doi:10.11887/j.cn.202003004 http://journal.nudt.edu.cn

多模块ROP碎片化自动布局方法*

黄 宁,黄曙光,潘祖烈,常 超
(国防科技大学 电子对抗学院,安徽 合肥 230000)

摘 要:返回导向式编程(Return Oriented Programming, ROP)是一种可有效绕过数据执行保护机制的技术。ROP通过搜索内存代码区中合适的汇编指令片段,可组成一段执行特定功能的程序。已有的ROP自动构造技术只考虑ROP链的功能实现,而忽视了ROP链布局对程序内存可控性的要求,导致自动生成的ROP链实用性低。为解决该问题,提出了一种基于符号执行的多模块ROP碎片化自动布局方法。该方法在ROP自动构造Q框架的基础上,以模块为单位对ROP链进行切片;使用符号执行工具S2E,对控制流劫持状态下的程序内存状态进行动态分析;为各ROP模块匹配相应的可控内存区域,构造碎片化布局的ROP链。实验证明,相比已有技术,该方法生成的ROP链有效降低了对程序内存可控性的要求。

关键词:数据执行保护;返回导向式编程;符号执行;碎片化布局
中图分类号:TP311 **文献标志码**:A **文章编号**:1001-2486(2020)03-022-08

Automatic fragmented layout for multi-module ROP

HUANG Ning, *HUANG Shuguang*, *PAN Zulie*, *CHANG Chao*
(College of Electronic Engineering, National University of Defense Technology, Heifei 230000, China)

Abstract: ROP (return-oriented programming) is a technique which is able to bypass the protection of the DEP (data execution prevention). The ROP can constitute a program that performs a specific function by searching for an appropriate assembly instruction fragment in the memory code area. Previous methods for automatic generation of ROP do not consider the limitation of the layout of ROP caused by the program memory requirement, which leads to poor practicability of ROP. In order to solve this problem, a new method for automatic fragmented layout of multi-module ROP based on symbolic execution was proposed. The ROP chain was divided into different modules on the basis of automatic ROP generation framework Q; the controllability of memory was dynamically analyzed by using symbolic execution tool S2E; the controllable memory areas for each ROP module was found, and the fragmented layout ROP was automatically constructed. Experiments show that, compared with the previous methods, the ROP chain generated by the proposed method can effectively reduce the requirements for the program memory controllability.

Keywords: data execution prevention; return-oriented programming; symbolic execution; fragmented layout

随着信息技术的发展,软件漏洞的发掘与利用成了一个热点问题。针对不同类型的漏洞利用技术,各种保护机制也层出不穷。但是,多年的漏洞利用实践证明,由于各方面条件的限制,依然存在许多可绕过这些保护机制,成功实施漏洞利用的技术手段。

以二进制程序漏洞利用为例,由于计算机无法区分内存空间中二进制码的代码或数据属性,可能导致进程执行外部数据,从而造成控制流劫持攻击[1]。针对这一问题,Linux和Windows等主流操作系统相继引入了数据执行保护(Data Execution Prevention, DEP)机制[2]。该机制的基本原理是,通过标记程序内存页为可执行/不可执行,实现内存空间代码区和数据区的区分。在DEP环境下,位于数据区的恶意代码将无法被执行,从而阻止控制流劫持攻击[3]。

由于DEP机制不会拦截可执行页面中的代码指令,solar designer提出了ret to libc(ret2libc)方法。该方法通过劫持程序控制流,使程序跳转至已有的系统函数。Schacham等[4-5]在ret2libc思想的基础上,提出了返回导向式编程(Return Oriented Programming, ROP)技术。相比ret2libc,ROP使用更小的汇编指令片段(gadget),提高了该类方法的泛用性。Lu等[6]基于Rix的方法,提出了可压缩、可打印的ROP构造方法,提高了ROP载荷的灵活性。

* **收稿日期**:2018-11-26
基金项目:国家重点研发计划"网络空间安全"重点专项资助项目(2017YFB0802905)
作者简介:黄宁(1990—),男,广东广州人,博士研究生,E-mail:tsukimurarin@163.com;
黄曙光(通信作者),男,教授,博士,博士生导师,E-mail:809848161@qq.com

