基于SEH的漏洞自动检测与测试用例生成

黄 钊 黄曙光 邓兆琨 黄 晖
(国防科技大学 合肥 230037)

摘 要 SEH即结构化异常处理,是Windows操作系统提供给程序设计者处理程序错误或异常的途径。然而SEH的链式处理方式使得程序中可能存在相应漏洞。针对该问题,为提升程序安全性,提出一种基于SEH的漏洞自动测试用例生成方法。首先判断程序是否存在基于SEH被攻击的漏洞风险性,若存在则构建和调整测试用例约束,并自动求解生成相应测试用例。该方法一方面扩展了当前的自动测试用例生成模式,另一方面可在GS保护开启时仍能生成有效测试用例。最后通过实验验证了该方法的有效性。

关键词 结构化异常处理,符号执行,自动测试用例生成
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.07.021

Automatic Vulnerability Detection and Test Cases Generation Method for Vulnerabilities Caused by SEH

HUANG Zhao HUANG Shu-guang DENG Zhao-kun HUANG Hui
(National University of Defense Technology, Hefei 230037, China)

Abstract Structured Exception Handling (SEH), which offered by Windows operating system, is a way to handle program errors or exceptions. However, while SEH handles exception based on link, there may be corresponding vulnerabilities. To solve this problem, in order to improve program security, a method was proposed to generate test cases base on SEH. First, the method judge whether the program has the risk of being attacked based on the SEH. If there is a risk, the test case constraints are constructed and adjusted. Then by solve these constraints, the corresponding test cases are generated automatically. On the one hand, this method extends the current automatic test case generation pattern. And on the other hand, it can generate effective test cases even when GS protection is turned on. Finally, the effectiveness of the method is verified by experiments.

Keywords Structured exception handling, Symbolic execution, Automatic test cases generation

1 引言

漏洞是在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷,可以使攻击者能够在未授权的情况下访问或破坏系统[1]。软件漏洞检测与测试用例生成是当前的研究热点,但漏洞检测技术所发掘的程序错误并不一定能造成实质性危害,往往具有较高的误报率和漏报率。因此针对漏洞特性,产生准确度高的稳定测试用例十分重要[2]。

SEH (Structured Exception Handling,结构化异常处理)是Windows操作系统下的异常处理机制,SEH为程序设计者提供了程序错误或异常的处理途径,使得系统容错性得到提升[3]。但由于SEH的链式处理方式,导致使用SEH的过程中出现了被攻击者能够利用的漏洞[4]。相较于直接进行IP寄存器劫持等传统攻击方式,基于SEH进行漏洞攻击一方面可获得更多有效栈空间,另一方面可轻松绕过GS保护达到攻击效果,因此它也成为攻击者的一种经典攻击手段。

手动漏洞分析方法存在人工经验依赖度强且难以满足软件规模需求等问题,尤其是随着软件规模的不断提升,该过程往往需要配合自动化漏洞分析技术进行。而以模糊测试为代表的传统漏洞自动检测工具的测试用例生成较为盲目,精确度不高。随着程序分析技术的不断发展,尤其是污点分析、符号执行等技术不断成熟,当前已有多个基于污点分析和符号执行进行漏洞检测和测试用例生成的研究及成果[5],如Avgerinos等提出的AEG[6]、Shih-Kun Huang提出的CRAX[7]、Shellphish提出的angr[8]等。但其仍然存在一些问题,如AEG结合了二进制分析和静态分析来生成一个控制流劫持测试用例,但其需要目标程序源码支撑,而实际中大多情况下难以获取目标程序源代码;CRAX能够对一般性缓冲区溢出

到稿日期:2018-06-13 返修日期:2018-09-12 本文受国家重点研发计划"网络空间安全"重点专项(2017YFB0802905)资助。
黄 钊(1994-),女,硕士生,主要研究方向为软件漏洞分析;黄曙光(1960-),男,教授,博士生导师,主要研究方向为信息安全,E-mail:hz0_mu@163.com(通信作者);邓兆琨(1993-),男,硕士生,主要研究方向为网络态势、漏洞分析;黄 晖(1987-),男,博士,主要研究方向为漏洞分析,E-mail:hhui_123@163.com。

