基于符号执行的 Return－to－dl－resolve 利用代码自动生成方法

方 皓 吴礼发 吴志勇
（陆军工程大学指挥控制工程学院 南京 210000）

摘 要 Return-to-dl-resolve是一种可突破复杂防护机制的通用漏洞利用技术，目前主要以手工方式实现，研究人员需要深入分析并理解ELF动态链接原理，泄露并解析任意库函数的地址，拼装攻击载荷，效率非常低。文中提出了一种基于符号执行的Return-to-dl-resolve自动化实现方法，该方法为ELF可执行文件提供符号执行环境，对程序崩溃点的符号状态进行约束，通过约束求解器对约束进行求解，实现了Return-to-dl-resolve利用代码自动生成系统R2dlAEG。实验结果表明，R2dlAEG可快速构造利用代码，并能够在NX和ASLR防护机制同时开启的条件下劫持程序的控制流。

关键词 符号执行，漏洞利用，安全防护机制，利用代码
中图法分类号 TP309 **文献标识码** A **DOI** 10.11896/j.issn.1002－137X.2019.02.020

Automatic Return－to－dl－resolve Exploit Generation Method Based on Symbolic Execution

FANG Hao WU Li-fa WU Zhi-yong
(Institute of Command and Control Engineering, Army Engineering University of PLA, Nanjing 210000, China)

Abstract Return-to-dl-resolve is a general exploit technology to bypass complicated protection mechanism, but the efficiency of manual shell-code' construction is very low. The thesis studies the core concept of ASLR, NX and Return-to-dl-resolve, and then set up a Return-to-dl-resolve model. The proposed model provides symbolic execution environment for ELF binary program, and generates exploit by constraint solving. It also inplements a control-flow hijacking exploit generation system named R2dlAEG. The experiment results show that R2dlAEG generates exploits in acceptable time, and the exploits can bypass both NX and ASLR.

Keywords Symbolic execution, Exploit, Security mechanism, Exploit code

软件漏洞利用是指利用软件本身的安全缺陷，实现通过软件的正常运行无法达到的目的，如执行代码、绕过认证和权限提升等[1]。传统方法主要是以手工方式构造利用样本并注入程序中实现对漏洞的利用。这一过程要求安全研究人员对系统底层知识及漏洞本身（二进制指令或代码）有非常透彻的理解和深入的分析。而如今软件功能越来越复杂，漏洞形式也越来越多样化，传统的分析方法难以应对上述挑战。因此，人们对高效的漏洞自动利用方法和技术有更加紧迫的需求。

现有漏洞自动利用方法主要有基于补丁比较的APEG[2]、基于控制流的AEG[3]、Mayhem[4]、Rex[5]、基于面向返回编程（Return Oriented Programming, ROP）[6]的Q[7]以及面向数据流的PolyAEG[8]方法等。但这些方法大多未考虑数据执行保护和地址随机化等因素带来的影响，其中，APEG，AEG和Mayhem考虑了简单条件下的漏洞利用模型，难以对抗复杂条件下的防护机制。Rex和Q虽然使用ROP技术来绕过数据执行保护机制（No-eXecute, NX）[9]与地址空间布局随机化（Address Space Layout Randomization, ASLR）[9]，但受程序中现有函数的限制，只能在少部分程序上获得成功，相当受限。Return-to-dl-resolve[6]是Federico等于2015年提出的一种可绕过NX与ASLR的漏洞利用技术，它利用ELF可执行文件的动态链接过程突破复杂防护机制。Return-to-dl-resolve的实现不需要泄露内存信息，且不受程序中现有的函数的限制。目前主要以手工方式编写Return-to-dl-resolve利用代码，效率低，同时对编写者的要求较高。

本文深入分析了Return-to-dl-resolve的原理，对Return-to-dl-resolve进行建模，提出了一种基于符号执行的Return-to-dl-resolve自动化实现方法，主要贡献如下。

1）深入分析了Return-to-dl-resolve的原理，提取了漏洞利用要素、Return-to-dl-resolve利用要素以及防护机制缓解绕过的相关要素，建立了Return-to-dl-resolve利用模型。这种利用模型也可以轻易扩展到其他的手工利用技巧上。

2）根据Return-to-dl-resolve利用模型，提出了一种基于符号执行的Return-to-dl-resolve自动化实现方法，利用本文设计的崩溃状态获取算法对程序的执行路径进行引导，可快速获取程序崩溃点的符号状态并提取运行时信息。