近年来出现多种针对控制流劫持类漏洞自动化分析[7]与测试用例自动生成技术[8-10]。Schwartz 等提出了面向二进制程序漏洞的 ROP 自动构造框架 Q[11-12]。其工作流程:首先,向 Q 框架可执行文件,搜索其中具备特定功能的 gadget 集合;对面向 gadget 的高级语言进行语义分析,构建中间指令序列;分析中间指令序列,为每条中间指令分配合适的 gadget 集合,形成 ROP 链。Q 框架的局限性[13]在于,该方案生成的测试用例仅从功能实现的角度出发,未考虑 ROP 布局对程序内存可控性条件的要求,降低了 ROP 链的实用性。

为解决上述问题,本文提出了基于符号执行的 ROP 碎片化自动布局方法。该方法将 ROP 链以模块为单位,切割成长度不一的碎片;使用导向式符号执行技术,引导源程序运行至控制流劫持点的同时,检查程序中可控内存区域是否满足 ROP 模块布局要求;以 ROP 模块切片与可控内存区域布局为根据,构建碎片化 ROP 链的数据约束;通过求解数据约束,自动生成满足程序可控内存分布条件的碎片化 ROP 链。该方法解决了 ROP 链对内存可控性要求高的问题,提高了 ROP 链的实用性。

1　ROP 技术原理

ROP 技术基于 ret2libc 技术发展而来。该技术主要针对 DEP 机制未限制代码页中已有代码的执行权限这一缺陷,实现 DEP 机制绕过[14]。其主要原理是,通过搜索程序内存代码页,构建以 ret 等跳转指令结尾的汇编指令片段 gadget 集合。从 gadget 集合中筛选出符合条件的部分,组合成一段可实现特定功能的 ROP 代码链。图 1 表示了一个 ROP 链的代码执行顺序及其相应的栈空间数据分布。

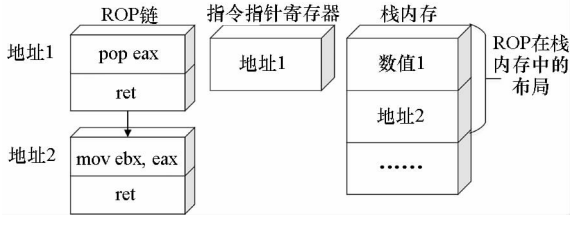

图 1　ROP 链在栈内存中布局结构示意图

Fig. 1　ROP chain and the structure of stack

Q 框架以自定义高级语言 ROPL 表示 ROP 链的目标功能程序。ROPL 高级语言到汇编指令序列的转换过程为:ROPL 高级语言—ROPL 中间表示序列—中间指令序列—gadget 汇编指令序列(ROP 链)。

目标程序模块指的是一个可执行相对独立功能的 ROPL 代码序列的集合,其结构类似于 C 语言中的函数。

ROPL 语言框架下,多模块目标程序定义为:目标程序包含至少两个以上模块,且调用了至少一个以上非 main 模块。多模块 ROP 中,除 main 模块外的 ROP 模块的切换过程由以下三个子过程组成:目标模块开始调用过程,目标模块参数初始化过程,目标模块返回调用过程。

2　整体思路

已有的 ROP 自动生成技术主要解决了 gadget 搜索与分类,高级语言语义分析,以及面向中间指令的 gadget 分配与排列三个方面的问题,实现了 ROP 链自动构造。但从实际运用效果看,Q 框架仍然无法满足多数场景下源程序的内存可控性状态对 ROP 链布局的限制。为解决这一问题,本文在 Q 框架的基础上,提出了基于碎片化布局的 ROP 自动构造方法。该方法构造碎片化布局的 ROP 链过程如图 2 所示。

