计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(10):44−52 [doi: 10.15888/j.cnki.csa.005991]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于符号执行的自动利用生成系统①

万云鹏[1,2], 邓　艺[2], 石东辉[3], 程　亮[2], 张　阳[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 软件研究所, 北京 100190)
[3](深圳大学 深圳南特商学院, 深圳 518060)

摘　要: 在本文中, 我们提出 BAEG, 一个自动寻找二进制程序漏洞利用的系统. BAEG 为发现的每一个漏洞产生一个控制流劫持的利用, 因此保证了它所发现的漏洞都是安全相关并且可利用的. BAEG 针对输入造成程序崩溃的情况进行分析, 面临的挑战主要有两点: 1) 如何重现崩溃路径, 获取崩溃状态; 2) 如何自动生成控制流劫持利用. 对于第一点, 本论文提出路径导向算法, 将崩溃输入作为符号值, 重现崩溃路径. 对于第二点, 我们总结多种控制流劫持的利用原理, 建立对应的利用产生模型. 此外, 对于非法符号读、写操作, BAEG 还可以让程序从崩溃点继续执行, 探索程序深层次代码, 检测崩溃路径逻辑深处是否还有利用点.

关键词: 自动利用生成; 符号执行; 路径追踪; 符号内存

引用格式: 万云鹏,邓艺,石东辉,程亮,张阳.基于符号执行的自动利用生成系统.计算机系统应用,2017,26(10):44–52. http://www.c-s-a.org.cn/1003-3254/5991.html

Automatic Exploit Generation System Based on Symbolic Execution

WAN Yun-Peng[1,2], DENG Yi[2], SHI Dong-Hui[3], CHENG Liang[2], ZHANG Yang[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[3](Shenzhen Audencia Business School, Shenzhen University, Shenzhen 518060, China)

Abstract: In this paper we present BAEG, a system to automatically look for exploitable bugs in the binary program. Every bug reported by BAEG is accompanied by the control flow hijacking exploit. The working exploits ensure robustness that each bug report is security-critical and exploitable. Giving BAEG a vulnerable program and an input crash, the challenges are: 1) how to replay crash and get the state of crash; 2) how to automatically generate exploit. For the first challenge, we present a path-guided algorithm, take crash input as symbolic data, and replay crash path. For the second challenge, we summarize the principles of multiple control-flow hijack and establish the corresponding exploit generation model. Besides, BAEG can explore deep code especially for invalid symbolic read and symbolic write, which can help us decide whether there still are exploits at deeper code.

Key words: automatic exploit generation; symbolic execution; path tracing; symbolic memory

1　引言

软件漏洞挖掘一直是安全领域的研究热点之一. 尽管利用模糊测试技术一定程度上解决了漏洞的自动发现问题, 但攻击者和防御者更关心的是这些程序漏洞或缺陷能否被利用[1,2]. 传统程序漏洞利用生成方式主要是人工构造的方式, 这个过程需要安全研究人员具备较为全面的计算机知识 (例如汇编代码, 操作系统原理, 系统架构等) 和安全背景知识 (例如漏洞形成原理). 但如今软件的规模越来越大, 也越来越复杂, 漏洞形式也越来越多样化, 传统程序利用方式已很难应对

① 基金项目: 国家自然科学基金 (61471344); 国家 242 信息安全计划 (2016A086)
收稿时间: 2017-01-22; 采用时间: 2017-02-15

上述挑战.

自动利用生成技术是发现并验证程序漏洞的重要而且有效的手段. 自动利用生成技术为开发者决定先修复哪些程序漏洞提供了具体、可行的信息[3]. 给定任意一个程序, 自动利用生成技术面临的两大挑战是如何自动寻找程序缺陷和如何自动生成利用. 国内外的安全研究人员一直致力于开发可以自动发现并验证程序漏洞的方法. 现有方法主要是面向控制流的 AEG[3]、MAYHEM[4]以及 heelan[2]提出的控制流劫持方法; 基于补丁比较的 APEG[5]方法和面向数据流的 FlowStitch[6]方法. 这些方法大多都是直接对程序路径进行探索, 难以解决路径爆炸的问题. 本论文以程序崩溃作为利用生成的切入点, 造成程序崩溃的原因往往是因为触发了程序中存在的错误, 那么这个崩溃执行路径上是否有可利用点, 或者说这个程序错误是否是安全相关的, 很值得我们研究.