漏洞进行自动测试用例生成，但其未能考虑 Windows 提供的各项安全保护机制且测试用例生成模式较为单一，若所应用漏洞非该既定模式则无法生成测试用例。

因此，针对现阶段漏洞检测及测试用例研究中存在的上述缺陷，为提高程序安全性，本文以自动检测漏洞并精确生成相应测试用例为目标，针对二进制程序，设计了一个基于 SEH 的漏洞自动检测和测试用例生成方法，并基于 S2E[9] 符号执行平台进行了相应的系统实现。该方法在程序崩溃时检测其被基于 SEH 异常处理机制进行漏洞攻击的风险性，并自动生成相应测试用例，该测试用例可绕过 GS 保护进行执行。

2 相关研究进展

2.1 符号执行和约束求解技术

根据手动漏洞检测方式及传统自动漏洞检测方式的各自缺陷，本文选择基于符号执行和约束求解技术的、面向二进制程序实现的基于 SEH 的漏洞自动检测和测试用例生成。

符号执行将程序中一些需要关注又不能直接确定取值的变量用符号表示其取值，该符号值可以表示程序在此接收的所有可能输入，然后逐步分析程序可能的执行流程，将程序中变量的取值表示为符号和常量的计算表达式[10]。在符号执行过程中维护符号状态和符号路径约束，在遇到分支指令时，程序的执行也相应地分叉以探索每个分支[11]。分支条件则被加入到当前路径的路径条件中形成路径约束。接着，利用约束求解器进行约束求解以判断路径的可满足性，即找出能满足当前各种条件约束的一个可行解。如果判定结果为可满足，则说明路径可行；如果判定结果为不可满足，则说明路径不可行。利用符号执行一方面可以获取到达漏洞触发点的路径约束，另一方面可构造形成相应测试用例约束，进而求解自动构造测试用例等。

现存的大量工作展示了符号执行和约束求解技术能有效地处理众多软件工程难题，目前典型的符号执行工具有 KLEE[12]，S2E[9] 等，常用的约束求解器有 CVC4[13]，STP[14]，Z3[15] 等。

2.2 符号执行和约束求解技术

2.2.1 S2E

2009 年，EPFL 的 George Candea 团队基于 KLEE 开发了 S2E，其开创了选择符号执行的方式。S2E 是一个用于分析软件系统行为和属性的平台，具体来说，其是一个带有符号执行和模块分析的虚拟机，可运行未经修改的 x86、x86-64 或 ARM 软件栈，包括程序、库、内核和驱动程序，能尽可能地模拟真实环境运行目标程序，然后符号执行将自动在系统中探索各种可达路径[9]。同时，其采用选择性符号执行方式，对研究者感兴趣部分进行符号执行，而对执行库或系统内核等非程序分析关键部分保持具体执行，进而既保证了符号执行效果，又减少了对目标程序进行符号执行的开销。基于 S2E 平台可进行二进制程序漏洞检测、测试用例生成等工作。

2.2.2 CRAX

2012 年，Huang 基于 S2E，KLEE 和 QEMU 提出了一个自动生成控制流劫持测试用例的框架 CRAX，其监视 ip 寄存器，通过检测 ip 寄存器是否会被覆盖为符号值来判定漏洞可利用性；并提出了一种系统的方法来搜索最大的连续符号内存，以自动化地确定一个可供放置 shellcode 的缓冲区，同时通过二分法确定一个可行 NOP 区域，再使得 ip 值指向其中。将以上 3 个因素合取形成约束并与到达漏洞点的路径约束合取后进行求解，即可得到相应测试用例。

CRAX 的工作思路能够解决一般性的二进制程序缓冲区溢出漏洞自动测试用例生成问题，但其未能考虑 GS，ASLR，DEP 等保护机制，所生成的测试用例的使用场景较为局限，且只有上述一种测试用例生成模式。

3 SEH 异常处理机制及 GS 安全编译选项

3.1 GS 安全编译选项

GS 是微软针对缓冲区溢出时可能发生的覆盖函数返回地址问题提出的编译选项[16]。其在函数被调用时向栈帧中压入一个额外的 DWORD 随机数，同时在. data 内存区域中存放一个该随机数的副本，该随机数又称为 canary。由于 canary 的存储位置在 BP 寄存器之前，使得栈中发生溢出时将首先覆盖该 canary，其次才覆盖 BP 寄存器和返回地址。函数返回前，系统将对比栈帧中的 canary 及. data 中的 canary 副本，若两者不吻合，则认为发生溢出，系统进入溢出处理流程而不会继续返回到返回地址处继续执行流程。因此程序开启 GS 编译选项后，直接覆盖返回地址以劫持 ip 寄存器的测试用例将失效。