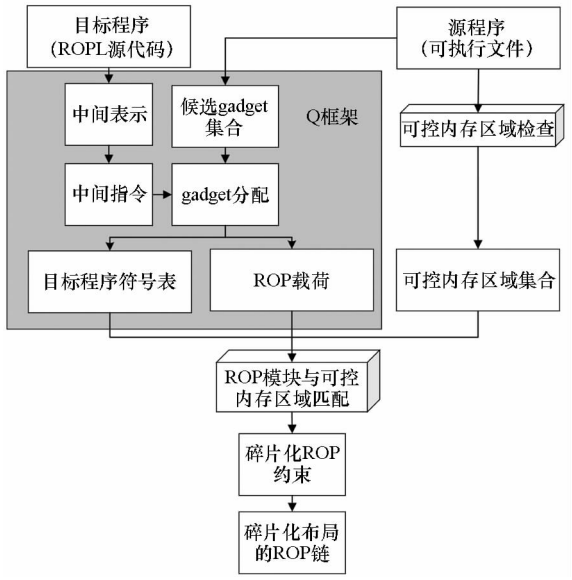

图 2　ROP 碎片化自动布局过程示意图

Fig. 2　Overview of ROP fragmented layout and automatic generation

在 Q 框架生成 ROP 载荷与目标程序符号表的基础上,本文将结合对源程序可控内存区域的检查结果,构建碎片化 ROP 约束,求解约束,生成碎片化 ROP 链。

为了避免符号执行对每条程序路径遍历导致的路径爆炸问题,本文在符号执行工具 S2E 的基础上,采用了经过路径选择算法优化的导向式符号执行技术[15-16]。以 crash 文件作为源程序的输入文件,可引导源程序沿着确定的程序路径动态运

行,直至触发控制流劫持状态。图 3 是通过导向式符号执行触发源程序控制流劫持状态的过程。

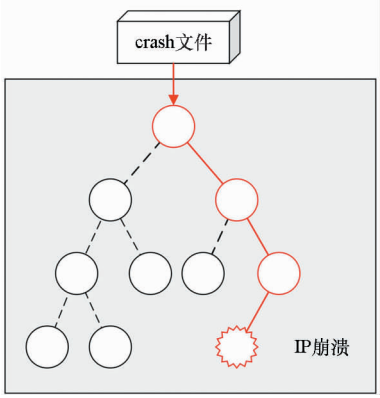

图 3　导向式符号执行路径选择过程示意图

Fig. 3　Path selection of source program with path-oriented symbolic execution

本文在导向式符号执行过程中,收集程序堆栈状态与可控内存区域状态。结合 ROP 载荷,分析上述状态是否满足 ROP 链的布局条件,并构建相应的 ROP 数据约束。通过约束求解,可实现碎片化布局的 ROP 测试例自动生成。

3　具体实现

3.1　可控内存区域搜索

如图 1 所示,ROP 链是由一个特定顺序的 gadget 序列组成的。由于每个 gadget 均以 ret 指令结束,并以此控制程序跳转至下一个 gadget 所在地址,其地址及 gadget 的相关操作数需存放于程序的堆栈中。当源程序处于控制流劫持状态时(即指令寄存器中的数值为符号值),堆栈是否有足够的可控空间用于 ROP 布局,决定了 ROP 是否适用于源程序。为此,本文首先对源程序内存状态进行分析,确定 ROP 布局条件。

针对本文方法的内存分析过程涉及的关键数据检查与状态变化,本文有如下定义:

symbolicBlock < *symbolicAddr*, *symbolicSize* >:表示源程序在控制流劫持状态下,除当前栈帧外一段连续的符号化内存区域。其中,*symbolicAddr* 表示该区域的起始地址,*symbolicSize* 表示该区域的长度。

stackPtr:表示首次控制流劫持状态下的当前栈顶指针。

stack_symbolicLength:若当前栈顶位置处于符号化区域中,该数值表示以 *stackPtr* 为起始地址的一段连续的符号化内存长度。

$mLen_{id}$表示 ROP 中名称为 id 的模块占用内存长度。每个模块的长度信息均记录于中间表示符号表中。

在一般的溢出漏洞中(比如栈溢出漏洞),覆盖当前栈顶位置的污点数据的起始地址通常位于上一个函数栈帧中。但对于执行 gadget 序列来说,需要关注的只是源程序的控制流劫持时刻的栈顶数据属性。因此,本文将在源程序控制流劫持时刻,从栈顶位置开始进行符号化检查。若源程序栈顶位置不为符号化数据,表示源程序无法跳转至第二个 gadget,不满足 ROP 开始执行的初始条件;若栈顶数据为符号化数据,计算以栈顶位置开始的符号化数据长度。图 4 显示的是源程序控制流劫持时刻,满足 ROP 布局条件的栈结构示意图。

