构特征中包含的偏移信息分别为(0x1, 0x2, 0x4)以及(0x1, 0x2, 0x6, 0x8), 而脚本引擎API对象参数内存访问模式对应的偏移为(0x1, 0x2), 则该脚本引擎 API 参数候选集将同时包含上述两种内置对象。精确的静态分析技术需要考虑多种因素, 如控制流信息, 数据依赖等, 针对大型脚本引擎存在性能开销过大的问题。我们的目标在于尽可能地缩小测试样本生成时 API 的搜索空间, 因此我们提出了一种上下文不敏感, 路径不敏感, 流不敏感的过程间数据流分析方法, 用于提取 API 对象参数的内存访问模式。基于内存访问模式, 可以获得参数类型的候选列表, 即正确参数类型的超集。该方法在缩小脚本引擎 API 参数搜索空间的同时, 保证覆盖正确的 API 对象参数类型, 减少参数类型错误导致的运行时异常。接下来, 我们将介绍如何提取脚本引擎 API 对象参数的内存访问模式。

4.3.2 提取 API 参数内存访问模式

不同于已有的基于控制流图或程序调用图的静态分析技术, 我们将重心放在提取脚本引擎 API 对象参数的内存访问模式, 主要关注 API 实现体中与对象类型参数相关的处理逻辑。

我们的分析方法为上下文不敏感以及路径不敏感, 因此我们不关注 API 实现体中的分支或循环等语句的控制条件, 粗粒度地认为 API 实现体中每一条语句均有被执行的可能。基于数据流分析的思路, 我们认为 API 实现体中涉及对象参数的语句均为分析目标。我们提出的分析方法为过程间的数据流分析, 因此遇到函数调用时, 需要展开函数调用, 嵌套分析函数调用内部的处理逻辑。在汇编语言中, 函数调用又可细分为直接函数调用以及间接函数调用。直接函数调用以函数名为句柄调用函数, 能在静态分析时直接展开; 间接函数调用通过函数表的方式进行函数调用, 仅能在动态执行时展开。接下来, 以图 8 的一段函数实现体为例, 具体介绍所使用的静态分析方法。

```
function F (arg1, arg2)
{
    var a = G (arg1);
    ……
    H (arg2);
    return a;
}
function H (arg2)
{
    var a = *(arg2 + 8);
    ……
    return 0;
}

function G (arg1)
{
    var a = *(arg1 + 4);
    var b = *(a + 8);
    ……
    return b;
}
```

图 8 直接函数调用实例

Figure 8 Example of direct function call

在图 8 中, 脚本引擎 API 对应的实现体 F, 其使用两个参数分别用 *arg1* 和 *arg2* 表示, 其中 *arg1* 和 *arg2* 对应的参数类型为对象。为了获取参数的内存访问模式, 需要跟踪函数实现体中 *arg1* 参数的使用。在函数实现体 F 中, *arg1* 被作为另一函数 G 的参数使用, 因而需要进一步展开函数 G 以分析函数 G 中参数 *arg1* 的使用情况。此处, 函数 G 表现为直接函数调用, 可以直接展开获得具体函数实现。对于间接调用的情况, 由于无法获取具体函数实现, 我们将在稍后进行说明。

在函数 G 中针对 *arg1* 进行了两次指针解引用操作, 并最终返回指针解引用的结果。两次指针解引用分别对应的固定偏移值为 0x4 和 0x8。因此, 我们将偏移值 0x4 和 0x8 作为函数 G 中 *arg1* 参数的内存访问模式, 记为<G, 1, [0x4, 0x8]>。同时, 该固定偏移与内置对象数据结构特征中的偏移对应, 代表着对象参数的类型信息。同理, 可以提取函数 H 中参数的内存访问模式, 记为<H, 2, [0x8] >。对于函数 F, 通过分析可以得知, 参数 *arg1* 被函数 G 使用, 而参数 *arg2* 被函数 H 所使用。结合函数 G 和函数 H 提取出

的参数内存访问模式, 函数 F 的参数内存访问模式可以归纳为<F, 1, [0x4, 0x8]>和<F, 2, [0x8]>。进一步, 我们称函数 F 为父函数, 函数 H 和函数 G 为子函数, 父函数参数的内存访问模式为所有子函数参数内存访问模式的并集。对于间接调用的情况, 以图 9 中代码为例进行说明。

```
function P (arg1) {
    var b = (Indirect-Call) (arg1);
    ……
    return Q(b);
}
function Q (arg1) {
    var c = *(arg1 + 12);
    var d = *(c + 8);
    ……
    return d;
}
```

图 9 间接调用函数实例

Figure 9 Example of indirect function call

在函数 P 的实现体中, Indirect-Call 代表间接函数调用, 其将 *arg1* 作为参数使用, 并将函数调用的返回值赋值给了变量 *b*, 后续变量 *b* 作为直接函数调用 Q 的参数使用。对于图 9 中的间接函数调用, 由于无法获得具体函数实现体进行参数分析, 我们粗粒度地认为变量 *b* 等同于参数 *arg1*, 即变量 *b* 提取出的内存访问模式也作为参数 *arg1* 的内存访问模式。进一步, 通过分析直接函数调用 Q 的实现体, 可以提取出函数 Q 中参数 *arg1* 对应的内存访问模式, 记为<Q, 1, [0x12, 0x8]>。最终, 可以得到变量 *b* 对应的内存访问模式为<Q, 1, [0x12, 0x8]>, 从而可以获得函数 P 中参数的内存访问模式为<P, 1, [0x12, 0x8]>。

使用静态分析的方式结合所有脚本引擎 API 的实现体, 可以提取出对象类型参数的内存访问模式。进一步, 将内存访问模式与对象数据结构信息匹配, 获得脚本引擎 API 参数的细粒度信息。由于使用粗粒度的提取方式, 最终提取结果为正确参数类型的超集, 如针对 Collab.getInitiatorSource 这一 API, 其正确参数类型为 FileStream 类型, 而提取出的参数候选集中包含 FileStream 类型, Stream 类型以及 ReadStream 类型。除此之外, 可以用同样的方法提取脚本引擎 API 返回值的内存访问模式, 并采用同样的方式进行存储(对于 API 返回值不像参数一样区分序号, 将统一用 R 代替, 如<print, R, [0x12, 0x4]>)。

4.4 识别对象别名关系

我们将 3.3 中生成的测试样本输入脚本引擎, 动态识别测试样本运行过程中的内置对象别名关系。由于内置对象别名关系的本质在于不同对象内部的共享内存单元, 相较于已有别名关系识别方法, 我们将内置对象数据结构特征转化为对应的数据结构特征图, 根据数据结构特征图状态的变化来高效准确地识别内置对象别名关系。