为了高效的找到程序中可利用的漏洞, 本系统的设计原则主要基于以下 4 点: 1) 系统在程序崩溃的情况下进行分析, 判断崩溃的可利用性. 2) 系统不应该做重复的工作, 之前的分析结果可以再次使用. 3) 系统应该能够识别出用户输入控制的内存. 4) 能够在程序崩溃处继续执行, 探索程序深层次代码, 检查崩溃路径逻辑代码深处是否还有可利用点. 目前已有的自动利用生成方法中, 例如 AEG, Mayhem 等都不能完全满足上述原则.

在本论文中, 我们提出 BAEG(Binary-based automatic exploit generation), 一个寻找二进制程序可利用漏洞的系统. BAEG 为它报告的每一个漏洞产生一个控制流劫持的利用, 这样, 保证了它所报告的漏洞都是可利用的. 本论文的主要贡献有以下几点:

(1) 不依赖于源码分析. 现在大多数软件厂商出于利益或知识产权的原因不再提供源代码; 更重要的是, 由于编译器会自动对源程序进行优化, 可能在编译、链接过程中产生漏洞, 面向源程序的漏洞检测技术无法分析和发现这类漏洞.

(2) 间接控制流检测. 对 EIP 是否符号化的监测是针对不同类型的控制流劫持漏洞一个较全面的、简单的方法, 对于大多数情况下是可行的的. 但是某些情况下, 攻击者可能不是直接修改 EIP, 而是通过篡改程序其他数据来改变 EIP, 例如修改程序全局偏移表 GOT 的值. 本论文除了对 EIP 做直接检查外, 还对间接修改 EIP 的情况做了检测, 主要表现在符号读写的情况.

(3) 探索程序深层次代码. 大多数软件发现的漏洞是比较浅显的, 有时很难执行到程序的深层次代码. 本论文通过将造成崩溃的非法操作修改为合法操作, 使程序继续执行, 探索程序逻辑深层次的代码, 检查崩溃路径逻辑代码深处是否还有利用点.

(4) 实用性和效率. 本论文测试了 11 款有漏洞的开源软件, 并且成功生成利用, 其中 8 个来自 AEG 和 MAYHEM 的测试集, 3 个来自 CVE 和 EDB 漏洞库. 实验证明, 本论文提出的方法具有实用性, 并且在效率方面普遍优于 AEG 和 MAYHEM.

2 系统概述

本论文实现基于符号执行[7]的自动利用生成方法, 以二进制程序为研究目标, 通过将符号执行和自动利用生成技术相结合, 设计并实现了一种基于符号执行的自动利用生成方法. 本方法的输入为一个含有漏洞的二进制程序和一个可以造成该二进制程序崩溃的字符串输入, 本论文的方法具体原理及思路为: 利用我们设计的二进制自动利用生成工具 (BAEG) 加载执行目标程序, 通过崩溃输入驱动目标程序运行. 在目标程序执行过程中, 目标程序的输入被当成是符号变量, 通过 BAEG 对目标程序的二进制代码进行实时的跟踪和分析, 重现崩溃路径, 获取崩溃状态, 收集运行时信息, 包括寄存器值和内存快照等; 再进一步判定崩溃的可利用性, 主要针对控制流劫持做出判定; 如果可利用性为真, 尝试产生一个控制流劫持的字符串, 产生的利用字符串可以使这个有漏洞的程序执行非法跳转, 实施攻击.

本系统分为三个模块, 运行时信息收集模块、可利用性判定模块和利用生成模块. 如图 1 所示.

运行时信息收集模块: 根据程序的崩溃输入, 驱使程序运行, 设计并实现路径导向算法, 重放崩溃路径, 收集程序崩溃状态下的运行时信息, 包括寄存器值、内存快照和堆栈使用情况等, 获取崩态.

可利用性判定模块: 根据收集的运行时信息, 可对崩溃类型的利用性做出判定. 本系统实现了对 EIP、EBP 符号化的判定, 符号写、符号读操作造成崩溃时可利用性的判定, 以及符号格式化字符串含有潜在利用可能的判定. 具体判定原理和依据见 3.2 节.