3）基于angr与Qemu设计并实现了漏洞利用自动生成系统R2dlAEG。该系统使用CTF试题及CVE漏洞程序进行测试，能够快速构建利用样本，且该利用样本能在开启NX

到稿日期：2018-01-24 返修日期：2018-03-01 本文受国家重点研发计划基金资助项目（2017YFB0802900）资助。
方 皓（1993－），男，硕士生，主要研究方向为网络空间安全，E-mail：cyyfh@qq.com；吴礼发（1968－），男，教授，博士生导师，主要研究方向为网络安全，E-mail：wulifa@vip.163.com（通信作者）；吴志勇（1982－），男，博士，副教授，主要研究方向为软件安全。

和 ASLR 防护机制的运行环境中被执行。

1 Return-to-dl-resolve 概述

Return-to-dl-resolve 通过 ELF 可执行文件动态链接机制实现对任意库函数的调用。它的基本思路是：利用动态链接器[10]和 ELF 格式的弱点，首先在程序中插入一个伪造的结构链表，然后引导动态链接器解析并执行任意库函数。动态链接可以有效提高应用程序的启动速度，Unix 类平台普遍应用了这种技术。

图 1 展示了 ELF 可执行文件的动态链接过程。其中，_dl_runtime_resolve()函数用于解析库函数名并返回库函数地址，.dynstr 节、.dynsym 节和.rel.plt 节是这一过程涉及的相关数据结构[11]。

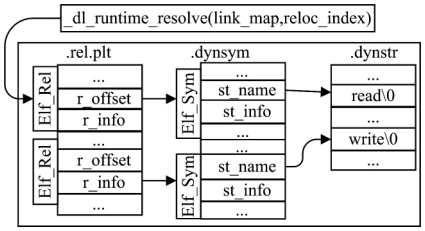

图 1 与库函数解析相关的数据结构[6]

Fig. 1 Data structure related to library function analysis

如图 1 所示，_dl_runtime_resolve()函数首先从.rel.plt 节得到 Elf_Rel 结构体，并根据其[r_info]字段的值获取 Elf_sym 结构体在.dynsym 节上的位置；接着，根据 Elf_sym 结构体中[st_name]字段的值获取相应的库函数名（如"read\0"）在.dynstr 节上的位置，最后根据得到的函数名计算并返回库函数在内存中的实际入口地址。之后，程序跳转到该地址开始执行，从而实现对库函数的调用。

根据 ELF 文件的动态链接过程，如果向程序插入一个伪造的结构链表，并引导_dl_runtime_resolve()函数从这个结构链表解库函数名，就可以实现对任意库函数的调用。

Return-to-dl-resolve 的实现方式如图 2 所示，图中左侧表示 ELF 可执行文件的动态链接过程，右侧表示伪造的结构链表及动态链接过程。

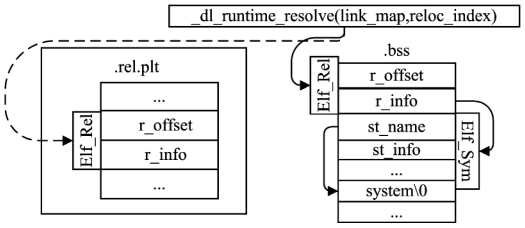

图 2 Return-to-dl-resolve 利用流程图[6]

Fig. 2 Principle of Return-to-dl-resolve's attack[6]

Return-to-dl-resolve 的实现分两个阶段。在第一个阶段，攻击者利用一个存在的漏洞在程序的可写区域构造一个结构链表[10]，其 Elf_Rel 结构的 r_info 域指向 Elf_Sym 结构，Elf_Sym 结构的 st_name 域指向的位置存放将要被解析的库函数名（如"system\0"）。在第二阶段，攻击者首先计算一个能够将_dl_runtime_resolve()函数导向至伪造的结构链表的 reloc_index 参数的值，并构造一个 ROP 链；然后将 reloc_index及 ROP 链的 gadgets 地址注入程序的内存，并调用 PLT0[10]代码；最后 PLT0[9]代码会调用_dl_runtime_resolve()函数，对 system 函数的函数名进行解析。

