Crash可利用性分析方法研究综述

张 婧 周安民 刘 亮 贾 鹏 刘露平
(四川大学电子信息学院 成都 610065)

摘 要 Fuzzing技术是现阶段用于漏洞挖掘的主流技术,目前绝大多数的软件漏洞都是利用该技术发现的。但是Fuzzing技术存在的一个主要问题是其会产生大量的crash样本,如何对这些crash样本进行快速的分析分类,是当前基于Fuzzing技术进行漏洞挖掘工作所面临的主要问题。针对crash可利用性分析的研究,首先,总结了导致程序crash的原因并对其分析技术发展的现状进行了概述;其次,着重分析了当前利用动态污点分析和符号执行等技术进行crash可利用性判定的4种有效分析方法;最后,对比了这4种方法之间的差异,并探讨了crash可利用性分析技术未来的发展方向及趋势。

关键词 Crash分析,可利用性判定,污点分析,符号执行

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.05.002

Review of Crash Exploitability Analysis Methods

ZHANG Jing ZHOU An-min LIU Liang JIA Peng LIU Lu-ping
(College of Electronic and Information,Sichuan University,Chengdu 610065,China)

Abstract Fuzzing technology is the main technology used in the current stage of vulnerability mining,and currently the majority of software vulnerabilities are discovered by using this technology. However,one of the main problems about Fuzzing technology is that it will produce a large number of crash samples,and how to quickly analyze these crash samples is the main problem of using Fuzzing technology for vulnerability mining work. This paper focused on the researches of crash exploitability. Firstly,it summarized the causes of crash and discussed the development status of its analytical technology,and then it seriously analyzed four effective methods of crash availability judgment by using dynamic taint analysis,symbol execution and other techniques. Finally,it compared the differences between the four methods,and explored the future development direction and trend of the crash exploitability analysis techniques.

Keywords Crash analysis,Exploitable determination,Taint analysis,Symbolic execution

1 引言

随着网络技术的飞速发展,网络安全已经成为人们日益关注的重要问题。近年来信息安全漏洞数量呈现出明显的上升趋势,不仅如此,新漏洞从公布到被利用的时间也越来越短。黑客不仅能在极短的时间内成功利用刚公布的新漏洞,还能挖掘并利用一些尚未公布的漏洞来发起攻击或出售资料,以达到获利目的。相对于黑客,安全研究人员在漏洞研究方面显得被动且滞后[1]。因此,加大对漏洞挖掘和利用技术的研究力度,才能更加主动且有效地保障网络安全。Fuzzing(模糊测试)技术是近几年发展起来的一项漏洞挖掘技术,主要用于挖掘软件漏洞[2]。Fuzzing是一种通过向目标程序提供非预期的输入并监视异常结果来发现漏洞的方法[3],它可以高效地产生大量的crash(崩溃),但并不是所有的crash都可以被攻击者用于发起攻击,相反地,在成千上万的crash中只有极少数是可利用的[4]。无论是攻击者还是防御者,都十分关心这些crash是否可能被利用。为了提高漏洞挖掘的效率,如何快速分析、评估crash的可利用性已经成为当前漏洞挖掘与分析的关键问题之一。

在早期的软件安全测试过程中,判定软件异常的可利用性通常由分析人员借助调试器、Valgrind[5]和AddressSanitizer[6]等工具完成,严重依赖于分析人员的经验和能力,也存在分析精度和深度不足的问题,误判率和漏判率较高[7]。随着crash可利用性分析研究的不断发展,所使用的主流技术从较为简单的字符串匹配、语义分析转变为较为复杂的静态污点分析、动态污点分析以及符号执行等;近几年也不断涌现了一些较为优秀的思想和方法,包括静态分析方法、静态与动态分析相结合的方法以及动态分析方法。静态分析以

到稿日期:2017-03-02 返修日期:2017-06-18

张 婧(1991-),女,硕士生,主要研究方向为二进制安全、漏洞挖掘与利用,E-mail:xinyi_lan1314@163.com;周安民(1963-),男,研究员,主要研究方向为安全防御与管理技术,E-mail:longlian_5555@163.com(通信作者);刘 亮(1982-),男,硕士,讲师,主要研究方向为漏洞挖掘、恶意代码分析;贾 鹏(1988-),男,博士生,主要研究方向为二进制安全、恶意代码检测;刘露平(1988-),男,博士生,主要研究方向为二进制安全、移动安全。

Microsoft发布的!exploitable 工具为代表，该工具使用散列算法分析 Fuzzing 产生的 crash dump（崩溃转储），从而分类 crash[8]。二进制分析框架 BitBlaze 包括静态分析组件和动态分析组件，综合利用动态污点分析和符号执行等技术来快速确定 crash 产生的根本原因[9]。基于漏洞利用自动生成平台 AEG 改进而来的动态分析框架 CRAX，利用符号执行技术可以自动生成漏洞利用样本[10]。此外，程序崩溃分析框架 ExpTracer利用动态污点分析技术跟踪到达崩溃点的痕迹，根据预先定义的崩溃类型进行 crash 可利用性分类[11]。

本文基于 crash 可利用性分析的研究，试图全面介绍、分析和总结当前 crash 可利用性判定发展过程中出现的优秀方法。第 2 节总结 crash 产生的原因；第 3－6 节介绍当前用于 crash 分析的 4 种典型方法；第 7 节对比分析这 4 种方法；最后总结全文并探讨未来的研究方向。