利用生成模块: 一旦判定崩溃的可利用性为真, 我

们就尝试产生一个利用输入, 这个输入中包含我们精心构造的一段 shellcode. 产生的利用输入直接使程序运行, 执行我们插入的 shellcode, 实现控制流劫持. 其中涉及约束信息的搜集, shellcode 插入位置查找, 以及约束求解. 具体构造过程见 3.3 节.

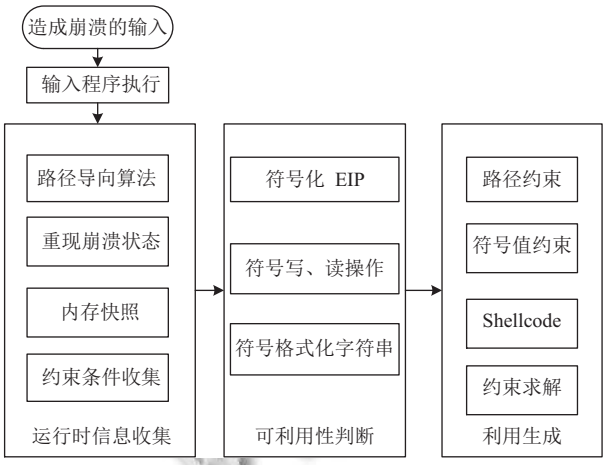

图 1 系统框架图

3 系统设计与实现

我们建立一个崩溃模型来分析崩溃路径, 其中包括符号输入、崩溃点的内存快照, 寄存器值等和到达崩溃点的约束信息. 假设给定程序一个崩溃输入, 我们通过跟踪这个输入的执行路径, 如果这个输入导致程序崩溃, 那么由崩溃输入执行的路径就很有可能被利用. 相比探索程序的所有路径, 这种做法很高效.

我们基于 angr[8]和 qemu[9]来实现符号输入, 路径导向, 崩溃点信息搜集和路径约束与求解. qemu 是一个 pc 仿真器, 可以获得输入为具体值时的基本路径信息和捕获程序信号量. angr 是一个二进制分析平台, 能够像 mayhem、klee[10]一样执行动态符号执行, angr 支持多体系架构 (x86, arrch64, mips, amd64, arm 等) 和 ipython, 并且具有高封装性、开放和可扩展性等优点.

首先, 我们利用 qemu 获取输入为具体值时程序各个基本块的地址, 此时我们知道了程序在各个分支的具体走向, 但是 qemu 的具体执行却无法为我们提供路径约束、用户输入控制的符号内存等重要信息, 这些信息对利用生成是至关重要的. 符号化方法却能满足我们的需求. 接下来, 我们基于 angr, 将输入值视为符号值, 根据之前得到的基本块地址信息动态导向符号执行路径, 重现崩溃路径, 同时收集运行时信息.

3.1 路径导向算法

在路径导向过程中, 主要涉及两个部分: 路径追踪和分支获取 (见算法 1 和算法 2). 目的是重现崩溃路径, 获取程序崩溃时的状态, 得到路径约束信息、内存快照等信息.

算法 1 描述了如何对当前路径进行追踪. Branches 表示分支集合, 类型属于 path_group[8]. path_group 是 angr 的主要接口, 是多条路径执行一次后的路径集合, 表示的是将来发生的状态, 其属性 active 表示接下来的合法分支集合, 属性 deadened 表示到达末尾的分支集合. bb_cnt 表示当前处理的基本块编号, 从 0 开始, trace_sum 表示 qemu 执行程序时得到的基本块的总数, trace 是一个地址集合, 记录了崩溃输入执行时所经过的各个基本块的地址. 我们先使用 qemu 动态执行程序和崩溃输入, 得到基本块起始地址集合 trace, 正如之前提到的, qemu 无法为我们提供程序执行过程中更加详细的信息, 也无法满足我们利用生成模型建立的需求. 所以, 我们第二次执行程序时, 将输入当成是符号值, 在分支处选择与 qemu 相同的执行路径, 裁剪不合法分支.