在该节中, 我们着重表述如何依据数据结构特征图识别内置对象别名关系。首先, 对于脚本引擎中的每个内置对象, 按照 3.2 节中描述的方法, 获取对应的数据结构特征图。当出现对象别名关系时, 不同对象对应的数据结构特征图之间将出现新的连通节点, 对应别名关系中的共享内存单元。接下来, 依据建立对象别名关系前后, 对象数据结构特征图的变化来具体说明如何识别对象别名关系。以 Stream 类型和 ReadStream 类型的内置对象为例, 在建立别名关系之前, 其数据结构特征图如图 10 所示。

在图 10 中左半部分 Obj1_Id 标识的数据结构特征图代表 Stream 类型的内置对象, 右半部分 Obj2_Id 标识的数据结构特征图代表 ReadStream 类型的内置对象, Obj1_Id 以及 Obj2_Id 分别为内置对象运行时的内存地址。由于执行脚本引擎 API 引起了内置对象的变化, 需要重新计算内置对象对应的数据结构特征图。当执行完某条脚本引擎 API 之后, Stream 对象和 ReadStream 对应数据结构特征图的变化如图 11 所示。

在图 11 中用我们虚线框代表别名关系对应的节点, 即别名关系对应的共享内存单元。我们采用图 12 中描述的算法, 通过对比脚本引擎 API 执行前后对象数据结构特征图的状态变化, 识别内置对象别名关系。

我们设置监控粒度为单条脚本语句, 如图 12 所示, 首先我们初始化最终输出的别名关系内置对象集合 *R* 为空集。在执行脚本引擎 API 前, 之后, 依据内置对象内存地址集合 *S*, 计算对应的对象数据结构特征图集 G^A(对应算法中步骤 2,3,4)。当执行完某条脚本语句后, 再次计算内置对象数据结构特征图集, 记为 G^B。最后, 以对象内存地址为索引, 对比先后两次生成的对象数据结构特征图, 根据数据结构特征图的变化来识别内置对象别名关系。

当建立内置对象别名关系时, 共享内存单元对应的数据结构特征图节点(图 11 中虚线框节点)的祖先属性将发生变化。因为共享节点可以归属于不同内置对象, 所以该节点的祖先属性将增多(对应图 12 中步骤 11)。我们依据脚本引擎 API 执行前后是否存在数据结构特征图节点的祖先属性符合上述变化, 来判断是否建立了内置对象别名关系。

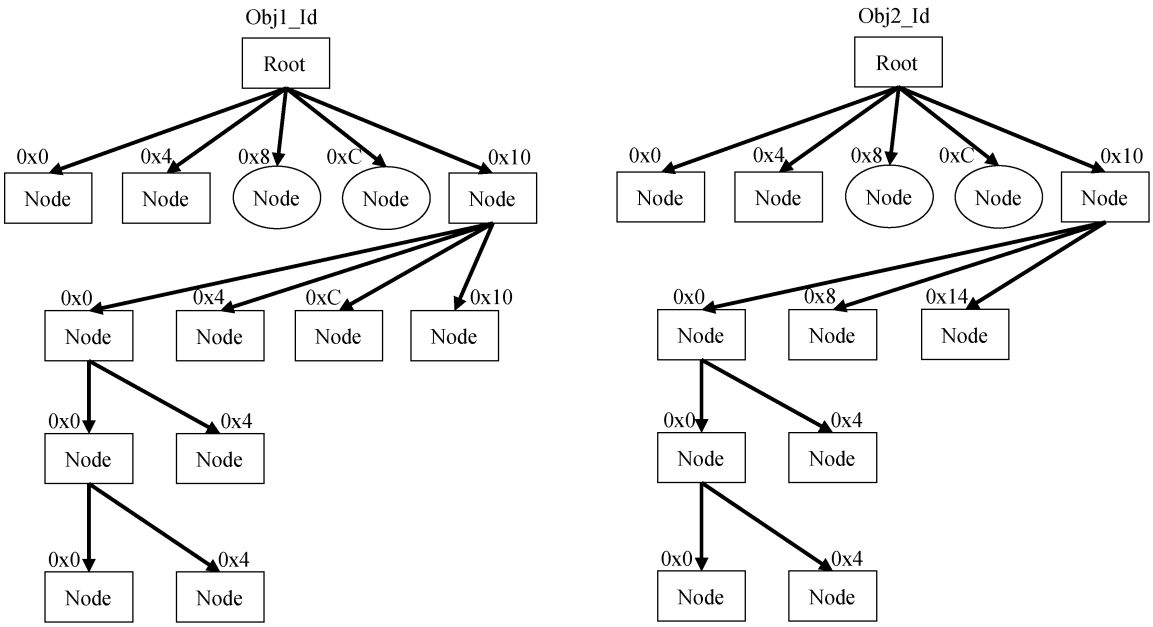

图 10　建立别名关系前的数据结构特征图

Figure 10　Data structure signature graph before establishing alias relationship

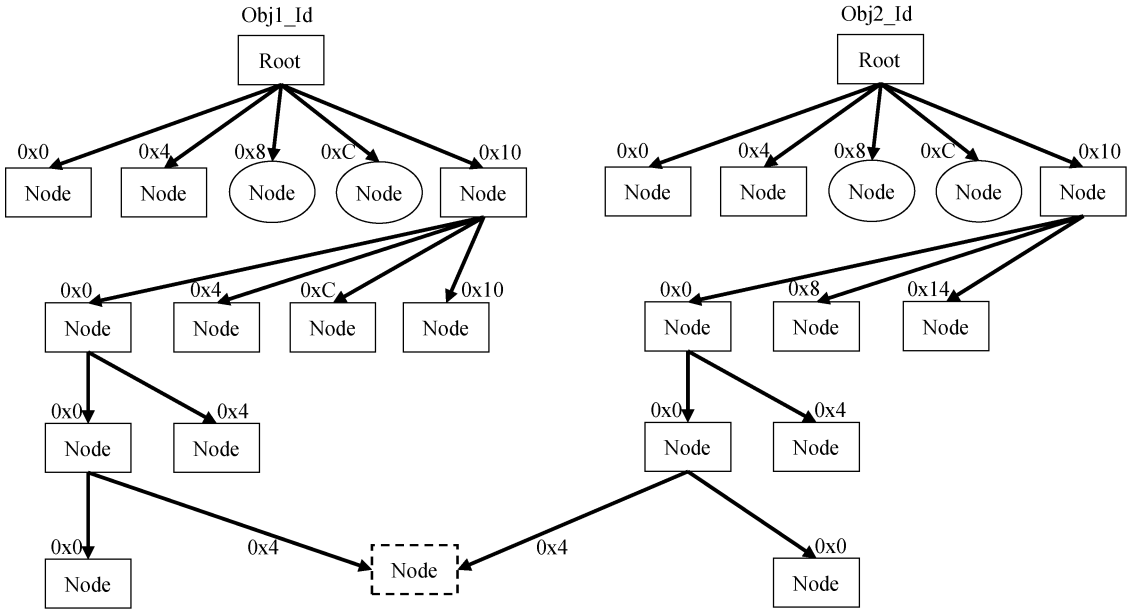

图 11　建立别名关系后的数据结构特征图

Figure 11　Data structure signature graph after establishing alias relationship

通常, 在单个测试样本中, 涉及的脚本引擎内置对象数量较少, 因此每次重新计算数据结构特征图的性能开销较小。相比于已有的别名关系识别方法, 我们提出的基于数据结构特征的方法具有更高的识别效率。