2　Crash 产生的原因

目前国内外常用的漏洞挖掘技术主要有 Fuzzing 技术、补丁比较、静态手工/自动分析和动态调试技术等。Fuzzing 源于软件测试中的黑盒测试技术，它的基本思想是把一组随机数据作为程序的输入，并监视程序运行过程中的所有异常，通过记录导致异常的输入数据来进一步定位程序中的缺陷[3]。经证实，Fuzzing 是收集程序错误信息的有效方法。然而，这些技术大多以实现异常为测试的终止条件，没有或无法对异常的可利用性做进一步分析和判定，即缺少自动评估程序异常威胁严重等级的环节[7]。

在这种情况下，发现和利用（或修复）安全漏洞的限制因素不是找到异常，而是对 crash 进行分组和区分优先级，并确定其根本原因[12]。研究人员无法手动分析数百万次的 crash，因此需要找到不同 crash 之间的联系。例如，由相同错误导致的 crash 或者由不同错误引发的相似 crash 都应被分为一组。另一种方法是优先处理 crash，常见 crash 应该在罕见 crash 之前被修复。分析 crash 产生的根本原因将有助于判定 crash 是否可被利用[9]。一般来说，程序 crash 是由于内存操作不当引起的。具体来讲，主要有以下原因[13]：

1）声明错误，即变量未声明；

2）初始化错误，即变量未初始化或初始化错误；

3）访问错误，具体表现为数组索引访问越界、指针对象访问越界、访问空指针对象、访问无效指针对象以及迭代器访问越界；

4）内存泄漏，即内存未释放或内存局部释放；

5）参数错误，如本地代理、空指针、强制转换等；

6）堆栈溢出，主要体现在递归调用、循环调用、消息循环、大对象参数及大对象变量；

7）转换错误，如有符号类型和无符号类型进行转换；

8）内存碎片，如小内存块的重复分配与释放会导致内存碎片，最后出现内存不足。

检测到程序崩溃时，程序内存可以保存在 crash dump 中[12]。分析 Fuzzing 创建的 crash dump，其目的是对 crash 进行分组和区分优先级。crash dump 包含程序崩溃时的详细状态信息，其固有限制是它只显示 crash 时程序的状态；调用堆栈可以提供重要的历史信息，但它不显示函数内的控制流，也不显示已返回的函数调用。在内存损坏的情况下，调用堆栈可能已经被破坏[12]。动态程序分析的方法可以提供 crash 前程序的执行信息，指出内存损坏发生的位置，有助于查找 crash 的根本原因，可作为 crash dump 分析的补充[14]。动态污点分析技术可以为 crash 的分类和优先级区分提供额外的判定信息；此外，使用执行轨迹的动态分析亦被证明是一种分析 crash 根本原因的有效方法[12]。

3　调用堆栈分析

Crash dump 在程序发生崩溃时自动生成，显示程序的最终状态，可以准确地描述程序如何崩溃，但它只能提示某些特定崩溃发生的原因，有用的提示是崩溃位置和调用堆栈[12]。2009 年，微软安全工程中心（MSEC）基于 Microsoft 在 Windows Vista 的开发过程中使用 Fuzzing 的经验创建了!exploitable 工具，用于自动崩溃分析和安全风险评估[15]。该工具通过对比调用堆栈对应的散列值对 crash 进行分类，进而分析 crash 上下文的语义信息来定义可利用性的级别。它只依赖于 crash dump，而不是生成 crash 的输入。微软的安全人员分析了 Vista 上的 1000 万个 crash，发现许多 crash 都有相似之处。崩溃触发时虽然路径不可达，但是根本原因相同，因此将一个代码区域中出现的 crash 归为一类[16]。

程序发生崩溃时，WinDbg 加载 MSEC. dll 插件，使用 !exploitable命令分析 crash 是否可利用，初步判断可利用性并记录结果。该工具完成两个功能：1）整理所有 crash，创建哈希值来确定崩溃的唯一性，以避免查看重复项并判定实际的崩溃量。微软安全科学小组的内部测试曾利用 4 个不同的 Fuzz 测试程序测试同一软件，!exploitable 从 57 次由 Fuzzing 引起的不同崩溃中识别出 15 处安全问题，其中只有 1 处被分类为可利用[8]。2）查看崩溃类型，并判定崩溃是否可以被恶意利用，然后给 crash 分配可利用性等级：Exploitable（可利用）、Probably Exploitable（可能可利用）、Probably Not Exploitable（可能不可利用）或 Unknown（未知）[17]。

!exploitable 使用散列算法来比较不同的调用堆栈。崩溃发生时，为了分离唯一的 crash，!exploitable 收集崩溃点的堆栈信息，并且创建一个调用堆栈的散列[12]。散列为 16 进制编码，由主散列和次散列组成。默认情况下，主散列基于前 5 个堆栈帧，阈值为 5 的情况下在源代码中可配置。计算主散列时，不考虑函数的偏移量。次散列包括偏移量，并且基于所有可用的堆栈帧。主散列可用于在前 N 个顶部函数名称相同的情况下对调用堆栈进行分组，次散列用于严格标识唯一的堆栈跟踪。所有堆栈帧必须完全相等，才能产生相同的次散列。主散列可被看作是类似崩溃的桶。如果前 N 个堆栈的函数名相同，则产生相同的主散列，这将用于捕获两种崩溃之间的关系：一种类型是发生在不同位置但具有相同功能且导致崩溃的函数调用相同的崩溃；另一种类型是向下调用堆栈时存在不相关的差异，但前 N 个堆栈帧相等[12]。