2 Return-to-dl-resolve 自动化实现原理

文中提出了一种基于符号执行的 Return-to-dl-resolve 自动化实现方法。该方法的主要思路为：首先给程序提供一个崩溃输入（即能造成程序崩溃的输入），监控并记录程序的执行路径；接着使用符号执行方法使程序按照上一步的执行路径模拟执行到崩溃点，获取崩溃点的符号状态，提取路径约束、可控 EIP 及可控内存等运行时信息；然后根据 Return-to-dl-resolve 的实现方式，对崩溃点的符号状态进行约束，综合路径约束、可控内存区域约束、针对 ASLR 和 NX 缓解绕过的约束条件和利用代码布局约束条件，使用约束求解器进行求解，并生成利用代码。

该方法具体分为 3 个步骤：崩溃状态的获取、可利用性判定、利用样本的构建，如图 3 所示。

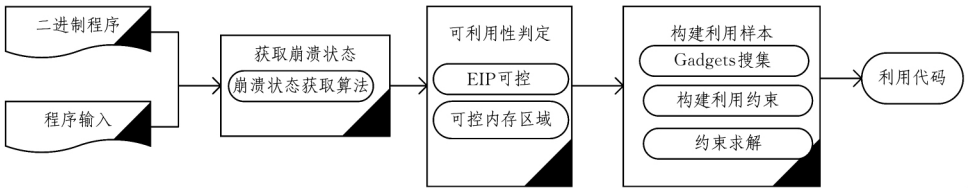

图 3 基于符号执行的 Return-to-dl-resolve 自动化实现流程图

Fig. 3 Automatic implementation flow of Return-to-dl-resolve attack based on symbolic exection

1）崩溃状态的获取：根据本文设计的崩溃状态获取算法，使程序沿崩溃输入产生的执行路径模拟执行到崩溃点，获取崩溃点的符号状态，提取运行时信息，包括路径约束、可控 EIP 和可控内存等。

2）可利用性判定：对程序当前崩溃点的可利用性进行判定，主要涉及程序的控制流是否能被劫持、可控内存区域是否足以插入伪造的结构链表等。

3）利用代码的构建：如果程序当前的崩溃点满足可利用性条件，则对崩溃点的符号状态进行约束，包括对符号化 EIP 和符号化内存进行约束，综合路径约束及程序符号状态的约束条件，使用约束求解器进行求解，构建利用代码。

2.1 获取崩溃状态

如果给程序一个崩溃输入，那么程序就会产生一条包含崩溃点的执行路径。本文自动化构建利用代码的基础是崩溃点的运行时信息，符号执行不能直接按照特定的执行路径分析程序，若直接进行动态分析，则需要实现搜索程序崩溃点的模块。而应用程序往往存在多条执行路径，因此有可能需要探索大量的执行路径，这使得时间开销增大，并且可能会因为

程序复杂出现路径爆炸的问题，分析效率较低。为此，本文设计了一种崩溃状态获取算法，使符号执行方法直接按照特定的执行路径分析程序，获取崩溃点的符号状态。只要记录了导致程序崩溃的执行路径，该方法便可单独分析这一条路径而无需探索程序的其他执行路径，从而快速获取相关的运行时信息。

本文基于 Qemu[12] 和 angr[4] 来实现获取程序的崩溃状态。程序的执行路径由 Qemu 记录，angr 将指令转化为中间语言 VEX，符号执行引擎根据 VEX 语句的语义模拟执行程序，修改程序状态并记录路径约束及可控的内存操作区域。崩溃状态获取算法描述了这一过程。其中，Tracer 表示具体执行引擎，S-Exec 表示符号执行引擎，trace 是存储执行路径相关地址的集合。

算法 1　崩溃状态获取算法

输入：二进制程序 binary，程序的缓冲区溢出数据 crash

输出：溢出状态 crash_state

```
1. CreateEnv()
2. trace=Tracer(binary)/*具体执行，获取执行记录*/
3. initialize(insn)
4. while insn<len(trace) do
5.     insn←update(tracet)
6.     if (isExit(trace(insn).op) ‖ isJmp(trace(insn).op) ‖
7.         is_Ret(trace(insn).op) then /*判断指令是否是跳转指令、
           返回指令或是否使程序结束*/
8.         insn←update(insn)
9.     else
10.        insn←update(tracet)
11.        block_trace.add(insn)
12.        insn←update(tracet)
13. proj=angr.proj(binary,crash)
14. se=S－Exec(proj)
15. break_addr=trace.getBreak()
16.    while(i<len(block_trace)－1)do
17.    addr←block_trace.getAddr()
18.    if addr !=break_addr then
19.        se.explore(addr)
20.        pathConstraint=se.pathConstrain.write( ) /*收集当前程
           序状态的路径约束*/
21.        if( !se.move(se.end[0],se.active[0]))then exit() /*路径
           探索出错，退出*/
22.        else se.move(se.end[0],se.active[0])
23.    else break
24. crash_prev_state=se.active[0]
25. crash_state=sm.step(crash_prev_state,state.insn_len－1)
26. return crash_state /*算法结束，返回崩溃状态*/
```