5　利用别名关系检测释放后使用漏洞

内置对象别名关系有助于检测与对象内存操作相关的程序漏洞。通过建立内置对象别名关系, 并搭配合适的脚本引擎 API 释放内置对象, 可以在多个对象内部引入悬挂指针, 进而触发释放后使用漏洞。该部分具体分为三个部分进行介绍, 分别为释放后使用漏洞的检测流程, 提取特殊 API 序列以及生成测试样本检测释放后使用漏洞。

5.1　释放后使用漏洞的检测流程

在释放后使用漏洞中, 根本原因是访问了被释

算法 1. 内置对象别名关系识别方法
输入：活跃对象集合S
输出：存在别名关系的内置对象集合R

G 代表对象数据结构特征图集，G^A 和G^B 代表执行JavaScript代码前后的结果

1. Initialize $R=\{\}$, $G^A=\{\}$,$G^B=\{\}$
2. FOR $S_i \in S$:
3. $G_i^A = \text{GenGraph}(S_i)$
4. $G^A = G^A \cup G_i^A$
5. Execute javaScript code
6. S'= update( )
7. FOR $S_i' \in S'$
8. $G_i^B = \text{GenGraph}(S_i')$
9. $G^B = G^B \cup G_i^B$
10. FOR $G_i^A \in G^A, G_i^B \in G^B$
11. IF $G_i^B.Ancient - G_i^A.Ancient \neq \Phi$ THEN
12. $R = R \cup (G_i^B.Ancient - G_i^A.Ancient)$

图 12 对象别名关系识别算法

Figure 12 Algorithm of object alias relationship recognition

放内存区域, 从而触发程序错误。对于释放后使用漏洞, 必不可少的环节为释放对象产生悬挂指针以及访问悬挂指针。同时, 必须遵循释放操作在前, 使用操作在后的原则, 否则无法触发释放后使用漏洞。因此, 需要针对内置对象提取释放 API 序列以及使用API序列, 并基于提取的API序列生成测试样本以检测释放后使用漏洞, 整个系统的具体检测流程如图 13 所示。

整个检测流程按从左到右的顺序执行, 核心在于利用内置对象别名关系。为了提取释放对象 API 序列以及使用对象 API 序列, 需要使用不同的判断逻辑。进一步, 利用提取出的释放序列和使用序列生成测试样本以检测脚本引擎中的程序漏洞。

对于建立的别名关系的两个内置对象(记为内置对象 *A* 以及内置对象 *B*), 分别获取 API 测试样本运行时的对象内存地址。进一步, 依据对象内存地址以及对象数据结构特征, 获取别名内存单元对应的内存地址, 又称为别名地址。我依据 API 测试样本运行时是否满足设定的内置对象释放条件来提取释放对象 API 序列(具体条件在 4.2 节中进行描述)。在提取使用对象 API 序列时, 我们针对别名地址设置内存访问断点。通过监控测试样本运行过程中内存断点的触发情况, 获取使用对象 API 序列。

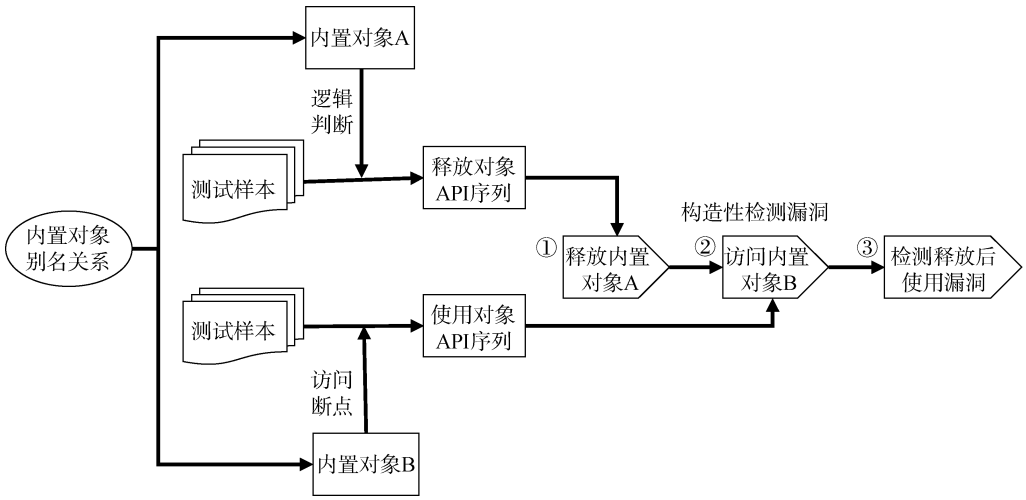

图 13 检测释放后使用漏洞流程图

Figure 13 Flow chart of detecting UAF vulnerability

最后, 需要按照一定模式组合释放对象 API 序列和使用对象API序列, 并在释放对象API序列执行前建立内置对象别名关系, 以检测脚本引擎中的释放后使用漏洞。

5.2 提取特殊 API 序列

在本节中, 我们主要叙述如何提取释放对象API序列以及使用对象API序列, 提取过程对应的具体流程如图 14 所示。我们首先使用动态插装技术获得内置对象内存地址, 之后结合建立的对象别名关系获得别名单元的内存地址。根据测试样本运行过程中, 别名单元内存地址是否满足特定释放/使用模式, 提取对应的释放/使用 API 序列。

5.2.1 提取释放 API 序列

首先介绍释放对象 API 序列的提取, 我们着重关注对象内部共享内存区域的释放情况, 通过判定对象内部共享内存区域是否被释放, 来提取释放对象 API 序列。我们具体使用 3 种检查策略来判定共享内存区域是否被释放, 分别为判定别名区域中数