!exploitable 的判定规则如表 1 所列[12]。异常类型、寄存器值和崩溃发生时基本块的代码分析是判定的重要因素。表中最后一列表示该规则的判定结果是否作为最终的分类结

果,若为假则表示最终结果可能被另一遵循的规则分类结果所覆盖;最后3行是在没有其他适用规则时使用的回退规则。

表1 !exploitable判定规则

Table 1 Decision rule of !exploitable

分类	描述	最终结果
没有例外	当前事件不是例外	真
可利用	堆栈中运行的代码异常	真
可利用	非法指令冲突	真
可利用	特权指令冲突	真
可利用	保护页冲突	真
可利用	堆栈缓冲区溢出(/GS异常)	真
可利用	堆损坏	真
可利用	内核模式数据执行保护冲突	真
可利用	数据执行保护冲突	真
可能可利用	靠近NULL的数据执行保护冲突	真
可利用	用户模式下写AV操作	真
可能可利用	NULL附近用户模式写AV	真
可利用	内核内存中写入AV	真
可利用	内核模式下写入AV	真
可利用	内核模式在指令指针处读取AV	真
可利用	指令指针处读取AV	真
可能可利用	指令指针靠近NULL处读取AV	真
可利用	内核读取控制流上的AV	真
可利用	读取控制流上的AV	真
可能可利用	读取控制流上NULL附近的AV	真
可能可利用	块数据移动中读取AV	真
可能可利用	块数据移动中内核内存读取AV	真
可能可利用	块数据移动中内存读取AV	真
可能可利用	污染数据控制代码流	真
可能可利用	污染数据控制后续写入地址	真
可能可利用	读取NULL附近的AV	假
可能可利用	内核态第一次读取用户内存中的AV	假
可能可利用	内核第一次在用户内存中写入AV	假
可能可利用	整数除以零	假
可能可利用	浮点除以零	假
未知	断点	假
未知	错误检查	假
未知	可能的堆栈损坏	假
未知	NULL附近的内核读访问冲突	假
未知	污染数据被用于后续的块数据移动操作	假
未知	块数据移动中的内存读访问冲突	假
未知	污染数据被用作函数调用中的参数	假
未知	污染数据可能被用作返回值	假
未知	污染数据控制分支选择	假
未知	读访问冲突	真
未知	写访问冲突	真
未知	数据执行保护冲突	真

评估崩溃严重程度的一个重要因素是崩溃能否由用户输入控制。!exploitable在最后一个基本块中执行静态污点分析,该分析的输出可用于判断损坏的数据是否被用作函数的输入参数或者崩溃函数的返回值[12]。其用于描述个别崩溃,分析结果是保守的,因为它假定所有数据都被污染。

虽然!exploitable是为了分类由Fuzzing产生的crash而创建的,但它可被用于分析任何崩溃[12]。!exploitable工具的易用性强,同时因为采用的是静态分析方法,所以速度较快,可以极大地帮助研究人员节省分析crash的时间和精力。但是该工具具有较高的假阳性率[8],并且只能给出Windows平台下准确的崩溃判断,对于其他第三方软件和一些更复杂的崩溃,例如堆溢出和UAF(Use-After-Free)等,准确率非常低[11]。因此该工具适用于对数量庞大的crash进行较为模糊的初步筛选和分析,而针对某个特定的crash,仅静态分析局部代码是不够的,还需要对程序的后续控制流和数据流做精细化分析。

4 执行跟踪

BitBlaze是UC Berkely开发的基于二进制信息的分析平台,提供了一种全新的通过二进制代码分析解决计算机安全问题的方法[18],主要用于确定在崩溃时哪些寄存器和内存地址来自攻击者控制的输入文件,以及对NULL指针进行切片并查看起始位置来快速排除可利用性。这个平台支持精确的分析,提供一种可扩展的架构,并且结合了静态分析技术和动态分析技术以及程序验证技术来满足普通需求;可以帮助快速确定由基于文件变异的Fuzzing生成的特定崩溃是否可利用,并且有助于确定crash产生的根本原因[9]。

使用中间语言是理解程序指令最为经典的方法,它可以提高代码质量和程序的可移植性[19]。BitBlaze平台主要结合了二进制程序的信息流分析、污点分析以及符号执行技术,主要由3部分组成:Vine,TEMU及Rudder[20]。Vine是它的静态分析组件,由前端、后端和中间语言(IL)组成(见图1),其提供一系列在IL上进行静态分析的核心工具(包括STP工具、控制流及数据流等)。TEMU是动态分析组件[21](见图2),是在虚拟机QEMU的基础上进行组件扩展的二进制动态分析平台。被测试程序运行于虚拟机的GuestOS中,同时整个系统对被测试程序做一次细粒度的污点分析,即将污点分析引擎集成到虚拟机中[22]。Rudder利用Vine和TEMU提供的功能实现二进制程序的混合符号执行,通过诸如决策过程的解算装置,可以判定什么输入能使程序按给定的路径执行。