算法1. 路径追踪

```
输入: constrained_addrs: 被约束的地址, 并且不应该被移除
输出: prev_path: 崩溃输入所执行的路径
      crash_state: 程序崩溃时的状态
1 branches←empty /*初始化*/
2 while(branches is empty or branches.active is not empty) and bb_cnt < trace_sum do
3     branches←getNextBranches() /*获取分支*/
4     if reachTheEnd() then /*达到路径末尾*/
5         bp←prev_path.state.inspectBP() /*插入断点, 记录地址约束信息*/
6         prev_path.step() /*前向执行*/
7         p_block←basicBlockOfprev_path()
8         succes←prev_pathStepForward(lengthOf-InstNumsOfp_block - 1) /*到达崩溃前一条指令*/
9         crash_state←prev_pathNextRunSuccessors[0]
10        return prev_path, crash_state
11 all_path←branches.active + branches.deadended
12 return all_path[0], None
```

刚开始时, 程序分支集合为空, 当未到达路径末尾并且分支为空或分支集合中含有有效分支时, 执行这样一个循环: 获取接下来要执行的分支, 这个部分包含分支选择和裁剪, 由算法 2 完成. 然后判断是否达到路径末尾 (崩溃点所在基本块); 若到达路径末尾, 那么插

入断点, 记录约束信息, 让程序继续执行到崩溃点前一条指令, 然后获取崩溃状态, 算法结束时, 返回崩溃执行路径和崩溃状态; 若没有到达路径末尾, 重复上述过程. 如果在到达路径末尾前退出循环, 那么返回当前执行路径.

算法 2 主要描述了如何获取接下来需要执行的分支, 为了和 qemu 的执行情况一致 (也就是动态导向崩溃输入的执行路径), 需要进行一系列的判断和裁剪多余分支 (符号路径) 的动作. path_group 是多条路径的集合. 算法在获取到当前的路径 curPath 时, 更新待处理的基本块 bb_cnt 值, 我们的目的是让当前路径的地址 current_addr 和 qemu 获得的基本块地址 trace[bb_cnt] 相同, 实现对符号路径的导向, 整个处理过程涉及判断是否到达路径末端 (deadended)、判断当前具体执行路径和符号执行路径是否一致、处理库函数和系统调用, 以及处理函数被挂起这五种情况, 经过这些处理之后, 保证当前路径集合中只有一条合法路径.

算法2. 分支获取

```
输入: trace: 基本块地址集合
      path_group: 初始化分支集合
输出: path_group: 接下来要执行的分支的集合
1 while size(path_group.active) == 1 do
2     curPath←path_group.active[0]/*获取当前需要处理的路径*/
3     bb_cnt←updateBb_cnt(bb_cnt) /*更新bb_cnt, 指向下一个需要处理的基本块*/
4     if bb_cnt >= trace_sum then /*达到路径末尾*/
5         return path_group
6     if curPath == trace[bb_cnt] then
7         bb_cnt←bb_cnt + 1
8     else if isSysCallHappen() then
9         pass /*发生系统调用*/
10    else if curPathJumpkindIsIjk_Sys then
11        bb_cnt←bb_cnt + 1
12    else if isLibraryCallorSimproceders() then
13        if current_addIsNotRecordInR_plt() then
14            bb_cnt←bb_cnt + 2
15    else if isCurPathHooked() then
16        while current_addr ≠ trace[bb_cnt] and bb_cnt < trace_sum then
17            bb_cnt←bb_cnt + 1 /*处理函数被挂起的情况*/
18    else
19        TracerMisfollowError
20    bb_max_bytes←getBlockMaxSize(trace, bb_cnt)
21    path_group←path_group.step(size= bb_max_bytes)
22    if bb_cnt >= trace_sum then
23        tpg←path_group.step()
24        if crash_mode is True then
25            tpg←tpg.stash (from='acitve', to='crashed')
26            return tpg
27        else
28            if tpg.active is empty then
29                path_group←tpg
30                return path_group
31    path_group←prunePathGroup(path_group)
32    return path_group
```

算法执行过程中, 处理顺序执行、库函数调用、模拟程序调用、函数被挂起和多个分支的情况, 然后程序向前执行, 同时记录相应的运行时信息和约束信息, 将获得的新的分支集合反馈给算法 1.

当算法 1 和算法 2 执行完毕之后, 我们得到崩溃路径和崩溃状态, 同时也完成了对崩溃路径运行时信息的收集. 如果崩溃状态获取失败, 返回执行路径.