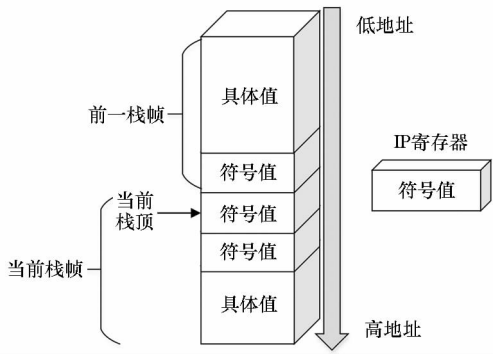

图 4　源程序控制流劫持时刻的栈内存结构示意图

Fig. 4　Structure of stack at the time of control-flow hijacked

为满足当前栈帧可控空间不足情况下 ROP 的布局要求,可控内存搜索算法将搜索并记录进程用户态内存空间中其余的符号化区域信息。其过程如算法 1 所示。

算法 1　可控内存区域搜索

Alg. 1　Searching for controllable memory area

输入:源程序内存空间 memory
输出:可控内存区域集合 memSet

```
for each byte ∈memory
  if  byte 是符号值
    if  byte 的前一个字节不是符号值
        symbolicAddr← byte 的地址
        symbolicSize← 1
    end if
  else
        symbolicSize← symbolicSize + 1
  end else
  end if
  if  byte 不是符号值
    if byte 的前一个字节是符号值
      memSet 集合中插入 symbolicBlock
    end if
  end if
end for each
```

3.2　ROP 链碎片化自动布局

本文从可控内存区域集合 *memSet* 中寻找满足 ROP 模块布局条件的元素。候选的符号化区域长度 *symbolicSize* 需要至少满足容纳某一 ROP 模块,且该区域包含的范围与源程序控制流劫持状态下的栈顶指针不应相互冲突。对集合 *memSet* 中的所有 *symbolicBlock* 元素进行第一轮过滤,选出若干符合 ROP 模块长度条件的内存区域,将该元素加入对应 ROP 模块的候选区域集合 $lenBlocks_{id}$ 中,该模块的候选区域集合需满足如式(1)所示条件。$lenBlocks_{id}$ 表示经过第一轮过滤后,模块名为 id 的 ROP 对应的所有候选区域集合。

$lenBlocks_{id}$:

$$\forall block \in symbolicBlock \mid (symbolicSize > mLen_{id}) \wedge [(stackPtr < symbolicAdd) \vee (stackPtr > symbolicAdd + symbolicSize)] \tag{1}$$

在候选区域长度满足 ROP 模块长度要求的基础上,通过比较候选区域数据可控性约束与 ROP 模块数据约束的兼容性,对每个 ROP 模块的候选区域集合进行第二轮过滤。第二轮过滤完成后,ROP 模块 id 对应的候选区域集合 $conBlocks_{id}$ 应满足式(2)所示条件。

$conBlocks_{id}$:

$$\forall block \in lenBlock_{id} \mid (canArea \subseteq block) \wedge (canArea.\ Size \geqslant ropModule_{id}.\ Size) \wedge Eq(area, module_{id}) = true \tag{2}$$

式中,*canArea* 表示候选区域 *block* 中任意连续的可控内存区域;*canArea. Size* 表示 *canArea* 的长度;$ropModule_{id}.\ Size$ 表示 ROP 模块 id 的长度,$module_{id}$ 表示该模块的数据约束;函数 *Eq* 用于实现约束条件 *A* 与约束条件 *B* 的兼容性比较。

$$result = Eq(A, B)$$

当约束比较函数 $Eq(A, B)$ 的返回值 *result* 为 *true* 时,表示约束条件 *A* 与 *B* 可兼容;为 *false* 时,表示 *A* 与 *B* 不可兼容。

当且仅当模块 *ropModule* 的数据约束 *module* 与 *canArea* 区域可控性约束 *area* 的兼容性判断结果为 *true* 时,该 ROP 模块是可执行的。数据约束 *module* 与 *area* 相兼容的检查条件包括以下两项:*module* 中所有连续字节的约束与 *area* 所有连续字节的可控约束相兼容;*module* 中剩余的字节数应不大于 *area* 中剩余的字节数,即 *canArea* 应有足够的可控空间容纳该 ROP 模块。

针对 *area* 与 *module* 的兼容性比较过程以字节为单位。对 $ropModule_{id}$ 中的每个字节与 *canArea* 区域中每个字节的约束条件逐一比较,构造待求解的模块数据约束 *mConstraint*。该过程如算法 2 所示。