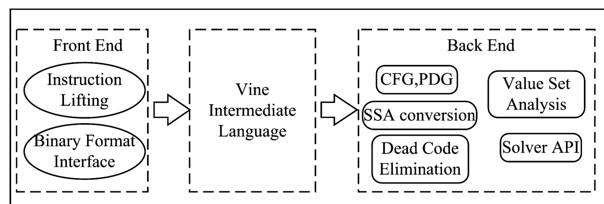

图1 Vine结构

Fig.1 Vine structure

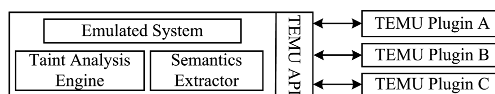

图2 TEMU结构

Fig.2 TEMU structure

BitBlaze为研究人员提供了各种功能,例如可以对导致crash的执行采取污染增强的跟踪,使用BitBlaze提供的工具进行离线数据流的分析[9]。对于正向跟踪的数据,它能够可视化执行跟踪中的污点信息;对于反向跟踪的数据,它可以对数据进行切片[23],以查看来源。相比之下,!exploitable工具只切片当前基本块,并假设所有数据都被污染。此外,BitBlaze可以提供带有污点信息的崩溃报告,该报告将有助于研究人员判定crash的优先级并确定可利用性[9]。

BitBlaze提供了各种工具,用于减少分析crash产生原因的时间,包括分析数据和执行的工具。其假设一个好文件不

会导致crash,而一个坏文件会引发crash[9]。好文件和坏文件只有几个字节的差异,大多数情况下只是单字节的差别。使用BitBlaze来跟踪坏文件(x.trace)和好文件(y.trace)的执行。通过污染两者不同的字节来观察这些字节如何通过程序传播,然后使用差别污染技术进行最后的跟踪(z.trace),此时再次污染两者不同的字节但删除其中运行时相同的污点,以消除不必要的过度污染。大多数实际情况下,只需要查看z.trace和y.trace。

以Adobe Reader进行基于文件变异的Fuzzing时发现的crash为例[24](见图3),该崩溃被!exploitable工具归为未知,通过BitBlaze进一步分析crash产生的原因表明其不可利用[9]。由分析可知,该crash可能是由空指针异常导致并且不可利用,但是0x14可能来自输入数据,因此!exploitable不能将其判定为无法利用。进一步分析发现ecx没有被污染,更加表明不可利用。此外,切片ecx查看0x14的来源,得知没有内存读取,其是通过mov和lea指令直接来源于xor,这也说明了该crash不可利用,因为这是一个空指针的引用。

```
$ trace_reader-trace z. trace-header | grep Number
Number of instructions:272998
$ trace_reader-intel-v-trace z. trace-first 272998
208063fb:mov edx, DWORD PTR [ecx+0x4] M@0x00000014[0x00000000][4](R)
T0 R@edx[0x00000000][4](W) T0 ESP:NUM_OP:4 TID:1712 TP:TP-None
EFLAGS:0x00000083 CC _ OP:0x00000008 DF:0x00000001 RAW:0x8b5104 MEMREGS:R@ecx[0x00000010][4](R)T0
```

图3　Adobe Reader的crash

Fig. 3　Crash of Adobe Reader

使用BitBlaze分析根本原因,首先比较好文件和坏文件的崩溃位置,如图4所示。好文件中切片ecx的合法值除了在切片末端用一个指针填充esi的值而不是仅仅为0外,与坏文件中ecx的切片相同。esi被填充在指令0x208f35f7中,也恰好在指令00266161处,验证可知该值设置在跟踪结束附近的非对齐区域内,因此ecx的初始值为0。在好文件中继续执行下一个分支,将esi设为有效指针;而在坏文件中,esi的值不会被设置,这将导致空指针引用。因此,执行中的差异来自于ax中0b11的值而不是0b10,这足以修复错误,但是ax的值可以被切片并且继续以这种方式导致crash[9]。

```
$ trace_reader-intel-trace z. trace-first 272998-last 272998
208063fb:mov edx, DWORD PTR [ecx+0x4] M@0x00000014[0x00000000][4](R)
T0 R@edx[0x00000000][4](W) T0
$ aligned. pl y_z. aligned. txt T1:272998
〈T0:271645〉 ~ T1:272998
(T0:00271451-00271645 ~ T1:00272804-00272998)
$ trace_reader-intel-trace y. trace-first 271645-last 271645
208063fb:mov edx,DWORD PTR [ecx+0x4] M@0x0202c5fc[0x025d91f8][4](R)
T0 R@edx[0x00000000][4](W) T0
```

图4　崩溃位置的比较

Fig. 4　Comparison of crash location

对于crash产生的根本原因的分析,目前主要通过手动或使用调试器完成。使用BitBlaze可以简化和加快进程,并提供可重复性。BitBlaze运作在GuestOS的更底层,对目标程序有较好的透明性,但该平台局部开源,不便于扩展,污点的设置同样缺乏灵活性[25]。此外,BitBlaze的效率低下,而且它是一种离线Fuzz的思路,首先收集条件约束,然后在程序运行轨迹文件上做程序分析,可操作性欠佳。

5　端到端

在2008年的IEEES&P会议上,Brumley等人首次提出了基于二进制补丁比较的漏洞利用自动生成方法APEG[26]。其基于以下假设条件,即补丁程序中增加了对触发原程序崩溃的过滤条件。因此只要找到补丁程序中添加过滤条件的位置,同时构造不满足过滤条件的“违规”输入,即可认为该“违规输入”是原程序的一个可利用的输入候选项。但它存在两方面的局限:1)无法处理补丁程序中不添加过滤判断的情况;2)所构造的利用类型主要属于拒绝服务,即只能造成原程序崩溃,无法造成直接的控制流劫持。