3.2 可利用性判定

3.2.1 符号化指令指针

EIP 寄存器存放着下一条将被执行的指令的地址, 控制 EIP 寄存器的值可以说是控制流劫持的最后一步. 因此, 在程序执行过程中, EIP 寄存器的状态监测是针对不同类型的控制流劫持漏洞一个较全面的、简单的方法. 如图 2 所示, 当检测到 EIP 被用户输入数据覆盖时, 尝试生成利用.

对 EIP 和 EBP 的检查分为两种, 一种是全为符号值, 另外一种是部分符号值. 部分符号值的情况是指 EIP 部分字节被符号化, 例如 EIP 的值为 0x12345678, 当缓冲区只有一个字节溢出时, 低字节 0x78 被符号值篡改, 变为符号值, 但高字节还是具体值.

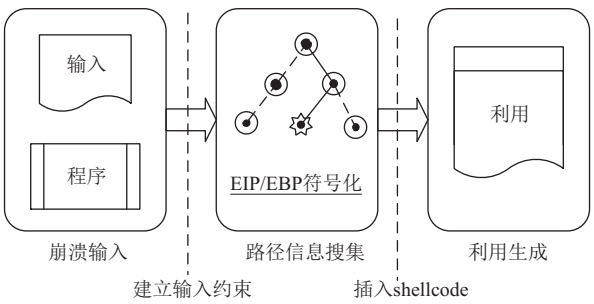

图 2 EIP 被修改, 利用生成图

3.2.2 符号写操作

除了修改 EIP 的值之外, 被损坏的指针可以间接的修改控制流. 尤其是当符号数据写入符号地址的时候意味着可以将任意数据写入任意地址. 当一个符号

写入动作被发生时 (被写地址也为符号值), 攻击者可将写操作的目标重定向到敏感数据, 如返回地址和 GOT 入口地址等, 相当于间接更新了 EIP 寄存器.

考虑实际可能发生的一种情况, 如图 3 所示, 一个 off-by-one 溢出将会破坏 EBP 的最低有效位 (LSB), 即使这个漏洞不会直接修改返回地址, 但是符号指针可以间接的污染 EIP 并且对程序进行控制流劫持.

```
1. #include <stdio.h>
2. int cpy(char *input)
3. {
4.     char buf[256];
5.     strcpy(buf,input);
6. }
7. int main(int argc, char *argv[])
8. {
9.     if(strlen(argv[1]) > 256){
10.        printf("bufferoverflowattempt!!!\n");
11.        return 1;
12.    }
13.    cpy(argv[1]);
14. }
```

图 3 off-by-one 示例程序

当我们输入 256 个由字符'A'构成的字符串时, 产生程序栈溢出. 图 4 左图为程序调用 strcpy 之前栈帧空间的情况. 当我们调用 strcpy 之后, EBP 的值被部分污染, 程序不再正常执行, 此时栈帧情况如图 4 右图所示. 由于输入刚好多出一位 null(0x0), 所以 EBP 的最低一个字节被修改, 在程序执行完之后, 不会弹出指向父函数的正确的帧指针 (frame pointer), 而是被我们修改过的帧指针. 我们修改后的帧指针会使我们跳转到程序输入 buf 中, 然后我们可以在 buf 中重现之前的栈帧, 修改调用者栈帧中的局部变量、EBP 和返回地址. 因此, 在被调函数返回时, 我们可以写入任意的返回地址, 实现控制流劫持.

除了符号数据写入符号地址外, 还有一种崩溃类型是具体数据写入符号地址, 符号写入动作可以让我们进一步对程序进行探索, 以探索程序深层次代码, 观察崩溃路径逻辑深处是否还有可利用点. 这里主要设计了针对具体数据写入符号地址情况下, 对程序继续执行, 此时, 我们需要将待写入的符号地址指向某段可写内存区域. 算法 3 描述了寻找可写内存段的过程, 其中, violating_action 表示崩溃发生时的动作, 此处为符号写入动作.