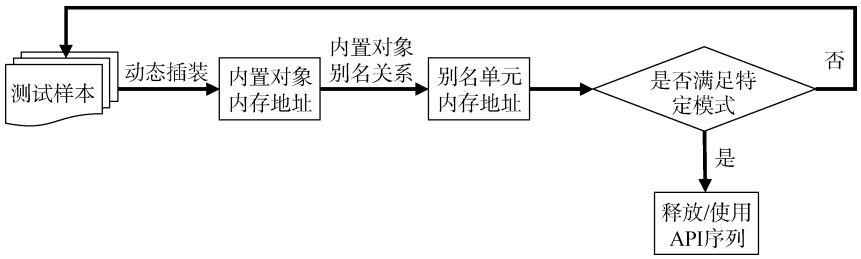

图 14　提取特殊 API 序列流程图

Figure 14　Flow chart of extracting particular API sequence

据的状态, 判定内置对象存活性以及判定系统释放函数是否触发。

我们将根据 3.2 节中提取的 API 参数信息, 随机生成 API 调用序列。通过归纳已知释放对象 API 序列的特征, 在填充 API 参数时, 尽量选择空值作为参数, 如长度为 0 的字符串, 数字 0 等。除此之外, 还可以利用脚本引擎垃圾回收机制释放内置对象。通过消除内置对象的引用或引入临时变量的方式可以影响对象生命周期, 如将指向内置对象的变量指向空对象, 或不使用变量存储 API 返回值, 进而触发脚本引擎垃圾回收机制释放内置对象。

在第一种检查策略中, 我们根据别名区域中数据的状态来判定对象是否被释放。当内置对象被系统回收时, 对象内存区域中的数据被填充为特殊值, 该特殊值可通过简单人工分析获得。根据别名地址中的数据是否等于该特殊值, 可以判断别名区域是否被释放, 进而判定 API 序列是否触发了释放行为。

第二种检查策略直接判断内置对象存活性。在第2章中我们介绍了活跃对象集合的概念, 通过检查内置对象是否从活跃对象集合消失来判定 API 序列是否触发了释放行为。

第三种检查策略为监控系统释放函数。脚本引擎本质为操作系统中的进程, 一切内存操作均需调用操作系统的底层函数, 即使用 API 创建和释放内置对象的同时, 也调用了操作系统的创建及释放函数。对于操作系统底层的释放函数, 其参数为需要释放的内存地址。可以通过监控底层释放函数的调用, 判断参数值是否等于对象别名单元内存地址, 决定当前 API 序列是否触发了释放行为。

在提取释放对象 API 序列的过程中, 我们依旧选择单个脚本引擎 API 作为插装粒度, 并在单条 API 调用语句前后加入检查点。每当遭遇检查点时, 需要实施上述三种检查策略, 判断当前情景下是否发生了内置对象释放行为。同时, 可以在单个测试样本中同时监控多个存在别名关系的内置对象, 提高测试效率。

5.2.2　提取使用 API 序列

在提取使用对象 API 序列时, 无需关注对象的状态, 即对象存活与否与是否能访问别名区域中的数据无直接关系。虽然, 正常的使用规范要求对象被释放后无法访问对象内部数据。但是, 当内置对象被释放后仍能成功访问其内部数据, 说明该内置对象未设置相应的释放标志位, 因此更容易产生悬挂指针。综上, 在提取使用对象 API 序列时, 仅关心对象内部的别名区域是否能被成功访问。

接下来介绍如何提取使用 API 序列。为了提取使用 API 序列, 在生成 API 测试样本时, 将待测内置对象作为 API 参数使用以满足内置对象访问条件。我们利用 Windbg 调试工具提供的内存读写监控功能来判断当前 API 序列是否能够访问特定内置对象。通过将内置对象内部别名单元内存地址设置为读断点, 依据 API 测试样本运行过程中是否触发了读断点, 提取对应的使用 API 序列。我们针对每一对存在别名关系的内置对象, 分别提取对应的使用 API 序列。

5.3　生成测试样本检测释放后使用漏洞

为了检测脚本引擎中的释放后使用漏洞, 需要构造特殊的脚本引擎 API 调用序列以满足释放后使用漏洞的模式, 对应到脚本引擎中为先调用释放对象 API 序列, 后调用使用对象 API 序列。由于单悬挂指针导致的释放后使用漏洞容易被修补, 需要引入内置对象别名关系以创建多悬挂指针, 具体的测试样本生成策略如图 15 所示。

对于存在别名关系的内置对象(记为对象 *A* 以及对象 *B*), 我们分别搭配对应的释放 API 序列集合以及使用 API 序列集合, 如图 15 中对象 *A* 搭配释放 API 序列集合, 对象 *B* 搭配使用 API 序列集合。按照第 2 章中介绍的别名关系释放后使用漏洞触发模式, 我们选取释放 API 序列集合和使用 API 序列集合中的元素进行组合, 并遵循释放在前使用在后的顺序, 生成不同测试样本。同时, 在执行释放 API 序列之前需要先建立内置对象别名关系。

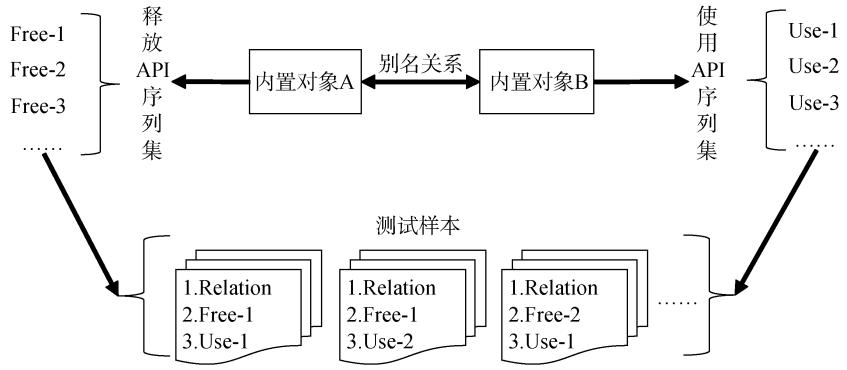

图 15　测试样本生成策略

Figure 15　Test case generation strategy

在实验中我们发现, 为了提高内置对象间别名关系的出现概率, 需要脚本引擎 API 间需要具有较强的关联性。通过提取 API 参数和返回值的内存访问模式, 可分别获得参数以及返回值的候选对象列表, 记为 Param_Set 以及 Return_Set, 而不同脚本引擎 API 之间 Param_Set 和 Return_Set 存在交集。在生成具体测试样本时, 我们考虑将脚本引擎 API 的返回值作为另一 API 的参数, 以增加 API 之间的关联性。同时, 为了保证合理语义, 如 API-1 的 Param_Set 与 API-2 的 Return_Set 存在交集, 在生成测试样本时, 将优先调用 API-2 创建内置对象, 再将 API-2 的返回值作为 API-1 的参数。最后, 将产生的测试样本输入脚本引擎, 以检测脚本引擎中的释放后使用漏洞。