3.2 SEH 机制及其链式处理

作为 Windows 系统下的异常处理机制之一，SEH 为 Windows 程序设计者提供了对程序错误或异常的处理途径，使得系统更加健壮。在 Microsoft Visual C++中，SEH 异常处理部分主要以 try-except 语句的形式来实现。当执行一个错误或非法指令时，为了能够让程序在发生异常时跳到异常处理代码，在栈中保存有指向这个异常处理例程代码的指针，即每一个函数/过程都有一个栈帧，如果在其中有异常处理，那么相应异常处理例程信息将以 SEH 异常处理结构体的形式储存在栈中。下面以 x86 架构下的 SEH 机制为例进行讲解。SEH 结构体主要包含两个 DWORD 指针：SEH 链表指针和异常处理函数句柄，共 8 个字节，具体如图 1 所示。

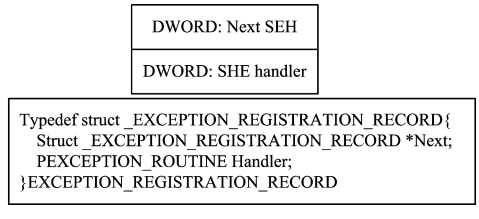

图 1 SEH 结构体及相应图示

Fig. 1 SEH structure and its diagrams

SEH 结构体的前 4 个字节为指向下一个 SEH 结构体的指针,后 4 个字节为一个指向当前异常处理例程的指针。该 SEH 结构体存放在栈中,由 TEB 的 0 字节偏移处 FS:0 指向距离栈顶最近的 SEH 结构。线程初始化时会自动向栈中安装一个 SEH 结构作为线程默认的异常处理。此后如果程序源代码中使用了_try,_except 或 Assert 宏等异常处理机制,编译器将向当前函数栈帧中继续安装一个 SEH 结构来实现异常处理,因此栈中一般会存在多个 SEH 结构。SEH 是基于线程的,且采取链式处理方式。这些 SEH 结构通过链表指针在栈内由栈顶向栈底串成单向链表。异常发生时,操作系统会中断程序查找出错线程的 TEB,从 FS:[0]中取出指向 SEH 链表开头的指针,并顺着 SEH 链表依次尝试执行异常处理函数。SEH 链表在栈中的分布大致如图 2 所示。正是 SEH 的这种链式处理方式,使得在使用 SEH 的过程中出现了能够被攻击者利用的漏洞。

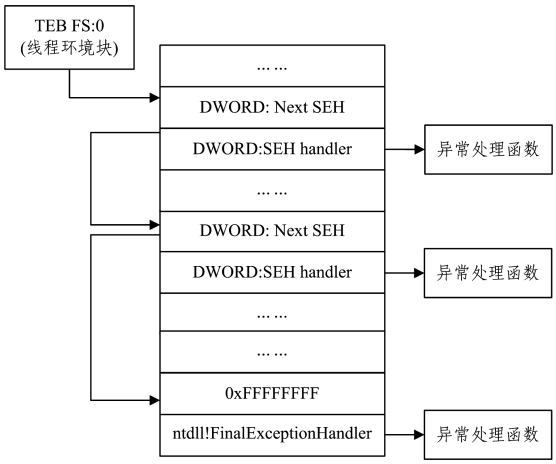

图 2　SEH 链表

Fig. 2　SEH list

4　测试用例生成

4.1　基于 SEH 的测试用例数据布置方案

由于 SEH 结构存放在栈中,溢出缓冲区的数据可覆盖 SEH 结构的内容,进而替换 Next SEH 域及 SEH handler 域为指定内容,并最终导致控制流劫持。同时,由于 GS 机制并未对 SEH 提供保护,若通过 SEH 进行攻击,程序将转入被劫持后的异常处理,进而能在程序检测 canary 值前劫持程序控制流。

根据本方案的测试用例生成策略,系统会构造一段能够造成溢出效果的输入数据,该数据能够实现 3 个效果:首先,将具体的 shellcode 代码布置到离被覆盖 SEH 结构体位置较近的区域;其次将 SEH 结构中的 next SEH 域覆盖为一个 jump code(如 jmp 0x6)指令,通过设计该 jump code 内容,使得该处 jump code 指令执行后能跳转到 shellcode 区域的起始地址处;最后,将 SEH 结构中的 SEH handler 域覆盖为一个指向 pop pop ret 指令串的地址。输入数据的关键数据布置如图 3 所示,其中 shellcode 实际位置既可以在 SEH 结构体之前,也可以在 SEH 结构体之后。