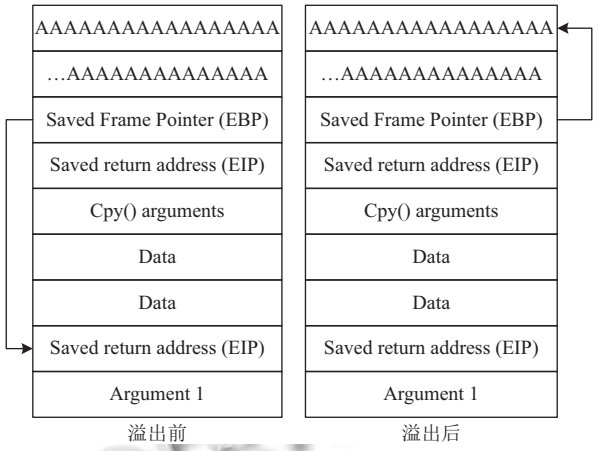

图 4 调用 strcpy 前后栈帧情况

算法3. 探索任意写

输入: crash_state: 程序崩溃状态

输出: new_state: 构建在崩溃点后新的执行状态

```
1 min_write←min(violating_action.addr) /*非法动作的最小符号地址*/
2 max_write←max(violating_action.addr)
3 segs←allWritableSegmentsOfElfObjects()
4 segs←filterBetween(min_addr, max_addr) /*非法动作可以覆盖的段*/
5 wirte_addr←empty 6 constraint←empty
6 foreach seg in segs do
7       foreach page in range(seg.min_addr, seg.max_addr, 0x1000) do
8           write_addr = page
9           constraint = violating_action.addr == page
10           if crash_state.satisfiable( violating_action.addr == page) then
11               break
12           constraint←empty
13 if constraint is empty then
14       CannotExploitError
15 crash_state.add_constraint(constraint)
16 new_state←createNewCrashObjectStartingHere(crash_state)
17 return new_state
```

在发生符号写操作之前, 我们通过查找程序各段可能的写入地址, 我们将造成崩溃的写入地址约束为我们找到的合法写入地址, 并将此约束传给约束求解器, 判断选择的写入地址是否可行. 一旦找到某个可行的写入地址, 我们就将造成崩溃的写操作的写入地址指向该可行地址, 进行深层次代码的探索.

3.2.3 符号读操作

与符号写操作相对应的是符号读操作. 程序崩溃时的符号动作可能为读操作, 这也是一种比较特殊的漏洞类型, 我们可以在此基础上继续让程序执行.

我们将导致程序崩溃的无效读取地址, 改为指向某个有效内存区域, 这个有效内存是由用户输入控制的, 或者是在程序有效范围内. 算法 4 描述了将无效读取地址定位到有效内存的过程, violating_action 表示程序崩溃时的执行动作, 此处为读操作.

算法4. 探索任意读

输入: symbolic_mem: 用户输入控制的符号内存, 是以地址为键, 长度为值的表
输出: new_state: 构建在崩溃点后新的执行状态

```
largest_regions←sorted(symblic_mem) /*符号内存从大到小排序*/
min_read←crash_state.se.min(violating_action.addr) /*非法操作的最小符号地址*/
max_read←crash_state.se.max(violating_action.addr) /*非法操作的最大符号地址*/
largest_regions←filter(larget_regions, min_read, max_read) /*过滤掉不可读的地址*/
min_addr←get_min_addr() /*程序的逻辑起始地址*/
max_add←get_max_addr()
pages←range(min_addr, max_addr, 0x1000) /*分页*/
pages←filter(pages, min_addr, max_addr) /*过滤掉不可读的页*/
/*现在, 我们得到了合法的读取区域*/
foreach addr in lagrset_regions + pages do
    constraint←violating_action.addr == addr
    if crash_state.se.satisfiable(extra_constraints=constraint) then
/*是否有解*/
        break
    constraint←empty
if constraint is empty then
    UnableToFindSuitableReadAdressWarning
crash_state.add_constraints(constraint)
new_state←createNewCrashObjectStartingHere(crash_state)
return new_state
```

与符号写操作的处理相似, 将不合法的读取地址指向合法的读取地址, 不同的是, 程序的读取范围分为用户输入控制的内存和程序本身控制的内存两个部分. 用户输入控制的内存在输入被符号化的时候被标记, 程序控制的内存为整个二进制程序的逻辑地址空间. 通过对这两个内存部分进行查找, 一旦确定某个合法的读取地址, 记录读取约束 (地址约束), 从崩溃点继续执行, 探索深层次的代码. 此外, 通过符号读操作获取的数据可以视为一种伪符号数据, 虽然我们不能改变读取数据的值, 但是我们可以选择读取哪一块内存.