6　实验结果评估

本章中, 我们针对论文中提出的内置对象别名关系识别方法设计多项评估实验, 分别从别名关系准确率, 测试样本生成效率以及别名关系识别开销三个方面进行评估。同时, 对于提出的释放后使用漏洞检测方法, 我们在真实软件 Adobe Reader 内嵌 JavaScript 引擎上实施了漏洞检测实验, 评估该方法检测释放后使用漏洞的性能, 验证该方法的有效性和可用性。

6.1　别名关系准确率

论文的一个核心点在于利用对象数据结构特征建立并识别内置对象别名关系, 因此需要评估内置对象别名关系的准确性。

为了评估测试样本中内置对象别名关系是否准确, 我们生成了长度相同的测试样本, 分批识别其中的内置对象别名关系。对于每个待测样本, 依据提取出的脚本引擎 API 参数类型信息, 生成 100 条脚本引擎 API 调用语句, 并动态监测样本运行过程中是否建立了内置对象间别名关系。最终, 我们统计了 100 个测试样本的运行结果, 总计包含 18354 条别名关系记录, 平均每个测试样本包含 183 条别名关系, 平均每条脚本引擎 API 生成了 1.8 个内置对象别名关系。通过对 18354 条别名关系记录进行去重, 我们最终提取出 284 条不同的内置对象别名关系, 共涉及 27 个不同类型的内置对象, 占动态内置对象比例的 51%。

我们采用图 16 中的模式来评估内置对象别名关系识别的准确性。针对一组待验证别名关系的内置对象, 变更其中一个内置对象的状态, 如改变对象属性值等, 并监控另一内置对象的变化。依据待验证别名关系的内置对象状态是否同步变化, 来判定内置对象别名关系是否准确。

存在别名关系的两个内置对象分别记为 *ObjA* 和 *ObjB*, 并将 *ObjA* 对象与 *ObjB* 对象最终存在别名关系的内存单元抽象为 *ObjA.X.Y* 以及 *ObjB.E.F*(对应图 16 中共享内存单元 *M*)。初始时, 共享内存单元 *M* 中存储的数值为 1。我们使用脚本引擎 API 对 *ObjA.X.Y* 进行赋值操作, 更改 *M* 中的数值为 2。之后, 通过 *ObjB.E.F* 读取 *M* 中存储的数值, 并判断读取的数值是否等于更改后的数值。当读取的数值为 1 时, 我们认为识别 *ObjA* 对象和 *ObjB* 对象间别名关系时产生了误报; 当读取的数值为 2 时, 我们认为对象别名关系识别正确。我们采用如上方式对提取出的 284 条内置对象别名关系进行了评估, 并人工审查了评估结果。最终, 我们证明了提取出的 284 条内置对象别名关系真实存在, 即提出的内置对象别名关系识别方法没有产生误报。

同样, 我们采用图16中的模式来评估文中提出的别名关系识别方法是否存在漏报现象。我们将测试范围设置为全体内置对象, 即脚本引擎中存活的所有静态内置对象以及动态内置对象。通过运行测试脚本代码, 并统计不同状态下脚本引擎中存在的所有内置对

象别名关系, 以此作为对比的标准。在实验过程中, 针对全体内置对象别名关系进行多次统计, 排除内置对象随机初始化数值引发的干扰。在统计多次实验结果后, 平均每个测试样本中包含总计196条内置对象别名关系, 使用论文中的别名关系识别方法能识别出183条内置对象别名关系, 最终漏报率约6.6%。

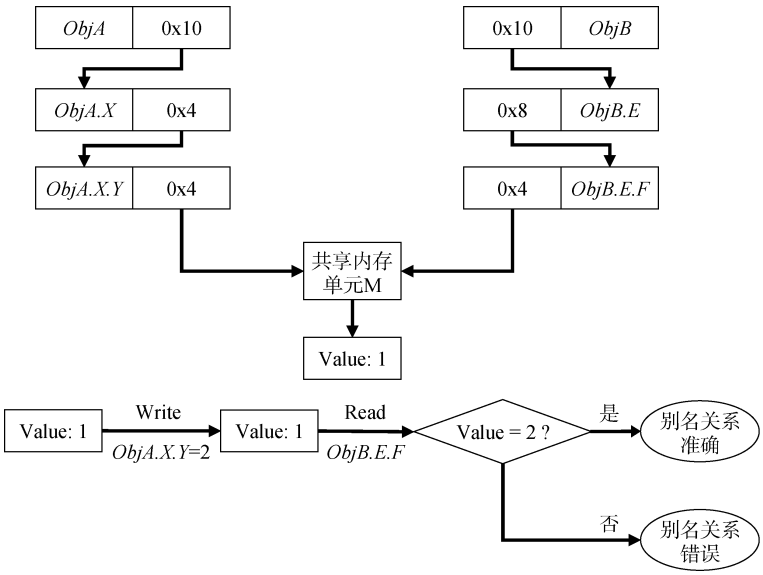

图 16　判定别名关系准确性流程图

Figure 16　Flow chart of judging alias relationship accuracy

进一步, 我们将本文中提出的别名关系识别方法与现有研究中的别名关系识别方法进行对比, 比较不同方法在识别脚本引擎内置对象别名关系时的性能差异。我们选用 Pintool 工具[31]实现现有研究中的别名关系识别方法[32], 核心思想为监测程序运行过程中的内存读写操作。当监测到内存写指令涉及的不同操作数分别属于不同内置对象时, 认为出现了内置对象别名关系。在保持测试样本一致的前提下, 我们分别统计了相同时间内, 不同别名关系识别方法的相关指标, 包括别名关系数量、误报率及漏报率, 同时将测试样本中别名关系总数作为对照标准, 最终统计结果如表 2 所示。

表 2　别名关系识别方法对比

Table 2　Comparison of different alias relationship recognition method

测试时长(min)	方法对比	别名关系数量	误报率(%)	漏报率(%)
10	传统方法	17	0	88
	本文方法	137	0	6
	别名关系总数	146	—	—
30	传统方法	48	0	79
	本文方法	221	0	7
	别名关系总数	237	—	—
60	传统方法	71	0	71
	本文方法	231	0	8
	别名关系总数	251	—	—

从结果中可见, 在测试时长为 10 min 时, 使用传统别名关系识别方法, 总共识别出 17 组内置对象别名关系, 漏报率为 88%, 而使用论文中提出的基于数据结构特征的别名关系识别方法能够识别出 137 组别名关系, 漏报率为 6%, 明显优于传统方法。同样, 我们分别测试了 30 min 以及 60 min 条件下, 不同别名关系识别方法所能识别出的别名关系数量, 最终证明了论文中的别名关系识别方法相较于传统方法有较大优势。