代码的第 2－12 行为程序的具体执行阶段，其中，第 2 行通过具体执行程序记录执行路径；第 4－12 行按基本块[14]划分记录下的执行路径，将识别出来的基本块首地址加入到基本块列表中。

第 13－25 行为程序的符号执行阶段，符号执行引擎将要执行的基本块逐个翻译成 VEX 语句以进行模拟执行。第 14－16 行为初始化阶段，其中，第 14 行用于为程序构建一个符号执行环境；第 16 行从基本块列表中取出崩溃点所在的基本块的首地址，记为 break_addr。第 17－24 行执行这样一个循环：每次从基本块列表中取出一条地址，记为 addr，第 19 行判断 addr 与 break_addr 是否相等，如果不相等，程序还未执行到崩溃点所在的基本块，第 20 行代码调用符号执行引擎模拟执行 addr 对应的基本块，若 addr 与 break_addr 相等，则程序下一次将要执行的基本块是崩溃点所在的基本块，循环结束。第 26 行调用符号执行引擎模拟执行崩溃点所在的基本块，获取崩溃点的符号状态。符号执行的整个过程监控程序输入在内存中的传播，第 21 行将其产生的路径约束记录到路径约束列表中。

2.2　可利用性判定

获取程序的崩溃状态后，对其可利用性进行判定。符号化 EIP 和符号化内存是一种有效的判定方法。

2.2.1　符号化寄存器和内存

符号执行使用一组符号化变量作为程序的输入，并且这组变量与程序输入的每个字节一一对应，输入在内存中的传播会被实时监控。符号执行过程中被符号化变量覆盖的寄存器和内存区域会分别被标记为相应的符号化寄存器和符号化内存。

当寄存器和内存被符号化变量覆盖后，通过对存储在寄存器和内存上的符号化变量限定约束条件即可控制寄存器和内存，改变程序的控制流及向内存插入数据。考虑一种实际情况，程序如图 4 所示。

```
1. #include<stdio.h>
2. #include<string>
3. void vuln()
4. {char buf[40];
5. read(0,buf,100)}
6. int main()
7. {printf("if your input'size>40,stack_smashing occurs !!");
8. vuln();
9. return 0;}
```

图 4　示例程序

Fig. 4　Example program

代码的第 3－7 行存在一个栈溢出漏洞，可能导致程序崩溃。我们给程序赋予一个 48 个字节的输入（包含 48 个“A”字符），当程序执行完 vuln 函数并返回 main 函数时，若返回地址是一个非法地址，则程序崩溃，第 7 行为崩溃点。图5(a)为第 6 行代码被执行前栈的布局，第 6 行代码被执行后栈的布局如图 5(b)所示。

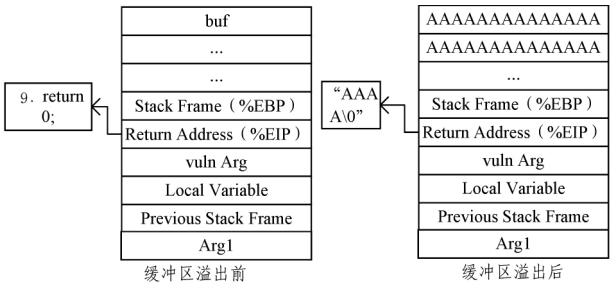

图 5　调用 read 函数前后栈的布局情况

Fig. 5　Change of stack memory after acallofread

使用崩溃状态获取算法将程序执行到崩溃点，EIP 和部

分内存区域被符号化变量覆盖，如果限定符号化 EIP 的每个字节与程序中某个函数入口地址的每个字节一一对应且相等，就可以劫持程序的控制流。同理，如果符号化内存也足够插入伪造的结构链，就尝试构建利用代码。因此，检测崩溃状态的 EIP 是否被符号化以及符号化内存是否足够布置相关的数据是一种简单、高效的判定方法。