3.2.4 符号格式化字符串操作

格式化字符串漏洞是一种常见的漏洞, 通常由格式化函数的误用产生, 例如: printf 或者 syslog 等. 攻击者可以通过注入精心构造的一段攻击字符串给格式化函数, 达到破坏程序或任意数据重写任意地址的目的.

典型的例子是拒绝服务攻击. 当程序请求一个无效的内存地址时, 就会导致程序终止. 比如代码中存在 printf(userName) 时, 攻击者可以插入一段格式化字符串, 让程序显示内存的值, 又或者让程序读取一个非法地址的内容, 导致程序崩溃等等. 例如攻击者输入一段字符串“%s%s%s%s%s%s%s%s%s%s%s”, 这段被攻击者构造的字符串使得 printf(userName) 在被执行时变为 printf(“%s%s%s%s%s%s%s%s%s%s%s%s”), 其中每个 %s 会使 printf() 从栈中弹出一个数作为地址, 然后打印出该地址指向的内容, 直到遇到一个空字符 (NULL 或 0). 当 printf() 遇到一个非法地址时, 就会导致程序崩溃.

格式化字符串的利用模型比较难以建立, 本系统暂时实现了定位被用户控制的格式化函数 printf. 首先在对崩溃路径的追踪过程中, 观察 printf 函数是否被调用, 然后, 判断 printf 中的格式化字符串是否被用户污染. 通过观察二进制程序反汇编的结果, 我们观察到寄存器 RSI 保存着指向 printf 要打印的字符串的地址, 取得该字符串后, 只需进一步判断用户输入的字符串是否包含于该字符串即可.

3.3 利用生成

3.3.1 符号内存查找

为了在内存空间中找到连续的符号数据, 我们需要检查被用户输入数据控制的内存区域. 这些被控制的内存区域可能是分开的, 也可能是连续的, 因为在某些分支条件判断时, 会约束输入的某一个或一些字节为具体值. 我们需要找到一串足够长的符号内存区域, 用来存放我们的 shellcode, 该 shellcode 可以是使程序跳转执行 shell 命令. 算法 5 描述了寻找符号内存空间起始地址及其大小.

算法5. 寻找内存区域

输入: user_writes: 用户输入, 已符号化
输出: segments: 地址和大小为键值对的集合

```
segments←{} /*初始化*/
user_writes←sorted(user_writes)
if size(user_writes) == 0 then
    return segments
cur_start = user_writes[0]
cur_end = cur_start + 1
foreach write in user_writes then
    write_start←write
    write_len←1
```

```
10      if cur_start > cur_end then
11          segments[cur_start]←cur_end - cur_start
12          cur_start←write_start
13          cur_end←write_start + write_len
14      else
15          cur_end←max(cur_end, write_start + write_len)
16 segments[cur_start] = cur_end - cur_start
17 return segments
```

3.3.2 注入 shellcode

在上一步得到符号内存后, 我们需要确定哪些符号内存可以存放我们的 shellcode. 对于可用符号内存, 我们都构造这样一个约束集合: 限定符号内存上的每个字节和 shellcode 上的每个字节一一对应且相等. 下一步, 将 shellcode 的约束集合和路径约束一起传递给约束求解器 clarify[8]进行求解, 算法 6 描述了这个过程.

算法6. 注入shellcode

```
输入: symbolic_mem: 用户输入控制的符号内存
      crash_state: 程序崩溃状态
输出: addr: 注入的shellcode的起始地址
1 shellcode←selfDefinedShellcode()
2 min_addr←get_min_addr() /*程序逻辑起始地址*/
3 max_addr←get_max_addr()
4 foreach addr in symbolic_mem do
5 if addr >= min_addr and addr < max_addr then
6       sc_bvv←BitVector(shellcode) /*构造shellcode的符号值*/
7       memory←load(addr, len(shellcode)) /*将shellcode写入符号内存*/
8       if crash_state.satisfiable(memory == sc_bvv, crash_state.ip == addr) then /*约束求解*/
9           crash_state.add_constraint(memory == sc_bvv) /**/
10          crash_state.add_constraint(crash_state.ip == addr) /*eip指向shellcode起始位置*/
11          writeExploitInputToFile()
12          break
```

最终, shellcode 的起始位置, 以及 EIP 指向的位置会被确定下来. 约束求解器会求解路径约束来生成利用.