6.2　测试样本生成效率

本节中, 我们主要评估使用对象数据结构特征对于生成测试样本的影响。当脚本引擎 API 参数为对象类型, 生成测试样本时搜索空间的大小与 API 参数候选集的大小成正比。在不提取细粒度参数类型的情况下, API 参数候选集的大小等于脚本引擎中所有内置对象的个数。依据统计结果, 平均一个脚本引擎 API 需要 3 个对象参数, 而脚本引擎中共包含 237 个 API 以及 163 种内置对象, 则生成测试样本时最终搜索空间大小为 237*163*163*163。我们通过实验发现, 不同脚本引擎 API 需要不同细粒度类型的内置对象作为参数, 而传递其余类型的内置对象将引发运行时错误。通过提取脚本引擎 API 参数的内存访问模式, 结合内置对象数据结构特征, 缩小了脚本引擎 API 对象参数的候选集大小。

我们分析并提取了每个脚本引擎 API 的细粒度

参数信息, 最终确定了每个脚本引擎 API 中对象参数的候选集大小, 最终的统计结果如图 17 所示。根据图 17 中的统计结果可以发现, 约 80%脚本引擎 API 的参数候选集的大小不超过 6。通过提取脚本引擎 API 细粒度参数信息, 能明显优化生成测试样本时的搜索空间, 大幅减少待测样本个数(从 237*163*163*163 缩小为 237*6*6*6), 提高了测试效率。

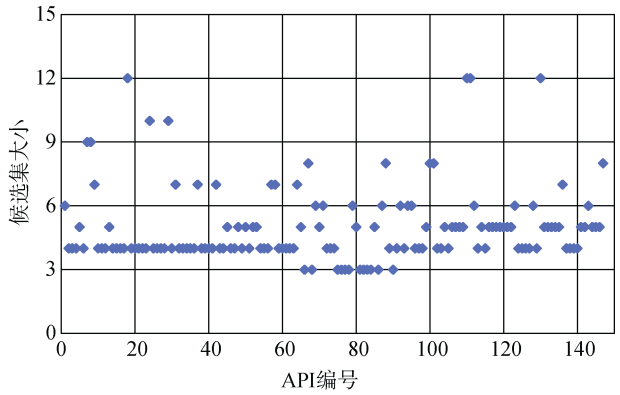

图 17 API 参数候选集统计图

Figure 17 Statistics of API parameter candidate set

除此之外, 提取脚本引擎 API 细粒度参数信息对于减少运行时错误有明显助益。我们分别在使用 API 细粒度参数信息与不使用的情况下, 生成相同数量的测试样本, 并统计测试样本运行过程中产生的运行时错误。通过对比运行时错误数量, 评估脚本引擎 API 细粒度参数信息对于减少运行时错误的效果, 具体统计结果如表 3 中所示。

表 3 运行时错误数量

Table 3 Number of runtime errors

测试样本数量	100 个	200 个	300 个	400 个
使用参数信息	17	32	46	57
不使用参数信息	73	153	231	316

在表 3 中, 我们分别生成了 100 个, 200 个, 300 个以及 400 个测试样本, 并统计产生运行时错误的样本数量。在使用 API 参数细粒度信息时, 存在运行时错误的样本数量分别为 17 个, 32 个, 46 个以及 57 个, 而在不使用 API 细粒度参数信息时, 运行时错误的数量分别为 73 个, 153 个, 231 个以及 316 个。可以发现, 使用 API 细粒度类型信息生成的测试样本中, 包含的运行时错误数量明显少于不使用细粒度参数信息。因此, 可以证明提取脚本引擎 API 细粒度信息对于减少运行时错误, 提高测试样本质量有明显帮助。

6.3 别名关系识别开销

为了评估论文中别名关系识别方法的高效性, 即相较于已有别名关系识别方法, 我们提出的别名关系识别方法具有较短的检测时长。我们生成了包含随机数量脚本语言代码的测试样本, 统计了测试样本在不识别别名关系, 使用传统别名识别方法以及基于对象数据结构特征识别别名关系三种场景下的运行时长, 取单条脚本语言代码平均运行时长作为性能评估的依据。

传统别名关系识别通过监控内存数据的方式进行实现, 具体为监控程序运行过程中的内存读写指令, 根据读写指令涉及的操作数来识别对象别名关系。内存读指令用于标明操作数与对象的从属关系, 内存写指令用于判断对象别名关系。当内存写指令涉及的两个操作数分别来自不同对象时, 认为检测到对象别名关系。我们总计测试了 100 组数据, 结果如图 18 所示。

在图 18 中, 运行耗时以秒为单位。在不识别内置对象别名关系时, 单条脚本语言代码的平均运行时长约为 1 s。使用传统别名识别方法时, 单条脚本语言代码的平均运行时长为 269 s, 相比于不识别别名关系, 其运行耗时明显增加。采用我们提出的基于对象数据结构特征的别名关系识别方法, 单条脚本语言代码的平均运行时长为 13 s。虽然相比于不进行别名分析运行时长有所上升, 但额外提取了内置对象别名关系, 且性能开销仍在可接受范围。同时, 相比于传统别名关系识别方法能显著缩短运行时长。因此, 我们提出的基于对象数据结构特征的别名关系识别方法, 能快速检测脚本引擎内置对象别名关系, 提升了别名关系识别效率。

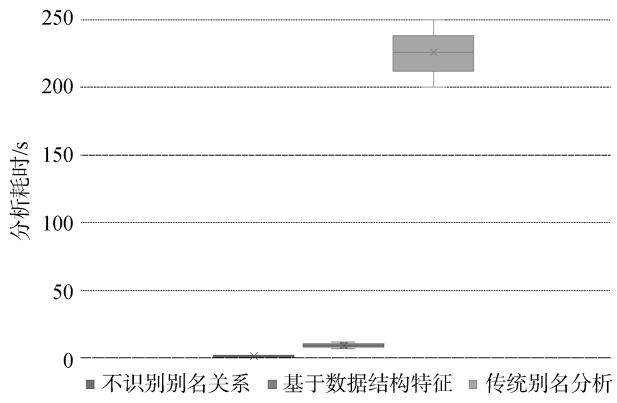

图 18 别名关系识别效率对比

Figure 18 Comparison of alias relationship identification efficiency

6.4 别名关系识别方案的通用性

在软件内嵌脚本引擎中, 为了方便用户, 定义了不同结构的内置对象与不同功能的脚本引擎 API, 搭配使用脚本引擎 API 与内置对象可实现多种复杂功能。根据我们的调查结果, 脚本引擎中的安全漏洞常与内置对象紧密相关, 究其原因在于内置对象的内部结构复杂, 且可通过脚本引擎 API 建立深层次联系, 使得管理和维护内置对象具有较大难度。在本文中, 为了检测脚本引擎中的深层次安全漏洞, 我们将重心放在识别内置对象别名关系。进一步, 为了证明论文中别名关系识别方法的通用性, 我们设计了如下实验。