算法 2　ROP 模块数据约束构造

Alg. 2　Construction for data constraint of ROP module

输入:当前 canArea 可控性约束 area
　　当前 ropModule 数据约束 module
输出:ROP 模块布局约束 mConstraint
　　ROP 模块与可控内存区域兼容性检查结果 isAvailable

```
for each mByte in  module
  for each aByte  in  area
    if  mByte 与 aByte 约束兼容 and area 剩余字节 < module 剩余字节
      mConstraint ←mConstraint ∧mByte
    end if
    else
      mConstraint ← false
    end else
  end for each
end for each
/ * 约束求解器求解 * /
isAvailable← solve( mConstraint)
```

算法 2 中,若 *isAvailable* 返回值为 *true*,表示 *canArea* 区域满足 ROP 模块 id 的布局条件,并将 *canArea* 区域标记为不可控区域后,重新构建可控内存区域集合 *memSet*。图 5 表示完成碎片化自动布局后,多模块 ROP 的执行过程。针对剩余 ROP 模块,重复执行第一轮与第二轮过滤,直至所有 ROP 模块完成布局区域分配。该过程如算法 3 所示。

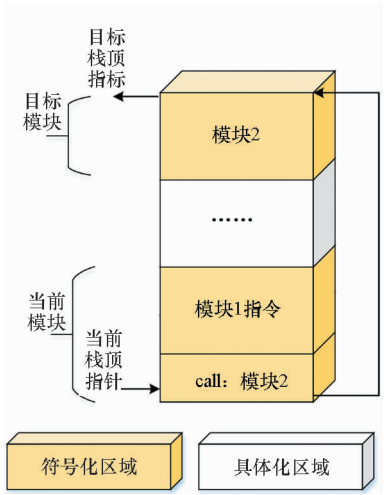

图 5　碎片化多模块 ROP 执行过程示意图

Fig. 5　Execution process of fragmented multi-module ROP

算法 3 ROP 碎片化约束构造

Alg. 3 Construction of data constraint of ROP fragment

输入:所有 ROP 模块 ropModules;可控内存区域集合 memSet

输出:碎片化 ROP 约束 ropConstraint

```
for each module in ropModules
  lenBlock← 第一轮过滤(ropModule, memSet)
  conBlock← 第二轮过滤(ropModule, lenBlock)
  for each block  in conBlock
      (isAvailable, newConstraint)← 执行算法 2
      if  isAvailable == true
          将 block 区域标记为不可控区域
          ropConstraint← ropConstraint ∧
                         newConstraint
      end if
      memSet← 执行算法 1
  end for each
end for each
```

4 实验与分析

4.1 ROP 碎片化布局可行性分析

以代码 1 中的 ROPL 高级语言作为目标程序。通过对目标程序进行语义分析,其对应的高级语言中间表示符号表如代码 2 所示。

代码 1

Code 1

```
function main( )
f1( )
function f1( )
foo( )
function foo( )
x = 1
```

代码 2

Code 2

```
0x00  Enter(0): main
0x04  Call(f1, args:)
0x30  Ret(main)
0x4C  Enter(0): f1
0x50  Call(foo, args:)
0x7C  Ret(f1)
0x98  Enter(0): foo
0x9C  Assign(x, Const 1)
0xC4  Ret(foo)
0xE0  Global_end
```

针对代码 1 所示目标程序生成的 ROP 链长度为 224(0xE0)B。该 ROP 链分为 3 个模块:模块 main 的长度为 76(0x4C)B;模块 f1 的长度为 76(0x4C)B;模块 foo 的长度为 72(0x48)B。

为验证通过本文方法实现的 ROP 链在实际程序中的碎片化布局效果,选取了 7 个包含控制流劫持漏洞的源程序进行实验验证。本文将可能覆盖程序内存关键位置的污点数据标记为符号值,并通过构建 ROP 数据约束及约束求解,判断相应的内存位置是否满足 ROP 布局条件。表 1 为各实验程序的实验环境。

表 1 漏洞程序实验环境

Tab. 1 Experimental environment for programs

漏洞样本	系统环境	源程序	初始符号化数据长度
MS06 – 055	Windows 2000	IE 6	200 MB
CVE – 2010 – 3333	Windows XP	Word 2003	400 B
CVE – 2012 – 0158	Windows XP	Word 2003	400 B
CVE – 2014 – 0322	Windows 7	Flash 11	4608 B
CVE – 2015 – 5119	Windows 7	Flash 11	2048 B
CVE – 2017 – 11882	Windows 7	Word 2013	9400 B
CVE – 2018 – 8174	Windows 7	IE 10	420 B