2.2.2 漏洞可利用性

使用 Return-to-dl-resolve 技术对漏洞进行利用的条件是：1)程序的控制流能够被劫持；2)可控的内存区域足够插入伪造的结构链表。

将崩溃状态表示为 crash_state，其 EIP 寄存器记为 crash_state.EIP，若 EIP 是符号化的，则 Symbolic(crash_state.EIP)为 True，否则为 False；崩溃状态的符号化内存集合 SymbolicMem=[$SymMem_0$，$SymMem_1$，…，$SymMem_n$]，每块符号化内存的空间大小记为 len($SymMem_i$)($0\leqslant i\leqslant n$)；伪造的结构链表记为 Exploit-Data，其长度为 len(Exploit-Data)。程序的崩溃状态满足 Return-to-dl-resolve 的可利用条件，当且仅当满足：

$$Symbolic(crash\_state.EIP)\wedge(\exists SymMem_i(0\leqslant i\leqslant n\rightarrow(len(SymMem_i)\geqslant len(Exploit\text{-}Data)))=True$$

当符号化 EIP 和符号化内存满足定义时，根据定义可以推断 EIP 寄存器被符号化变量覆盖，且符号化内存也足够插入伪造的结构链。EIP 寄存器存储的是 CPU 下次要执行的指令的地址，如果对符号化 EIP 限定劫持控制流的约束条件，以及对符号化内存限定与存储伪造的结构链表相关的约束条件，就可以实现 Return-to-dl-resolve。

2.3 利用代码的构建

本文为符号化 EIP 和符号化内存限定约束条件，利用约束求解器对路径约束及约束条件求解得到利用代码，按顺序将两段利用代码发送给程序以实现 Return-to-dl-resolve。如图 6 所示，利用约束 1 向可控内存区域插入伪造的结构链表，利用约束 2 在利用约束 1 被实现的基础上使程序进入伪造的动态链接过程执行。

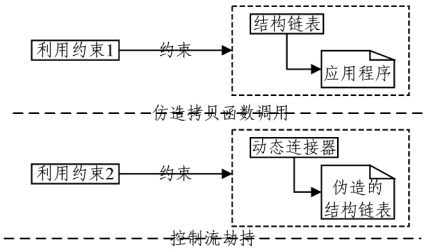

图 6 利用约束示意图

Fig. 6 Exploit constraint

1)利用约束 1：对于符号化 EIP，限定①：符号化 EIP 的每个字节与拷贝函数(如 strcpy 函数)地址的每个字节一一对应且相等。伪造这个函数调用所需的参数表示为 Arg=[arg_0，arg_1，…，arg_n]，存放 Arg 的符号化内存记为 Mem_1。限定②：Arg 的每个字节与 Mem_1 上的每个字节逐个对应且相等。伪造的结构链表表示为 Exploit-Data，存放 Exploit-Data 的符号化内存记为 Mem_2。限定③：Exploit-Data 的每个字节与 Mem_2 上的每个字节逐个对应且相等。

2)利用约束 2：与利用约束 1 类似，对于符号化 EIP，限定①：符号化 EIP 的每个字节与 _dl_runtime_resolve()函数入口地址的每个字节逐个对应且相等。伪造 _dl_runtime_resolve()函数调用所需的相关参数表示为 Arg=[link_map，reloc_index]，存放 Arg 的符号化内存记为 Mem_1。限定②：Arg 的每个字节与 Mem_1 上的每个字节逐个对应且相等。

当利用约束 1 产生的利用代码发送给程序后，程序的可控内存区域中已经被插入了伪造的结构链表，利用约束 2 基于这个基础伪造 _dl_runtime_resolve()函数调用。通过搜索二进制程序反编译的结果即可获取利用约束 1 要求的拷贝函数。

ASLR 防护机制的实现是将堆栈及共享库加载的基址随机化，即所有库函数地址都可能是随机的，而 Return-to-dl-resolve 活用了动态链接功能，通过动态连接器解析并调用库函数，ASLR 对这类攻击无效。NX 将数据所在内存页标记为不可执行，即使代码被植入数据区域也无法被执行。为了绕过 NX，我们使用一段 ROP 链来实现函数调用。具体来说，我们重用程序代码段里的一些指令片段，通过将这些指令片段的地址写在栈上并连续地执行返回指令使其被组合使用。