在软件内嵌脚本引擎中, 除了具有特殊结构的内置对象, 还存在用户自定义对象, 即由用户自定义对象内部结构及属性。我们采用包含自定义对象的数据集, 并使用论文中方法识别内置对象别名关系。进一步, 使用 6.1 节中描述的方法来验证别名关系的准确性, 最终实验结果如表 4 所示。

表 4 自定义对象别名关系识别统计表

Table 4 Statistics of custom object alias relationship

对象别名关系总数	自定义对象别名关系总数	识别自定义对象别名关系数量	漏报率(%)	误报率(%)
153	36	33	8.3	0
165	43	39	9.3	0
138	25	23	8	0
144	31	29	6.4	0

如表 4 所示, 在所使用的数据集中, 总共包含对象别名关系 153, 165, 138 以及 144 组, 其中自定义对象别名关系总数为 36, 43, 25 以及 31 组, 而使用论文中的方法共能识别出自定义对象别名关系 33, 39, 23 以及 29。进一步我们计算了自定义对象别名关系识别的漏报率以及误报率, 其中误报率采用 6.1 节中方法计算, 最终证明无误报现象, 而漏报率分别为 8.3%, 9.3%, 8%以及 6.4%, 证明了论文中方法能以较高效率识别自定义对象别名关系, 从而证明方法的通用性。

6.5 释放后使用漏洞检测结果

在本节中, 我们以发现的未知释放后使用漏洞为例, 介绍具体的检测思路。首先是漏洞编号“CVE-2020-3745”的释放后使用漏洞, 具体漏洞代码如图 19 所示。

```
① VAR ObjA = util.streamFromString(' ')
② VAR ObjB = Collab.drivers[9].getInitiatorSource({},
 ObjA)
③ this.resetForm()
④ util.stringFromStream(ObjA.data)
```

图 19 CVE-2020-3745 漏洞触发代码

Figure 19 Triggering code of the CVE-2020-3745 vulnerability

通过调用脚本引擎 API util.streamFromString 创建了 Stream 类型的内置对象, 记为 *ObjA* 对象, 对应图 19 中第①行。在图 19 第②行中, 调用了脚本引擎 API Collab.drivers.getInitiatorSource 并将 *ObjA* 对象作为参数, 创建了 ReadStream 类型的内置对象, 记为 *ObjB*。同时, 该 API 也建立了 *ObjA* 对象和 *ObjB* 对象间的别名关系。之后, 通过调用 API this. resetForm 释放了 *ObjB* 对象。由于 *ObjA* 对象和 *ObjB* 对象存在别名关系, 导致 *ObjA* 对象内部出现了悬挂指针。在图 19 第④行中, 通过 API util. stringFromStream 访问了 *ObjA* 对象内部的悬挂指针, 触发了释放后使用漏洞。我们将上述漏洞的触发模式总结为图 20。

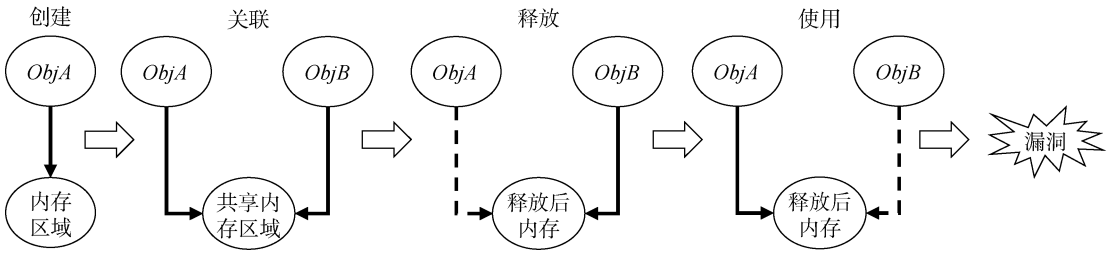

图 20 别名关系漏洞触发模式

Figure 20 Trigger mode of alias relationship vulnerability

通过使用论文中提出的方法, 总计检测出 4 个未知的释放后使用漏洞, 现将每个未知释放后使用漏洞对应的漏洞编号, 及其涉及的具体脚本引擎内置对象信息总结如表 5 所示。

7 总结与展望

在脚本引擎漏洞挖掘领域, 利用模糊测试来检测漏洞的方法已经被证明有效, 但现有的模糊测试

方法主要关注如何生成符合脚本语言语法规范的测试样本, 未能很好地利用脚本引擎 API 以及内置对象, 在挖掘脚本引擎深层次漏洞方面未能取得较好的效果。

表 5 检测出的释放后使用漏洞列表

Table 5 List of detected UAF vulnerabilities

漏洞编号	内置对象别名关系
CVE-2020-3745	Stream 对象与 ReadStream 对象
CVE-2020-3746	app.fs 对象与 app.media.version 对象
CVE-2020-3749	app.fs 对象与 app.calculate 对象
CVE-2020-3750	app.thermometer 对象与 app.monitors 对象

针对上述局限性, 本研究提出了一种基于数据结构特征的脚本引擎内置对象别名关系识别技术, 通过快速识别内置对象别名关系来辅助检测释放后使用漏洞。与传统脚本引擎漏洞检测方案不同, 我们重点关注脚本引擎中的特殊 API 及内置对象, 采取自动化的方式提取内置对象数据结构特征, 并基于提取出的对象特征识别脚本引擎 API 细粒度参数信息。进一步, 搭配合理 API 参数, 我们可以生成高质量的测试样本, 提高内置对象别名关系的出现概率。同时, 我们利用对象数据结构特征快速准确地识别内置对象别名关系, 大幅缩短了现有别名分析技术所需时间。最后, 利用识别的内置对象别名关系, 搭配特定 API 序列有针对性地释放和使用内置对象, 构造性地检测脚本引擎中的释放后使用漏洞。最终, 我们提取出 284 组内置对象别名关系, 并据此检测出 4 个未知的释放后使用漏洞。

诚然, 本方法也存在诸多不足和需要继续改进的地方。首先, 在提取脚本引擎 API 细粒度参数信息时, 现有的静态分析方法在处理间接函数调用时粒度较粗, 提取的参数信息不够精确, 后续可针对识别精度进行改进, 进一步缩小脚本引擎 API 参数候选集的大小, 实现精确的 API 参数细粒度类型识别。

其次, 生成测试样本时, 对于如何填充脚本引擎 API 的参数考虑不足, 如针对基础类型只是简单地选取随机常量值, 对象类型仅考虑了参数候选集中的元素。在后续工作中, 可以更深入地探究脚本引擎API的执行逻辑, 探究参数对于API执行轨迹的影响, 从而更针对性地提供 API 所需参数。除此之外, 使用内置对象别名关系检测释放后使用漏洞时, 提取的内置对象别名关系数量及释放 API 序列数量较少。对于这一问题, 可以通过加强脚本引擎 API 的关联性, 以及改进监控策略, 来发现更多的释放 API 序列。

最后, 本工作未在 JavaScript 以外的脚本引擎中进行实验, 后续考虑将实验移植到 JavaScript 语言以外的脚本引擎中, 进一步证明方法的通用性。在其他脚本引擎中, 可通过分析对象内部结构, 来识别对象内部别名关系, 进而检测安全漏洞。

参考文献

[1] Vulners. Vulnerability Assessment Platform. https://vulners.com/search?query=type:zdi%20acrobat%20javascript, 2020.