为了弥补APEG对补丁的依赖以及无法构造控制流劫持的缺陷,Avgerinos等人在2011年的NDSS会议上首次提出了一种有效的漏洞自动挖掘和利用方法AEG[27]。它借助程序验证技术找出使得程序进入非安全状态且可被利用的输入。非安全状态包括内存越界写、恶意的格式化字符串等;可被利用主要是指程序的EIP被任意操纵。其局限性主要体现在:1)依赖于源代码搜索程序错误;2)所构造的利用样本主要面向堆栈缓冲区溢出或者格式化字符串漏洞;3)利用样本受限于编译器和动态运行环境等因素。

黄世昆等人在2012年的IEEE会议上提出了基于AEG方法的改进自动化利用生成框架CRAX[28]。它是一个基于S^2E[29]的符号执行平台。为了生成控制流劫持攻击,需要检测符号EIP以及其他基于连续的寄存器和指针,为此提出了一种搜索最大连续符号内存用于有效载荷注入的系统方法。检测符号寄存器是一种处理各种控制流劫持漏洞的全面且简单的方法[30]。

该框架实现了concolic模式模拟来对符号执行进行concolic测试[31],使得在符号执行和concolic测试之间进行切换时,不需要修改内存模型;此外,代码选择过滤了一些复杂和不相关的库函数,减少了SMT解算器[32]的开销,加快了利用生成的速度;同时,通过识别影响连续性的重要输入(称为热字节),使用选择性符号输入来减小符号变量,通过仅标记这些字节可以明显减少整个利用程序的生成时间。该框架使用端到端的方法为各种应用程序生成漏洞利用,包括几个中等规模的基准程序和几个大规模的应用程序,例如Mplayer(媒体播放器)、Unrar(归档器)和Foxit pdf(阅读器)[10]。

CRAX的具体实现方法包括以下步骤[33]。

1)检测符号程序计数器——x86机器中的EIP寄存器。EIP寄存器包含要执行的下一条指令的地址,因此控制寄存器是所有控制流劫持攻击的常见目标。当符号执行探索路径和污点内存时,用符号数据更新EIP寄存器将触发漏洞利用。漏洞利用生成将搜索内存以找到可用的内存区域来注入shellcode和NOP sled,并将EIP寄存器重定向到shellcode。

检测符号EIP寄存器和漏洞利用的生成过程如图5所示。

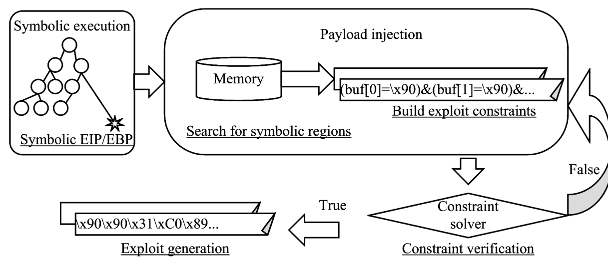

图5 漏洞利用的生成过程

Fig.5 Generation process of vulnerability exploitability

2)shellcode注入。为了注入shellcode,首先需要找到所有符号化并且足以容纳有效载荷的内存块。即使符号块由许多不同的变量组成,只要块是连续的,它也仍然可以用于注入shellcode。然而,手动分析源代码很难找到由用户输入污染并与变量组合的连续内存区域。此外,编译器通常会改变变量的顺序或分配大小进行优化,因此很难手动找到shellcode缓冲区。通过系统搜索最大连续符号内存可以自动化该过程。

3)NOP sled和漏洞利用生成。确定shellcode的位置时,NOP sled将尝试在shellcode前面插入尽可能多的NOP指令序列。这种填充有助于利用不同系统中shellcode的不准确位置,或者扩展shellocde的入口点。最后,由符号数据损坏的EIP寄存器将指向NOP填充的中间。所有漏洞利用约束(包括shellcode,NOP sled和EIP寄存器约束)都被传递给具有路径条件的SMT解算器,以确定漏洞利用是否可行。如果不可行,则漏洞利用生成返回shellocde注入的步骤来改变shellcode的位置,直到生成漏洞利用或者没有更多的可用符号缓冲区。

漏洞利用生成的步骤如下:1)收集必要的运行时信息;2)构建漏洞约束;3)利用约束将漏洞传递给约束解决器,实现漏洞利用。S^2E中的内存模型也是实现所提方法的关键因素[34]。除了返回到内存的漏洞利用外,还实现了两种其他类型的漏洞利用:返回libc和跳转到寄存器。通过绕过一些保护,生成的漏洞利用可以在实际系统中发挥作用[33]。

CRAX的优势主要体现在两个方面:1)解决了没有源代码的大型软件系统的漏洞利用自动生成;2)确定崩溃优先级。目前,只有从错误分析器和随机模糊器产生的崩溃报告可用。该框架可以粗粒度地确定错误修复的优先级[10]。

然而CRAX比AEG更耗时,因为它进行整个系统的符号执行,而AEG只是在应用程序级别执行。通过减少在concolic执行期间的约束数量可以进行时间优化,优化之后整个系统的执行性能接近AEG,甚至可以达到AEG速度的50倍[33]。CRAX还可以扩展到更大的程序,包括Open-office,Microsoft Office和Web浏览器。驱动模式应用程序,如Windows防病毒软件和内核模块(如虚拟机管理程序)等,也是未来的目标。