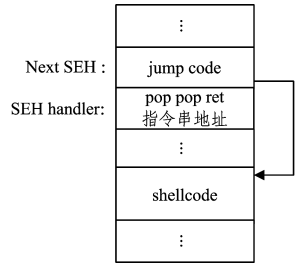

图 3　测试用例中关键数据布置

Fig. 3　Key data layout of test case

依照图 3 进行数据布置后,异常发生时,系统会先在 FS:[0]处找到指向 SEH 链表起始处的指针,然后沿着 SEH 链表依次调用各 SEH 结构中的 Exception handler 所指向的异常处理函数,直到有一个异常处理函数能处理当前异常。而在调用异常处理函数时,首先要将其参数压入栈帧,其函数原型如下。

```
EXCEPTION_DISPOSITION
_cdecl _except_handler(
struct _EXCEPTION_RECORD
    * ExceptionRecord,
void * EstablisherFrame,
struct _CONTEXT  * ContextRecord,
void * DispatcherContext
);
```

其中,参数 EstablisherFrame 域指向当前遍历到的 SEH 结构的 next SEH record 域的地址,因此异常处理函数被调用时,其堆栈分布情况如图 4 所示。

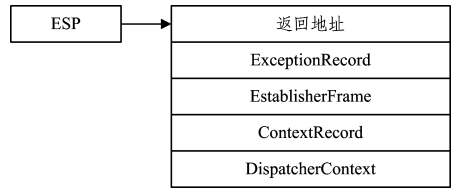

图 4　异常处理函数调用时的堆栈分布

Fig. 4　Layout of the stack when an exception handler function is called

可以看到当一个例程被调用时,被压入的 EstablisherFrame 域位于 ESP+8 处。因此用 pop pop ret 指令串的地址覆盖 SEH handler 后,第一个 pop 将弹出栈顶的 4 bytes,接下来的 pop 继续从栈中弹出 4bytes,最后的 ret 将把此时 ESP 所指的 EstablisherFrame 域中的值(即 next SEH record 的地址)弹入 EIP 中,导致 next SEH record 中存储的 jump code 跳转指令被执行,进而程序流程跳转到 shellcode,完成控制流劫持。

4.2　漏洞风险性检测

可基于 SEH 进行测试用例生成的前提是 SEH 链表中可被运行到的 SEH 结构应能被外部输入影响,因此首先应在异常发生后,对 SEH 结构进行定位并检测其是否能被外部输入

影响，进而才能判断是否可能存在基于 SEH 型的栈溢出漏洞。

如图 2 所示，FS:[0]指向 SEH 链表的第一个 SEH 结构体，由此可定位 SEH 结构。当前函数内发生异常时首先使用 FS:[0]所指向的 SEH 结构体，故首先考虑该 SEH 结构的内容是否会被外部输入影响；将外部输入进行符号化后，即判断 SEH 链表中可被运行的 SEH 结构内容是否为符号值，若是符号值则可被影响，存在基于 SEH 对当前漏洞进行利用的风险，此时进一步进行相应测试用例的构造，若不是符号值则认为 SEH 结构内容不可被影响，无法生成测试用例。

4.3 pop pop ret 指令串搜索

目标程序发生异常后，为生成相应测试用例，需收集构造测试用例所需的 pop pop ret 指令串地址等信息。

首先，通过驱动监视目标程序模块的加载情况，即通过 WDK(Windows 系统的驱动开发套件)提供的用于模块加载通知的用户回调函数 PsSetLoadImageNotifyRoutine 对模块加载进行监视，每当有模块加载时，记录下当前被加载模块的模块名、模块起始地址及模块大小等信息，并将这些信息交付给 S2E 系统。目标程序发生异常后，若判断当前程序存在基于 SEH 被攻击的风险，则在目标程序已加载模块的地址空间中搜索所有可能的 pop pop ret 指令组合，并记录下 pop pop ret 指令串的地址存储在 ppr 数组中。