通过漏洞触发代码触发源程序首次控制流劫持状态。本文对首次控制流劫持时刻的程序内存状态进行分析,内存各区域可控污点数据情况如表 2 所示。

表 2 控制流劫持状态时刻源程序可控内存区域分布情况

Tab. 2 Layout of controllable tainted data in memory at the first time of control flow hijacked

漏洞样本	栈 *	堆	数据段
MS06 – 055	92 B	200 MB	0 B
CVE – 2010 – 3333	16 B	100 MB	0 B
CVE – 2012 – 0158	400 B	0 B	0 B
CVE – 2014 – 0322	4 B	380 MB	0 B
CVE – 2015 – 5119	4 B	380 MB	0 B
CVE – 2017 – 11882	600 B	10 MB	0 B
CVE – 2018 – 8174	12 B	420 B	0 B

注:“ * ”表示栈内存中的可控数据长度从当前栈顶位置开始计算表示。

对比表 1 与表 2 中的数据可发现,部分漏洞

程序的引入污点数据长度与首次控制流劫持状态时刻的内存各部分可控污点数据长度并不完全一致。造成这一现象的原因主要有以下几种:

1)引入污点数据通过复制等操作传播至内存其他区域,造成部分污点数据的约束条件有重合部分。属于该类情况的漏洞程序包括 CVE - 2017 - 11882 等。

2)污点数据覆盖其他内存区域的关键数据。属于该类情况的漏洞程序包括 CVE - 2014 - 0322 等。

此类漏洞利用方式为,通过数组越界读写,实现程序任意内存地址读写,进而导致函数地址覆盖与程序控制流劫持。由于函数地址在内存代码段,因此不在本文对 ROP 布局内存分析的范围内。

3)污点数据符号属性丢失或污点数据不可控。属于该类情况的漏洞程序包括 CVE - 2010 - 3333 等。

CVE - 2010 - 3333 是栈溢出漏洞,但在栈内存中,可控污点数据仅在栈顶位置向下 16 B 的范围内。对于不可控的污点数据,由于不具有利用价值,因此,本文未做统计与分析。

在对源程序内存可控污点数据布局状态分析的基础上,针对代码 1 目标程序构造的 ROP 链经过碎片化布局处理后,各模块 ROP 在内存空间中的分布如表 3 所示。

表 3　各模块 ROP 在源程序内存中的布局情况

Tab. 3　Layout of fragmented ROP chain

漏洞样本	main 模块	f1 模块	foo 模块
MS06 - 055	栈	堆	堆
CVE - 2010 - 3333	堆	堆	堆
CVE - 2012 - 0158	栈	栈	栈
CVE - 2014 - 0322	堆	堆	堆
CVE - 2015 - 5119	堆	堆	堆
CVE - 2017 - 11882	栈	堆	堆
CVE - 2018 - 8174	堆	堆	堆

MS06 - 055 的栈内存仅满足 main 模块的布局条件。本文通过堆喷射漏洞触发代码,实现源程序内存中的堆块大量布局,并将写入堆块的数据标记为污点数据。通过内存分析,确定进程的堆内存满足 f1 与 foo 模块的布局条件。

CVE - 2010 - 3333 的栈内存不满足任一模块布局条件。本文对该漏洞实验做了手工调整,在其栈内存中布置了简化的堆栈伪造指令,使其堆栈指针指向堆内存区域。通过分析,该进程堆中的可控数据区域满足代码 2 所有模块的布局条件。

CVE - 2012 - 0158 漏洞的栈内存布局情况满足代码 2 所有模块的布局条件,因此未将任何一个模块布置于其他内存区域中。

CVE - 2014 - 0322 与 CVE - 2015 - 5119 漏洞的栈内存不符合代码 1 中模块布局条件。为了验证碎片化布局方法,本文首先通过伪造栈空间的方法,在堆内存中开辟一段伪造的栈内存。与上述两个漏洞布局进行对比,f1 与 foo 模块均布置于堆内存中。实验结果证明,CVE - 2014 - 0322 与 CVE - 2015 - 5119 的实验程序内存满足表 3 所示的布局条件。

根据表 2,CVE - 2017 - 11882 栈内存的可控空间大于代码 1 ROP 所需的内存空间长度。但通过进一步分析发现,该漏洞程序的栈内存空间过于碎片化,其中最大的一块栈内存可控区域长度仅为 120 B,不足以容纳代码 1 中的任意两个模块。因此,本文仅将 main 模块布局于栈内存中,f1 与 foo 模块布局于堆内存中。CVE - 2017 - 11882 的 ROP 碎片化布局实验结果如表 3 所示。