6 基于静态优化的细粒度污点分析

在2014年IEEE会议上,张普含等人提出了程序崩溃分析框架ExpTracer[11]。该框架使用基于污点分析的数据流引导分析技术[35]直接分析二进制程序,判定崩溃点是否可以被攻击者控制,实现了崩溃威胁的分类。其核心思想是首先识别崩溃的类型,然后对程序入口点到崩溃点之间的跟踪进行污点分析,并记录内存集和寄存器的污点信息,最后整合上述记录以及后续指令分析,得出判定结果。该框架使用二进制插桩平台PIN[36]作为原型系统,工作流程如图6所示[11]。

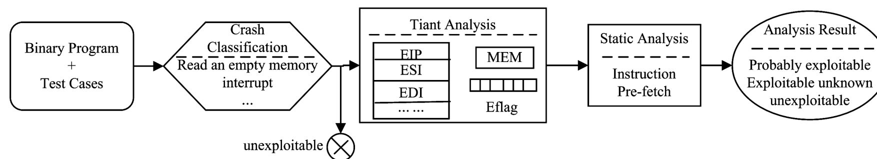

图6 ExpTracer的工作流程

Fig.6 Workflow of ExpTracer

该框架基于DBI(动态二进制工具)平台[37]的静态优化提出了一种细粒度污点分析方法,该方法不仅可以确定内存单元/寄存器是否被污染,而且能够识别污染目标内存或寄存器的污染源的偏移量,以便于识别用户输入和软件崩溃点之间的关系。通过静态分析提取污染传播的语义信息,删除与污染传播无关的指令,并合并重复顺序的扩展。根据静态分析的反馈,动态分析将完成具体的污点分析[38]。

为了减小进一步确定的范围并提高系统执行的效率,首先删除那些已知不可利用的crash,然后根据系统崩溃期间生成的崩溃消息和导致崩溃的指令类型将crash分为以下4类[11]。1)NULL指针引用:①由于软件发布版本是初始化指针程序的错误版本,因此经常报告"无法读取数据,内存地址为空",但是这个问题不会出现在调试版本;②在内部程序中逻辑处理不当,导致零指针[39]和读取错误。但是这两个原因都不能改变程序控制流,因此判定"读空内存"不可利用。2)直接跳转指令:对于执行跳转指令(例如jmp,call)时出现crash的情况,首先判断目的地址是否被污染[40]。如果被污染,进一步确定EIP寄存器是否被污染。只有目的地址和EIP寄存器都被污染时才能确定崩溃可利用。如果跳转指令没有被污染,则系统将分析后续指令是否包含可以改变控制流的指令。3)内存/寄存器修改指令:当崩溃指令是内存/寄存器修改指令(例如mov)时,需要确定源操作数是否被污染以及被污染的字节数,同时跟踪污染内存/寄存器的使用位置,以及是否发生信息泄漏。如果发生泄漏,崩溃可利用。然后,静态分析后续指令,并检查被污染的内存/寄存器是否影响后续跳转指令。如果存在影响,则控制流可能被劫持并且崩溃可利用。4)中断指令:为了防止控制流被劫持,编译器通常在编译时插入一些中断指令(0xCC),因此进程崩溃时崩溃点经常出现"CC"命令,这种类型的crash难以实现自动化判断,而且这些crash通常难以利用,因为改变控制流相对困难。这类crash将被视为"可能可利用"。

可利用性判定最终取决于是否可以控制EIP寄存器。

确定EIP寄存器是否被污染的最直接方法是污点分析技术，另一种方式是通过模式匹配提取崩溃模式[41]。ExpTracer先分类崩溃，然后根据不同类型的crash通过污点分析识别污染内存/寄存器，最后判定crash是否可利用。在此框架下，分别提取不同崩溃的模式，并通过模式匹配来执行更准确的崩溃判定，可实现对该框架的扩展。崩溃分析的具体流程[11]如下。

1)类型识别。对于可触发崩溃点的二进制程序和样本，通过模拟执行再触发崩溃来收集崩溃点的信息，然后根据崩溃触发时系统提示的错误和命令信息来确定崩溃的类型。例如，针对“读空内存”崩溃，控制流不能被直接利用，因此判定为“不可利用”；对于“中断”类型的崩溃，程序本身有控制流保护机制，给出的控制流更加难以被劫持，因此认为“可能可利用”。

2)污点分析。污点分析模块包括静态模块和动态模块。静态模块将指令转化为一种中间表示，完成非污染指令管理和重复污染传播的优化，并将优化结果返回给动态污点分析模块。动态污点分析根据样本提取污染源，完成指令存根以及污染传播记录，建立和分析实时更新的污染记录表，最后将分析结果提交给崩溃判断模块[42]。

3)指令预取。通常崩溃点的指令不能改变程序控制流，因此很难实现漏洞利用。可利用崩溃的利用点可能在崩溃点之前，也可能在崩溃点之后。前者需要通过回溯法进行具体分析，后者则需要分析崩溃点之后的指令[43]。这里只考虑后一种情况，并以粗粒度的方式读取崩溃点之后的指令，从中提取改变程序控制流的指令，如call，jmp等。通过静态分析这类指令查看污染情况，如果目的地址被污染，则认为可能可利用：一方面因为不确定路径是否会到达这一点，另一方面是不确定EIP指令是否被污染。