4.4 自动测试用例生成

检测到漏洞存在后，将按下述过程进行测试用例构造：

1)在内存空间中搜索符号内存，找到离该 SEH 距离较近的一块符号化内存，此后该块内存将用于存储 shellcode。

2)计算被覆盖的 SEH 结构到上述用于存储 shellcode 的内存区域起始地址之间的距离 D，并根据 D 的大小构造 jump code 跳转机器码，用于覆盖 SEH 结构的 next SEH record 域，使得通过该 jump code 可跳转到 shellcode 起始地址处。

3)在已加载的 DLL/EXE 模块而非栈中搜索 pop pop ret，并记录下该指令串的地址，用于接下来覆盖 SEH 结构中的 SEH handler 域。

基于以上过程，分别构建 shellcode_constrain 约束、jmpcode_constraint 约束和 ppr_constraint 约束，3 项约束经过合取即形成测试用例约束 TestCase_constraint，如式(1)所示。

$$\text{TestCase\_constraint} = \text{shellcode\_constraint} \wedge \text{jmpcode\_constraint} \wedge \text{ppr\_constraint} \quad (1)$$

为便于描述约束构建过程，定义如表 1 所列的表示形式。

表 1 约束构建过程函数/数组定义

Table 1 Definition of function/array used in the process of constrains building

函数名/数组名	含义
NotExpr	用于生成一元“取反”形式的 bool 型约束的函数
EqExpr	用于生成二元比较“等于”形式约束的函数
Memory	以内存地址为下标，记录该地址中内存值的数组

4.4.1 shellcode_constraint 约束

shellcode_constraint 约束是对用于存储 shellcode 的内存区域的约束，首先判断该区域的内存大小是否大于或等于预设的 shellcode 代码的长度，若满足则进行 shellcode_constraint 约束构造，逐字节构造约束，判断由该块内存起始地址开始的一段内存区域的值是否能与 shellcode 中相应偏移处的字节相等。

shellcode_constraint 约束的构建过程如下：

```
if formatArea_size < shellcode_size
  return
for i<- 0 to shellcode_size do
shellcode_constraint=EqExpr(Memory[formatArea_startAddr+i],
shellcode[i])
```

4.4.2 jmpcode_constraint 约束

jmpcode_constraint 约束是对跳转到 shellcode 的跳转指令的约束。首先计算具体需要跳转的距离，即计算被覆盖的 SEH 结构到上述用于存储 shellcode 的内存区域起始地址之间的距离 D。由于跳转指令有效部分本身占 2 字节，故为使得该跳转指令能跳转到 shellcode 起始地址，实际下一步需构造约束，以判断 next SEH record 域内容能否与 jmp (D-2)的机器码相等。

jmpcode_constraint 约束的构建过程如下：

```
jmpcode[0]='\xEB'dis=formatArea_startAddr-next_seh_addr-2
jmpcode[1]=dis&0xff
jmpcode[2]='\x90'
jmpcode[3]='\x90'
for i<- 0 to 3 do
jmpcode_constraint=EqExpr(Memory[next_seh_addr+i],jmpcode
[i])
```

4.4.3 ppr_constraint 约束

ppr_constraint 约束是对 SEH 结构中 SEH handler 域的约束，即构造约束判断 SEH 结构中的 SEH handler 域的内容与查找而得的 pop pop ret 指令串的地址(4 字节)是否相等。将预先找到的 ppr 指令串地址存储在 ppr 数组中，则 ppr_constraint 约束的构建过程如下：

```
for i<- 0 to ppr.size do
ppr_addr[0]=ppr[i]&0xff
ppr_addr[1]=(ppr[i]>>8)&0xff
ppr_addr[2]=(ppr[i]>>16)&0xff
ppr_addr[3]=(ppr[i]>>24)&0xff
for j<-0 to 3 do
ppr_constraint=EqExpr(Memory[seh_handle_addr+j],ppr_addr[j])
```

4.5 约束求解及约束调整

4.5.1 测试用例约束求解

将 shellcode_constraint，jmpcode_constraint 和 ppr_constraint3 项约束进行合取即形成 TestCase_constraint 约束，然后调用约束求解器，将当前路径约束与 TestCase_constraint 的数据约束合取后再进行求解，使得所得解既能保证到达漏洞触发点，又能满足测试用例约束，从而能真正完成测试用例所指定的作用。若有解则证明目标程序存在漏洞，优化路径约束，然后进行求解，得到一个满足约束条件的实例。