CVE - 2018 - 8174 漏洞程序的栈内存可控空间仅满足进程跳转功能,不满足代码 1 ROP 模块布局功能,因此,该漏洞程序的 ROP 模块全部布局于堆内存中。

4.2　案例分析

本文挑选了 CVE - 2014 - 0322 漏洞实验样本,对其 ROP 碎片化布局过程进行具体分析与阐述。

当符号执行工具 S2E 检测 CVE - 2014 - 0322 样本程序进入控制流劫持状态时,首先检查源程序状态结果为:

栈顶指针寄存器 ESP:0x0307B7D4;

符号化寄存器:EAX;EIP;

符号化内存区域:0x0307B7D4 ~ 0x0307B7D7;0x10000000 ~ 0x28180000;0x28180000 ~ 0x28280000。

如上文所述,控制流劫持状态下的样本程序栈内存不符合任何一个 ROP 模块的布局要求。为此,需要向样本程序中加入伪造栈空间的步骤。该步骤通过下述 gadget 完成。

XCHG EAX, ESP;

RETN;

该 gadget 的内存地址为 0x5ED6E850。

伪造的栈空间栈顶指针为:0x1A1B3010。根据对比,发现该区域位于符号化区域中。

针对 main 模块进行两轮可控内存区域过滤后,该模块的最终布局范围为:0x1A1B3010 ~ 0x1A1B305C。

针对 f1 模块进行两轮可控内存区域过滤后,该模块的最终布局范围为:0x1A1B4010 ~ 0x1A1B405C。

针对 foo 模块进行两轮可控内存区域过滤后,该模块的最终布局范围为:0x1A1B5010 ~ 0x1A1B5058。

4.3 系统时间开销

为验证本文方法的时间开销,本文记录了该方法的时间开销为 $t1$。时间开销 $t1$ 的定义为:从源程序开始符号执行,到生成碎片化布局的 ROP 测试用例所消耗时间。

作为对比,本文还记录了符号执行工具 S2E 对源程序进行分析的时间开销 $t2$。时间开销 $t2$ 的定义为:从源程序开始符号执行,到生成控制流劫持路径测试用例的消耗时间。对各实验样本进行 10 次实验,各样本的平均时间消耗如图 6 所示。

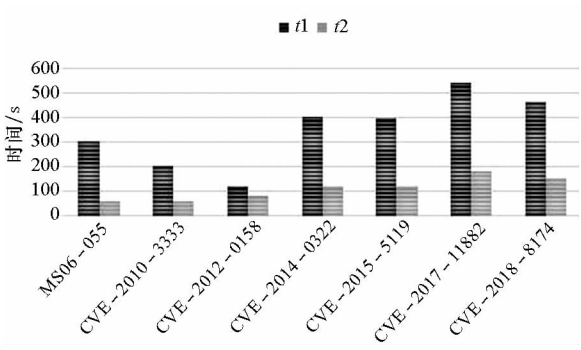

图 6 碎片化 ROP 布局方法与 S2E 系统时间开销对比

Fig. 6 Comparison of analysis time by ROP fragmented layout method and S2E

由于本文提出的碎片化 ROP 布局方法是在源程序控制流劫持状态下,通过内存分布状态分析实现的,因此,图 6 中各实验样本的时间开销之差,即 $t1 - t2$,基本可被视为可控内存区域分析过程的时间消耗。

图 6 所示实验样本中,$t1$ 与 $t2$ 的差值较大。原因是 crash 文件触发源程序控制流劫持状态过程中,使用了堆喷射技术,使源程序内存空间存在大量可控区域,导致内存搜索算法运行时间增加。

5 结论

针对目前 ROP 自动化构造过程中存在的空间效率低与内存布局要求高等问题,提出了基于符号执行的多模块 ROP 碎片化自动布局方法。该方法以 ROP 模块为单位,在动态分析源程序可控污点数据分布情况的基础上,实现各模块 ROP 的碎片化分布,降低了 ROP 布局对源程序内存布局条件的要求。

此外,本文提出的基于碎片化布局的多模块 ROP 自动生成方法仍然存在局限性。首先,ROP 模块调用过程未考虑地址随机化对目标模块地址寻找的影响。其次,符号执行过程中造成的污点数据符号属性丢失等问题会影响该方法对源程序内存状态分析的结果。如何减少上述问题对 ROP 自动生成过程的影响,是下一步工作的研究重点。