如算法 2 所示，利用代码生成算法描述了利用代码构建的过程。

算法 2 利用代码生成算法

输入：崩溃状态 crash_state，二进制程序 Binary

输出：利用代码 shellcode1，shellcode2

```
1. memCfunc←selfFindMemCpyFunciton()
2. ip←crash_state.EIP.value
3. crash_state.Constraint(ip=memCfunc.addr) /*对 EIP 的值进行约束*/
4. SymbolicMem←selfFindMemControled
5. for addr in SymbolicMem do
6.   if SymbolicMem[addr] >=memCfunc.arg then
7.     MemToPuts=selfMemGet(addr,len(memCfunc.arg))
8.     crash_state.Constraint(memCfunc.arg=MemToPuts) /*对可控栈区进行约束*/
9.     break
10. shellcode1-Exploit-Data(selfCreatExploit() /*构建结构链*/
11. memory←loadSymMem(len(shellcode1-Exploit-Data))
12. crash_state.Constraint(shellcode1-Exploit-Data=memory)
    /*将假结构链插入符号化内存*/
13. crash_state.useROP()
14. crash_state1←selfWindUpToNextState()
15. _dl_runtime_resolve←selfFindMemCpyFunciton()
16. ip←crash_state1.EIP.value
17. crash_state1.Constraint(ip=_dl_runtime_resolve.addr) /*使程序进入伪造的动态链接流程*/
18. SymbolicMem←selfFindMemControled
19. for addr in SymbolicMem do
20.   if SymbolicMem[addr] >=_dl_runtime_resolve.arg then
21.     MemToPuts=selfMemGet(addr,len(_dl_runtime_resolve.arg))
22.     crash_state.Constraint(_dl_runtime_resolve.arg=MemToPuts)
23.     Break
24. Hook(memAddr=r_offset,funcName=_dl_runtime_resolve)
25. crash_state1.useROP()
```

```
26. self. symbolicExec()
27. if crash_state. . EIP. value==self. getMmeory(addr=r_offset):
28.    shellcode(selfConstraintResult()
29. else
30.    RaiseError("Attack Failed !")
31.    Exit()
32. return shellcode.
```

代码的第 3－12 行为利用约束 1 构建的过程，其中，第 3，8，12 行分别实现利用约束 1 的限定①、限定②及限定③。第 10 行产生一个伪造的结构链表，伪造的结构链表中的数据根据_dl_runtime_resolve()函数的实现即可被自动化填充。第 13 行产生一个 ROP 链。第 14 行调用符号执行引擎，将伪造的结构链表、_dl_runtime_resolve()函数参数及 ROP 链的 gadgets 地址插入可控的内存区域，得到 $crash\_state_1$。同理，第 17－23 行为利用约束 2 的构建过程。根据_dl_runtime_resolve()函数实现，若_dl_runtime_resolve()函数成功解析了库函数，则它会将库函数的地址写入 r_offset 域指向的内存，并调用相应的库函数。第 24 行对 r_offset 指向的内存及_dl_runtime_resolve()函数进行 Hook 操作。第 27 行检查 EIP 寄存器的值与 r_offset 值索引的地址是否相等，若相等则说明_dl_runtime_resolve()函数从伪造的结构链表中解析了库函数名，攻击成功，第 28 行调用约束求解器，生成两段利用代码，并将其加入到利用代码列表中，算法结束；否则攻击失败，算法退出。

3　原型系统实现与实验

3.1　系统实现

基于上述研究，本文设计并实现了 Return-to-dl-resolve 利用代码自动生成系统 R2dlAEG，其框架如图 7 所示。

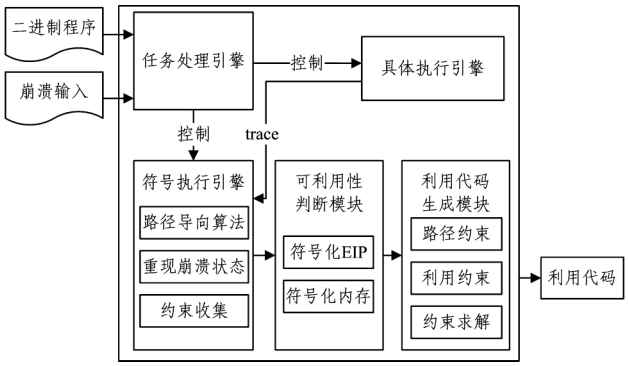

图 7　原型系统 R2dlAEG 的框架图

Fig. 7　Framework of R2dlAEG

其中，具体执行引擎基于 Qemu 实现，符号执行引擎基于 angr 实现，约束求解子模块基于 Z3[15] 实现。系统运行过程中，具体执行引擎负责监控程序的运行过程并记录执行路径，符号执行引擎通过上文提到的路径引导算法获取程序的崩溃状态，任务处理引擎监控具体执行引擎和符号执行引擎的运行状态。当符号执行引擎获取到崩溃状态后，可利用性判断模块对崩溃点的可利用性进行判断。若判定结果为真，则利用代码生成模块收集路径约束以及构建利用约束，约束求解子模块对约束求解，生成利用代码。