4)可利用性判断。可利用性判断模块根据崩溃点的指令类型使用不同的判定方法。对于jmp指令，主要基于污点分析的结果，查看jmp指令的目的地址是否被污染，可以被污染源的哪些字节污染。如果可以被污染并且EIP寄存器可以被劫持，那么程序的控制流就可以被改变，以实现漏洞利用。对于内存/寄存器修改指令，首先判断源操作数是否可以被污染，如果可以，则进一步跟踪目的操作数的污染情况。如果检测到可以改变程序控制流的指令的污染传播，就表明崩溃可利用。

从微软安全中心、CVE漏洞库和国家漏洞数据库选择公开漏洞进行验证[11]。ExpTracer框架对崩溃可利用性的判定结果与!exploitable工具的判定结果的比较如表2所列。

ExpTracer在识别改变控制流的crash可利用性方面优于!exploitable工具[11]。但是对于UAF漏洞，由于POC有许多内存重写指令会导致更加复杂的逻辑结果，因此ExpTracer不能准确判定而只能给出“可能可利用”的分析结果；ExpTracer的时间开销明显高于!exploitable，然而离线分析与手动分析相比，耗时是可以接受的；而且ExpTracer是一种粗粒度的异常判断方法，特别是对一些逻辑关系相对复杂的异常，并不能给出准确的判断结果，而需要借助反汇编、指令识别方法[44]等。

表2　崩溃可利用性的判定结果

Table 2　Judgment results of crash availability

编号	崩溃指令	污染源	!exploitable	ExpTracer
MS06-040	call ds:_imp_wcscat	128	可利用	可能可利用
MS08-067	mov ecx, dword ptr ss:[ebp+8]	128	可利用	可利用
CVE-2011-2130	movzx eax, word ptr [eax+1ch]	64	未知	可利用
CVE-2011-2595	mov ecx, dword ptr ds:[eax]	128	未知	可利用
CVE-2013-2551	mov ecx, dword ptr [eax+14h]	256	可能可利用	可能可利用
CVE-2013-0753	movzx eax, word ptr [ecx+4ah]	128	未知	可利用
CVE-2011-0609	call dword ptr[ebp+68h]	64	未知	可利用
CNNVD-201310-129	mov ecx, dword ptr ss:[ebp+34h]	256	可能可利用	未知
CNNVD-201309-301	movzx eax, word ptr [ecx+34h]	128	未知	未知

7　方法的比较

本文第3－6节对现阶段用于分析crash的4种代表性方法分别进行了归纳与总结，表3对这4种方法进行了对比分析。

表3　4种方法的比较

Table 3　Comparision of four methods

方法	调用堆栈分析	执行跟踪	端到端	基于静态优化的细粒度污点分析
代表工具	!exploitable	BitBlaze	CRAX	ExpTracer
创建时间	2009	2008	2012	2014
主要方法	散列算法	信息流分析/污点分析/符号执行	符号执行	污点分析/数据流分析
对象	崩溃转储	二进制程序	应用程序	二进制程序
动态/静态	静态	动静结合	动态	动态
针对漏洞	栈溢出	缓冲区溢出	栈/堆溢出、初始化错误	缓冲区溢出
是否开源	开源	局部开源	未开源	未开源
能否扩展	可扩展	可扩展	可扩展	可扩展
平台	windows	linux	linux/windows/web	windows
主要优点	适用崩溃数量巨大/易用性强	加快崩溃原因分析	无源代码的软件系统漏洞利用生成	较准确分类改变程序控制流的崩溃
主要缺点	假阳性率较高/堆溢出等分类不准确	效率低下、无法分析无效读取崩溃	耗时较长	无法准确判断逻辑关系复杂的异常

综合上述分析，可知以上4种方法都有各自的优点和局限性，且有不同的适用场合。CRAX和ExpTracer两种分析方法尚未开源，应用覆盖范围较窄，但其优秀的思想对于程序crash可利用性判定研究具有重要的参考价值。目前主要采用!exploitable工具静态分析crash dump的方法初步判定crash的可利用性，并结合BitBlaze平台执行跟踪查找crash的根本原因以进一步确定是否可利用。

结束语　Fuzzing技术作为漏洞挖掘的有效手段，近年来不断受到广泛关注，然而Fuzzing产生的大量crash也令研究人员十分苦恼，尤其针对crash的可利用判定，往往需要投入大量的人力和时间[45]。Fuzzing技术发展过程中涌现了许多

优秀的方法用于判断 crash 的可利用性判断。本文就目前 crash 分析技术的研究现状,总结了 4 种具有代表性的判定方法,并且分别展开了研究和讨论。由分析可知,如何实现 crash 可利用性分析的自动化、高效性及准确性,将是未来漏洞挖掘与利用方向的研究重点。

参 考 文 献

[1] LAI Y P,HSIA P L. Using the vulnerability information of computer systems to improve the network security [J]. Computer Communications,2007,30(9):2032-2047.

[2] TAKANEN A,DEMOTT J,MILLER C. Fuzzing for software security testing and quality assurance[M]. Artech House,2008.

[3] ZHANG X,LI Z J. Survey of Fuzz Testing Technology [J]. Computer Science,2016,43(5):1-8.(in Chinese)
张雄,李舟军. 模糊测试技术研究综述[J]. 计算机科学,2016,43(5):1-8.

[4] LIU Y,XIE J J,ZHANG C R,et al. Crash analysis for off-by-one stack based buffer overflow [J]. Computer Engineering & Design,2015,36(12):3172-3182.(in Chinese)
刘渊,谢家俊,张春瑞,等. 单字节栈溢出的分析[J]. 计算机工程与设计,2015,36(12):3178-3182.