具体实现时,为提升约束构建及求解效率,对于 shellcode_constraint,jmpcode_constraint 和 ppr_constraint3 项约束,每构建完毕一个约束后,即将其加入到 TestCase_constraint 约束中,再与当前路径约束合取,并判断合取后约束的可解性,可解则再进行下一个约束的构建、加入 TestCase_constraint 约束并再次与路径约束进行合取判断可解性,直至 3 项约束都加入到 TestCase_constraint 约束中。

若 3 项约束都加入到 TestCase_constraint 约束后,TestCase_constraint 约束与路径约束进行合取后仍可解,则化简路径约束,然后进行求解,即可得到一个满足约束条件的实例。由于所得测试用例既满足到达异常点的路径约束又满足本文定义的 TestCase_constraint 约束,因此所得测试用例既能保证到达漏洞触发点,又能满足异常触发后跳转到 shellcode 的关键数据的布置,从而能真正完成测试用例所指定的作用。

若 shellcode_constraint,jmpcode_constraint 和 ppr_constraint 约束中某些加入到 TestCase_constraint 约束后,使得 TestCase_constraint 约束与路径约束合取后判定为不可解,则需按一定规则调整 TestCase_constraint 约束,再次构建后重新判断可解性。

4.5.2　测试用例约束调整

shellcode_constraint,jmpcode_constraint 和 ppr_constraint 约束所约束的内容可能会由于存在坏字符或与路径约束冲突等原因,导致按 3.4 节中的方案构建而得的约束与路径约束合取后求解失败。故此时应在保持测试用例原有功能不被影响的前提下,对这些约束内容按照一定规则稍作调整,重新判断可解性。

若初次将 shellcode_constraint 约束加入到 TestCase_constraint 约束后求解失败,则调整 shellcode 摆放位置,将 shellcode 在当前符号区域后移一位重新尝试构造约束,直至加入 shellcode_constraint 约束后求解成功得到测试用例;或 shellcode 结尾已到达当前符号区域末尾但仍不可解,则判定测试用例生成失败。

由于 jmpcode_constraint 约束的具体内容由 shellcode 位置决定,故若初次将 jmpcode_constraint 约束加入到 TestCase_constraint 约束后求解失败,同样调整 shellcode 的摆放位置。

若初次将 ppr_constraint 约束加入到 TestCase_constraint 约束后求解失败,则选取 ppr 数组中下一个 pop pop ret 指令串地址进行约束构建尝试,直至可解,得到测试用例;或 ppr 数组中所有地址都尝试完毕仍不可解,则判定测试用例生成失败。

5　实验分析

基于 S2E 符号执行平台,依照本文思想构建实现了一个基于 SEH 的漏洞检测和自动测试用例生成方法,该系统可检测 Windows 下 32 位二进制程序中是否存在基于 SEH 机制进行漏洞攻击的风险性,并自动生成相应测试用例。

本文以如表 2 所列的两个已发布的 MP3 播放器为目标程序进行实验验证。其中 Soritong 未开启 GS 保护,Mediacoder 开启了 GS 保护。

表 2　目标程序漏洞信息

Table 2　Vulnerability information of target programs

目标程序	实验平台	漏洞成因
Soritong MP3 player 1.0	Windows XP SP2	安装目录下的 Skin/Default 下的皮肤文件 UI.TXT 过长时将导致缓冲区溢出
Mediacoder 0.8.34.5716	Windows XP SP3	其播放 m3u 文件时存在缓冲区溢出可能

为直观说明实验效果,以 Mediacoder 二进制程序的测试用例生成过程为例,对本文方法进行演示。ReadFile 函数处的符号引入效果如图 5 所示。

```
ReadFile( ) return hook invoked at 0x10001902
fd      : 0x3c4
foffset : 0
esp     : 0x12e1d8
address : 0x24ee60
length  : 0x3a0
taint-var introduce: range : [848, 928]
[0x24f1b0] = (Read w8 0x0 v0_exploit.m3u_offset_848_0)
[0x24f1b1] = (Read w8 0x1 v0_exploit.m3u_offset_848_0)
[0x24f1b2] = (Read w8 0x2 v0_exploit.m3u_offset_848_0)
[0x24f1b3] = (Read w8 0x3 v0_exploit.m3u_offset_848_0)
```

图 5　符号引入效果

Fig. 5　Result of symbolic import

异常发生后,记录异常时刻 eip 寄存器的内容并检验 SEH 结构是否被符号化,若被符号化,则输出被符号化的 Next SEH 域的地址及当前加载的模块数目,如图 6 所示。

```
pc=0x5d0a76e4
TB: SEH is symbolic
next_seh_addr=0x12f434
--hz seh TCgeb:--
Hmodule_list.siaze() = 86
```