[2] Cisco Talos. Vulnerability Reports. https://talosintelligence.com/vulnerability_reports.

[3] S. Groβ. Fuzzil: Coverage guided fuzzing for javascript engines[D]. Technical University of Braunschweig, 2018: 134-142.

[4] Liang H L, Pei X X, Jia X D, et al. Fuzzing: State of the Art[C]. *IEEE Transactions on Reliability*, 2018: 1199-1218.

[5] Holler C, Herzig K, Zeller A. Fuzzing with Code Fragments[C]. *USENIX Security Symposium*, 2012: 445–458.

[6] Dave Aitel. An introduction to SPIKE, the fuzzer creation kit[C]. *The Black Hat USA*, 2002: 33.

[7] MozillaSecurity. Funfuzz. https://github.com/MozillaSecurity/funfuzz, 2008.

[8] Aschermann C, Frassetto T, Holz T, et al. NAUTILUS: Fishing for Deep Bugs with Grammars[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019: 56-71.

[9] Guo T, Zhang P H, Wang X, et al. GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and Structural Mutation[C]. *2013 Second International Conference on Informatics & Applications*, 2013: 212-215.

[10] Han H, Oh D, Cha S K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019: 35-48.

[11] Park S, Xu W, Yun I, et al. Fuzzing JavaScript Engines with Aspect-Preserving Mutation[C]. *2020 IEEE Symposium on Security and Privacy*. 2020: 1629-1642.

[12] Lee S, Han H, Cha S K, et al. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer[C]. *29th USENIX Security Symposium*, 2020: 2613-2630.

[13] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. 2013.

[14] Cha S K, Woo M, Brumley D. Program-Adaptive Mutational Fuzzing[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 725-741.

[15] Wang T L, Wei T, Gu G F, et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[C]. *2010 IEEE Symposium on Security and Privacy*, 2010: 497-512.

[16] Ganesh V, Leek T, Rinard M. Taint-Based Directed Whitebox Fuzzing[C]. *2009 IEEE 31st International Conference on Software*

Engineering, 2009: 474-484.

[17] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing through Selective Symbolic Execution[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 21.

[18] Patrice Godefroid, Michael Y. Levin, David Molnar. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact[C]. *Microsoft Queue*, 2012: 20-27.

[19] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-Aware Evolutionary Fuzzing[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, 2017: 19.

[20] Lemieux C, Sen K. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage[C]. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018: 475-485.

[21] Dinh S T, Cho H, Martin K, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases[C]. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021: 61.

[22] Coppik N, Schwahn O, Suri N. MemFuzz: Using Memory Accesses to Guide Fuzzing[C]. *2019 12th IEEE Conference on Software Testing*, *Validation and Verification*, 2019: 48-58.

[23] Donglin Liang, Mary Jean Harrold. Efficient points-to analysis for whole-program analysis[C]. *7th European software engineering Conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, 1999: 199-215.

[24] Gorbovitski M, Liu Y A, Stoller S D, et al. Alias Analysis for Optimization of Dynamic Languages[C]. *The 6th symposium on Dynamic languages - DLS '10*, 2010: 27-42.

[25] Hind M, Burke M, Carini P, et al. Interprocedural Pointer Alias Analysis[J]. *ACM Transactions on Programming Languages and Systems*, 1999, 21(4): 848-894.

[26] Manu Sridharan, Satish Chandra, Julian Dolby, et al. Alias analysis for object-oriented programs[C]. *Computer Science*, 2013: 196-232.

[27] Gutzmann T, Lundberg J, Lowe W. Towards Path-Sensitive Points-to Analysis[C]. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007: 59-68.

[28] Whaley J, Lam M S. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams[C]. *The ACM SIGPLAN 2004 conference on Programming language design and implementation - PLDI '04*, 2004: 131-144.

[29] Berndl M, Lhoták O, Qian F, et al. Points-to Analysis Using BDDS[C]. *The ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003: 103-114.

[30] Sridharan M, Gopan D, Shan L X, et al. Demand-Driven Points-to Analysis for Java[C]. *The 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*, 2005: 59-76.

[31] Pin User Guide. https://software.intel.com/sites/landingpage/ pintool/docs/98425/Pin/html

[32] Wang T T, Su X H, Ma P J. Research on Pointer Analysis Algorithm for Program Standardization[J]. *Acta Electronica Sinica*, 2009, 37(5): 1104-1108.
(王甜甜, 苏小红, 马培军. 程序标准化转换中的指针分析算法研究[J]. *电子学报*, 2009, 37(5): 1104-1108.)

张羿伟 于 2018 年在中国人民大学信息安全专业获得学士学位。现在中国人民大学信息安全专业攻读硕士学位。研究兴趣包括模糊测试、二进制分析。Email: yiweizhang@ruc.edu.cn

游伟 于 2016 年在中国人民大学信息学院计算机应用技术专业获得博士学位。现任中国人民大学信息学院副教授。研究领域为信息安全。研究兴趣包括安全漏洞挖掘、恶意程序分析及移动安全等。Email: youwei@ruc.edu.cn

梁彬 于 2004 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任中国人民大学信息学院教授。研究领域为信息安全。研究兴趣包括: 软件安全性分析、信息安全攻防对抗及系统软件安全机制等。Email: liangb@ruc.edu.cn

万欣宇 于 2018 年在中国人民大学信息安全专业获得学士学位。现在中国人民大学信息安全专业攻读硕士学位。研究领域为软件安全性分析。研究兴趣包括: 二进制安全分析, 模糊测试。Email: wxyxsx@ruc.edu.cn

郭苏越 于中国人民大学信息安全专业攻读学士学位。研究领域为模糊测试。研究兴趣包括漏洞挖掘, 漏洞利用。Email: suyue.guo@ruc.edu.cn