3.2　实验

1）实验环境

本实验采用处理器为 Intel Core I7-6770 3.4 GHz，运行内存为 8 GB，系统为 ubuntu16.04 64 位的主机。为了验证本文方法的有效性，将其与其他方法进行对比，主要选取了 Rex 和 Q 的测试集中的程序进行实验，测试程序的源码均使用 gcc 5.4.0 进行编译，构建利用代码所用的时间采用 ubuntu16.04 的 time 命令进行记录，并精确到小数点后一位。

2）实验结果

表 1 所列的前 5 个程序为 Rex 方法测试过的程序，中间 4 个程序为 Q 方法测试过的程序，最后 3 个程序为本次实验单独选取的测试程序。表中第 3 列表示程序中是否存在 system 函数，“－”表示当前方法未对该程序测试。为了对比本文方法与 Q、Rex 对抗 ASLR 和 NX 防护机制的能力，本次实验在仅开启 ASLR 和 NX 防护机制的条件下测试利用代码的有效性。若程序成功执行 system 函数，则判定利用样本有效，否则判为无效。

表 1　利用代码的有效性测试结果

Table 1　Eeffectiveness validation of shellcode

程序	来源	链接 system 函数	调用 system 函数		
			R2dlAEG	Rex	Q
baby-re	DefconCTF 2016	√	√	√	－
crackme2000	DefconCTF 2017	√	√	√	－
angry-reverser	Hackcon CTF 2016	×	√	×	－
crypto400	Whitehat CTF 2015	√	√	√	－
zwiebel	TUMCTF 2016	×	√	×	－
opendchub	CVE-2010-1147	×	√	－	×
gv	CVE-2006-6563	√	√	－	√
proftpd	CVE-2006-6563	√	√	－	√
rsync	CVE-2004-2093	√	√	－	√
Aeon	CVE-2005-1019	×	√	－	－
dnsmapq	CVE-2017-14493	×	√	－	－
PInfo	EDB-ID-40023	×	√	－	－

表 2 记录了本文方法构建利用代码所用的时间，表 2 的第 1 行表示测试程序，第 2 行表示构建利用代码所耗费的时间。

表 2　构建利用代码的时间记录

Table 2　Time recording of shellcode' generation

程序	baby-re	crackme2000	angry-reverser	crypto400	zwiebel	opendchub	gv	proftpd	rsync	Aeon	dnsmapq	PInfo
耗时/s	20.0	43.0	160.3	27.0	6.3	47.3	67.1	55.3	67.0	13.5	27.1	26.9

3）实验结果分析

从表 1 的结果可以看出，R2dlAEG 针对 12 个测试程序全部生成了利用代码，并且都能在开启 ASLR 与 NX 防护机制的运行环境中被执行，证明了本文所提方法的有效性。表 2 则显示了本文方法可以快速地构建利用样本，提升分析实际漏洞的效率。

Rex 方法和 Q 方法构建利用样本依赖于程序中的函数，因此缺乏有效对抗 ASLR 与 NX 防护机制的能力。R2dlAEG

通过 Return-to-dl-resolve 能够不依赖程序中的函数构建利用样本，绕过 ASLR 与 NX 防护机制。实验结果表明，R2dlAEG 不仅能够有效对抗复杂防护机制，且能够在可接受的时间内生成利用代码。

4 相关工作

APEG 由 Brumley 等于 2008 年提出。它的主要流程是：首先使用二进制程序差异比较工具(如 EBDS)比较并分析补丁程序与原程序，识别补丁程序对用户输入添加的安全检查条件的位置。之后利用符号执行技术自动生成违反安全验证的输入，最后利用污点分析方法进一步找出能造成原程序崩溃的输入作为利用样本。APEG 构建的利用样本大多只能造成程序的崩溃，而 R2dlAEG 主要的关注点是劫持程序的控制流，并且不依赖于程序补丁。

为了摆脱对补丁程序的依赖，以及构造能够劫持控制流的利用样本，Avgerinos 等提出了 AEG，其核心思路是：利用预处理符号执行和指令插桩技术，自动化地实现从生成劫持程序控制流的利用样本再到判断样本有效性的过程。AEG 分析程序需要基于程序的源代码，R2dlAEG 分析程序则不依赖程序的源码。