图 6　异常时刻信息

Fig. 6　Information of exception time

生成的测试用例部分的内容如图 7 所示,其中阴影部分即为输入后将覆盖 SEH 结构的数据。

为便于观察实验效果,本实验采用弹出计算器的 shellcode 生成测试用例。将本系统生成的测试用例作为输入运行目标程序,可见弹出计算器,故生成的测试用例有效。

```
00000320 0F DF FA 08 B0 35 38 35 33 BC C0 C2 2B B5 C5 8F  .....5853...+...
00000330 EB 25 B7 80 99 49 64 A0 8B 29 EB 32 57 AE 42 42  .%...Id..).2W.BB
00000340 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
00000350 42 42 42 42 42 42 42 42 42 42 42 42 EB 06 90 90  BBBBBBBBBBBB....
00000360 EE 04 01 66 E9 A5 FD FF FF 44 44 44 44 44 44 44  ...f.....DDDDDDD
00000370 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
```

图 7　测试用例展示

Fig. 7　Display of test case

实验结果表明,本文提出的方法可针对 Win32 平台下的二进制程序,自动基于 SEH 精确地生成执行指定 shellcode 的测试用例。

结束语　本文针对目前测试用例自动生成过程中存在的模式单一、准确度不高等问题,提出了一种基于 SEH 的漏洞自动检测与测试用例自动生成方法。该方法能检测 Windows 下二进制程序被基于 SEH 攻击的风险性,并自动生成相应测

试用例。本文最后通过实验验证了测试用例的有效性。

参 考 文 献

[1] 林桠泉. 漏洞战争:软件漏洞分析精要[M]. 北京:电子工业出版社,2016.

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

[3] PIETREK M. A Crash Course on the Depths of Win32 Structured Exception Handling[J]. Microsoft Systems Journal,1997,1.

[4] XU Y F,ZAHNG J H,WEN W P. Windows Security:The gradual improvement of SEH mechanism [J]. Netinfo Security,2009(5):47-50. (in Chinese)
徐有福,张晋含,文伟平. Windows 安全之 SEH 安全机制分析[J]. 信息网络安全,2009(5):47-50.

[5] HE L,SU P L. Automatic software vulnerabilities exploit generation research progress [J]. China Education Network,2016(z1):46-48.
和亮,苏璞睿. 软件漏洞自动利用研究进展[J]. 中国教育网络,2016(z1):46-48.

[6] AVGERINOS T,SANG K C,REBERT A,et al. Automatic exploit generation[J]. Communications of the Acm,2014,57(2):74-84.

[7] 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. IEEE Computer Society,2012:78-87.

[8] YAN S,WANG R,SALLS C,et al. SOK:(State of) The Art of War:Offensive Techniques in Binary Analysis[C]// Security and Privacy. IEEE,2016:138-157.

[9] CHIPOUNOV V,GEORGESCU V,ZAMFIR C,et al. Selective Symbolic Execution[C]// The Workshop on Hot Topics in System Dependability. 2009:1286-1299.

[10] 吴世忠,郭涛,董国伟,等. 软件漏洞分析技术[M]. 北京:科学出版社,2014:134.

[11] ZHANG Y F. Improving the Scalability and Feasibility of Symbolic Execution [D]. Changsha:National University of Defense Technology,2013. (in Chinese)
张羽丰. 符号执行可扩展性及可行性关键技术研究[D]. 长沙:国防科技大学,2013.

[12] CADAR C,DUNBAR D,ENGLER D. KLEE:unassisted and automatic generation of high-coverage tests for complex systems programs[C]// Usenix Conference on Operating Systems Design and Implementation. USENIX Association,2009:209-224.

[13] STUMP A. CVC:a Cooperating Varidity Checher[C]// Proc. of International Conference on Computer-Aided Verification. 2002.

[14] GANESH V,DILL D L. A Decision Procedure for Bit-Vectors and Arrays[C]// Computer Aided Verification,International Conference,CAV 2007. Berlin:DBLP,2007:519-531.

[15] MOURA L D,BJØRNER N. Z3:An Efficient SMT Solver[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin:Springer,2008:337-340.

[16] 王清. 0day 安全:软件漏洞分析技术(第 2 版)[M]. 北京:电子工业出版社,2011.