[5] NETHERCOTE N,SEWARD J. Valgrind :A Program Supervision Framework [J]. Electronic Notes in Theoretical Computer Science,2003,89(2):44-66.

[6] SEREBRYANY K,BRUENING D,POTAPENKO A,et al. Address Sanitizer:a fast address sanity checker[C]// Usenix Conference on Technical Conference. Berkeley: USENIX Association,2012:28.

[7] PENG J S,WANG Q X,OUYANG Y J. Exploitable Inference Based on space-time analysis of pointers [J]. Application Research of Computers,2016,33(5):1504-1508.(in Chinese)
彭建山,王清贤,欧阳永基. 基于指针时空分析的软件异常可利用性判定[J]. 计算机应用研究,2016,33(5):1504-1508.

[8] MICROFOST. The History of the !exploitable Crash Analyzer [EB/OL]. http://blogs. technet. com/b/srd/archive/2009/04/08/the-history-of-the-exploitable-crash-analyzer/.

[9] MILLER C,CABALLERO J,BERKELEY U,et al. Crash analysis with BitBlaze [J]. Revista Mexicana De Sociología,2010,44(1):81-117.

[10] HUANG S K,HUANG M H,HUANG P Y,et al. CRAX:Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations [C]// IEEE Sixth International Conference on Software Security and Reliability. New York:IEEE,2012:78-87.

[11] ZHANG P,WU J,XIN W,et al. Program Crash Analysis Based on Taint Analysis[C]// International Conference on P2P. New York:IEEE,2015:492-498.

[12] KROHNHANSEN H. Program crash analysis: evaluation and application of current methods [D]. Norway:University of Oslo,2012.

[13] WU S Z. Review and Outlook of information security vulnerability analysis [J]. Journal of Tsinghua University (Science and Technology),2009(S2):2065-2072.(in Chinese)
吴世忠. 信息安全漏洞分析回顾与展望[J]. 清华大学学报(自然科学版),2009(S2):2065-2072.

[14] LASK J,STANLEY M. Dynamic Program Analysis[M]// Software Verification and Analysis. London:Springer. 2009:368.

[15] NOH M S,NA J B,JUNG G U,et al. A Study on MS Crash Analyzer [J]. Kips Transactions on Computer & Communication Systems,2013,2(9):399-404.

[16] LI L,JUST J E,SEKAR R. Online Signature Generation for Windows Systems[C]// Computer Security Applications Conference. New York:IEEE Computer Society,2009:289-298.

[17] Microsoft. !exploitable Crash Analyzer. MSEC Debugger Extensions[OL]. http://msecdbg. codeplex. com.

[18] SONG D. WebBlaze:New Techniques and Tools for Web Security & BitBlaze:Computer Security via Binary Analysis [OL]. http://bitblaze. cs. berkeley. edu/dragonstar/lec4. pdf.

[19] CHEN K M,LIU Z T,REN C S. Design and Implement of User-Oriented Intermediate Language in Decompilation System [J]. Mini-Micro System,2002,23(10):1173-1176.(in Chinese)
陈凯明,刘宗田,任传胜. 逆编译中面向用户的中间语言设计和实现[J]. 小型微型计算机系统,2002,23(10):1173-1176.

[20] SONG D,BRUMLEY D,YIN H,et al. BitBlaze: A New Approach to Computer Security via Binary Analysis [C]// Information Systems Security,International Conference(Iciss 2008). New Zealand:DBLP,2008:1-25.

[21] NEWSOME J,SONG D. Dynamic taint analysis for automatic detection,analysis,and signature generation of exploits on commodity software [J]. Chinese Journal of Engineering Mathematics,2005,29(5):720-724.

[22] WANG X C. Branch Obfuscation with Machine Learning and One-way Prefix-preserving Encryption Algorithm [D]. Tianjin: Nankai University,2015.(in Chinese)
王晓初. 结合机器学习与单向保留前缀加密算法的分支混淆方法[D]. 天津:南开大学,2015.

[23] JACKSON D,ROLLINS E J. Chopping:A Generalization of Slicing [OL]. http://www. dtic. mil/dtic/tr/fulltext/U2/a282683. pdf.

[24] HAN X,WEN Q,ZHANG Z. A mutation-based fuzz testing approach for network protocol vulnerability detection [C]// International Conference on Computer Science and Network Technology. New York:IEEE,2013:1018-1022.

[25] YE Y H,WU D Y,CHEN Y. Reverse platform based on finegrainted taint analysis [J]. Computer Engineering and Applications,2012,48(28):90-96.(in Chinese)
叶永宏,武东英,陈扬. 一种基于细粒度污点分析的逆向平台[J]. 计算机工程与应用,2012,48(28):90-96.

[26] BRUMLEY D,POOSANKAM P,SONG D,et al. Automatic Patch-Based Exploit Generation is Possible:Techniques and Implications [C]// IEEE Symposium on Security and Privacy,2008(SP 2008). New York:IEEE,2008:143-157.

[27] AVGERINOS T,SANG K C,HAO B L T,et al. AEG: Automatic Exploit Generation [J]. Internet Society,2011,57(2).

[28] HUANG S K,LU H L,LEONG W M,et al. CRAXweb: Automatic Web Application Testing and Attack Generation[C]// IEEE,International Conference on Software Security and Reliability. New York:IEEE Computer Society,2013:208-217.

(下转第 23 页)