Mayhem 系统利用符号执行与具体执行相结合的方法，实现了高效的利用样本自动化生成的方法。该方法的主要流程是：首先寻找以条件跳转指令(如 jump eax 和 call ebx 指令)结尾的指令片段，并将其标记为候选的漏洞利用点；然后将被标记的指令片段转化为 IL 中间语言，利用符号执行技术生成利用样本，最后由具体执行子系统对利用样本的有效性进行判断。

与 Mayhem 系统不同的是，Rex 系统降低了具体执行方法与符号执行方法的耦合度。它的核心思想是：首先给定一个导致程序崩溃的 crash，将其交由具体执行子系统执行程序，获取符号化内存、符号化 EIP，并构建利用样本。与 Mayhem 和 Rex 相比，Mayhem 关注的是寻找程序中潜在的漏洞利用点，R2dlAEG 则根据崩溃点追踪算法得到漏洞利用点，比 Mayhem 更加高效；Rex 只对系统的部分库函数进行建模，导致其难以分析实际的漏洞程序，R2dlAEG 能够处理更多的漏洞程序，且能够在程序没有与 system 函数链接的情况下调用 system 函数。

Q 是一种 ROP 代码自动生成方案。虽然它利用 ROP 技术可以绕过 NX 与 ASLR，但与 R2dlAEG 相比，它依然无法在程序没有与 system 函数链接的情况下劫持控制流，仅能分析少数漏洞程序。

结束语 本文提出了一种基于符号执行的 Return-to-dl-resolve 自动化实现方法，并将其用于自动化地构建能绕过 ASLR 与 NX 防护机制的利用代码。与其他方法相比，该方法构建的利用样本可以不依赖程序中的函数，并且可以快速获取程序的运行时信息，为后续程序分析提供帮助。下一步工作包括研究如何绕过 stackcanary 防护机制，以及增加利用模型，如 UAF 攻击模型等，这将使得该方法具备分析更多实际漏洞程序的能力。

参 考 文 献

[1] LIU J, SU P R, YANG M, et al. Software and Cyber Security—A Survey[J]. Journal of Software, 2017, 28(7): 42-68. (in Chinese)
刘剑，苏普睿，杨珉，等. 软件与网络安全研究综述[J]. 软件学报，2017，28(7)：42-68.

[2] BRUMLEY D, POOSANKAM P, SONG D, et al. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications[C]// IEEE Symposium on Security & Privacy. 2008.

[3] AVGERINOS T, SANG K C, HAO B L T, et al. AEG: Automatic Exploit Generation [J]. Internet Society, 2011, 57(2): 74-84.

[4] SANG K C, AVGERINOS T, REBERT A, et al. Unleashing Mayhem on Binary Code [C] // Security and Privacy. IEEE, 2012: 380-394.

[5] STEPHENS N, GROSEN J, SALLS C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]// Network and Distributed System Security Symposium. 2016.

[6] FEDERICO A D, CAMA A, YAN S, et al. How the ELF ruined Christmas[C] // Usenix Conference on Security Symposium. USENIX Association, 2015: 643-658.

[7] SCHWARTZ E J, AVGERINOS T, BRUMLEY D. Q: exploit hardening made easy [C] // Usenix Conference on Security. USENIX Association, 2011: 25.

[8] WANG M, SU P, LI Q, et al. Automatic Polymorphic Exploit Generation for Software Vulnerabilities[M]// Security and Privacy in Communication Networks. Springer International Publishing, 2013: 216-233.

[9] 王清，张东辉，周浩. Oday 安全：软件漏洞分析技术[M]. 北京：电子工业出版社，2011.

[10] 俞甲子. 程序员的自我修养[M]. 北京：电子工业出版社，2009：90-132.

[11] ORACLE. SYMBOLS[EB/OL]. [2017-12-27]. https://docs.oracle.com/cd/E26926_01/html/E25910/chapter6-79797.html.

[12] BARTHOLOMEW D. QEMU: a multihost, multitarget emulator[M]. Belltown Media, 2006.

[13] 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.

[14] SHEN L, DAI K, WANG Z Y. The Non-Sequential Instruction Prefetching Based on Basic Blocks[J]. Computer Engineering & Science, 2003, 25(4): 94-98. (in Chinese)
沈立，戴葵，王志英. 以基本块为单位的非顺序指令预取[J]. 计算机工程与科学，2003，25(4)：94-98.

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