参考文献(References)

[1] 邵思豪, 高庆, 马森, 等. 缓冲区溢出漏洞分析技术研究进展[J]. 软件学报, 2018, 29(5): 1177 - 1198.
SHAO Sihao, GAO Qing, MA Sen, et al. Progress in research on buffer overflow vulnerability analysis technologies[J]. Journal of Software, 2018, 29(5): 1177 - 1198. (in Chinese)

[2] 高迎春, 周安民, 刘亮. Windows DEP 数据执行保护技术研究[J]. 信息安全与通信保密, 2013(7): 77 - 79, 82.
GAO Yingchun, ZHOU Anmin, LIU Liang. Data-execution prevention technology in windows system[J]. Information Security and Communications Privacy, 2013(7): 77 - 79, 82. (in Chinese)

[3] 魏强, 韦韬, 王嘉捷. 软件漏洞利用缓解及其对抗技术演化[J]. 清华大学学报(自然科学版), 2011, 51(10): 1274 - 1280.
WEI Qiang, WEI Tao, WANG Jiajie. Evolution of exploitation and exploit mitigation[J]. Journal of Tsinghua University(Science & Technology), 2011, 51(10): 1274 - 1280. (in Chinese)

[4] Shacham H. The geometry of innocent flesh on the bone[C]// Proceedings of the ACM Conference on Computer and Communications Security, 2007: 552 - 561.

[5] Buchanan E, Roemer R, Shacham H, et al. When good instructions go bad: generalizing return-oriented programming to RISC[C]//Proceedings of the ACM Conference on Computer and Communications Security, 2008: 27 - 38.

[6] Lu K J, Zou D B, Wen W P, et al. Packed, printable, and polymorphic return-oriented programming[C]//Proceedings of International Workshop on Recent Advances in Intrusion Detection, 2011: 101 - 120.

[7] 常超, 刘克胜, 谭龙丹, 等. 基于图模型的 C 程序数据流分析[J]. 浙江大学学报(工学版), 2017, 51(5): 1007 -

1015, 1050.
CHANG Chao, LIU Kesheng, TAN Longdan, et al. Data flow analysis for C program based on graph model[J]. Journal of Zhejiang University (Engineering Science), 2017, 51(5): 1007 – 1015, 1050. (in Chinese)

[8] Chipounov V, Kuznetsov V, Candea G. The S2E platform: design, implementation, and applications [J]. ACM Transactions on Computer Systems, 2012, 30(1): 1 – 49.

[9] 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]// Proceedings of IEEE Sixth International Conference on Software Security and Reliability (SERE), 2012: 78 – 87.

[10] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code[C]// Proceedings of IEEE Symposium on Security and Privacy, 2012: 380 – 394.

[11] Schwartz E J, Avgerinos T, Brumley D. Q: exploit hardening made easy[C]// Proceedings of the 20th USENIX Conference on Security, 2011: 25.

[12] Bhatkar D, Jager I, Avgerinos T, et al. BAP: a binary analysis platform[C]. International Conference on Computer Aided Verification, 2011: 463 – 469.

[13] 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(2): 46 – 48.
HE Liang, SU Purui. Research progress on automatic exploitation of software vulnerabilities[J]. China Education Network, 2016(2): 46 – 48. (in Chinese)

[14] Wang J, Xie P, Wang Y, et al. A survey of return-oriented programming attack, defense and its benign use [C]// Proceedings of 13th Asia Joint Conference on Information Security, 2018: 83 – 88.

[15] 黄晖, 陆余良, 刘林涛, 等. 控制流污点信息导向的符号执行技术研究[J]. 中国科学技术大学学报, 2016, 46(1): 21 – 27.
HUANG Hui, LU Yuliang, LIU Lintao, et al. A research on control-flow taint information directed symbolic execution[J]. Journal of University of Science and Technology of China, 2016, 46(1): 21 – 27. (in Chinese)

[16] 肖奇学, 陈渝, 戚兰兰, 等. 堆分配大小可控的检测与分析[J]. 清华大学学报(自然科学版), 2015, 55(5): 572 – 578.
XIAO Qixue, CHEN Yu, QI Lanlan, et al. Detection and analysis of size controlled heap allocation[J]. Journal of Tsinghua University (Science and Technology), 2015, 55(5): 572 – 578. (in Chinese)