3.3.3 其他利用类型

1) 返回库函数: 返回库函数攻击一般应用于缓冲区溢出中, 其堆栈中的返回地址被替换为另一条指令的地址, 并且堆栈的一部分被覆盖以提供其参数. 这允许攻击者调用现有函数而无需注入恶意代码到程序中. 这种攻击方式可以绕过程序的 W⊕X 保护, 因为运行时的库函数总是可以被执行的.

本系统中, 我们先寻找到 system 函数, 然后将“/bin/sh”插入到符号内存中, 仿造一个函数调用, 让目标程序跳转执行 system(“/bin/sh”), 实现控制流劫持.

2) 跳转到寄存器: 地址空间布局随机化 (ASLR) 使得返回库函数攻击变得几乎不可能, 因为所有函数的内存地址都是随机的. 使用 NOP 填充也许可以绕过 ASLR, 但这种方法并不总是可行的. 我们将 shellcode 的地址存放在寄存器中, 通过跳转到寄存器来实现我们的利用生成. 比如 ESP 寄存器, 当一个函数返回时, 返回地址会被弹出栈, ESP 寄存器将指向下一个返回地址之后的栈空间. 我们可以将 shellcode 插入到返回地址之后, 并且使用 ESP 作为跳板. 如果存在指令“jmp %esp”, 那么就可以产生一个绕过 ASLR 的利用. “jmp %esp”的二进制编码为 0xffe4.

本系统将 0xffe4 插入到符号内存处, 如果插入失败, 就尝试寻找现有的“jmp %esp”, 然后将 eip 指向指令“jmp %esp”, 并且在 esp 所指的栈上插入 shellcode, 当约束求解器求解路径约束、符号值约束和地址约束有解时, 产生利用.

4 实验结果和分析

接下来的部分描述了我们的实验结果. 我们首先介绍实验环境, 然后介绍我们的实验结果以及分析.

1) 实验环境. 本论文的实验环境所用主机处理器为 Intel Core I7-4770 3.4 GHz, 2 GB 内存, 系统为 ubuntu 16.04 64 位. 测试程序使用 gcc 5.4.0 编译. 本实验所用的测试用例都是没有修改过的开源软件, 全部可以从网上下载. 时间测量方法采用 linux 上的 time 命令, 并且保留到小数点后一位.

2) 实验具体结果. 我们总共测试了 11 款有漏洞的开源软件, 本论文所使用的测试用例主要来自 CVE (common vulnerabilities and exposures)、OSVDB (open source vulnerability database) 和 EDB(exploit-DB). 表 1 总结了我们的实验结果. 第一列和第二列表示软件名称和对应的版本; 第三列是漏洞类型; 第四列是产生利用所用时间; 第五列表示这个漏洞软件的出处.

表 1 的安排如下: 前 5 个是来自 AEG 的测试集; 中间 3 个来自 MAYHEM 的测试集; 最后 3 个是我们额外添加的几个测试软件, 这些测试软件不曾出现在 AEG 和 MAYHEM 中.

表 1 BAEG 生成利用表

程序	版本	漏洞类型	耗时(秒)	出处
iwconfig	26	Stack overflow	1.8	CVE-2003-0947
nCompress	4.2.4	Stack overflow	9.8	CVE-2001-1413
Htget	0.93	Stack overflow	10.4	N/A
Glftpd	1.24	Stack overflow	3.6	CVE-2004-1418
exim	4.41	Stack overflow	10.2	EDB-ID-796
Htpasswd	1.3.31	Stack overflow	3.4	OSVDB-ID-1637
PSUtils	1.17	Stack overflow	42.3	EDB-ID-890
Tipxd	1.1.1	Format string	N/A	OSVDB-ID-12346
Gif2png	2.5.3	Stack overflow	3.2	CVE-2009-5018
Hsolink	1.0.118	Stack overflow	4.8	CVE-2010-2930
Unrar	3.9.3	Stack overflow	16.5	EDB-ID-17611

其中 htget 是 AEG 发现的 zero-day 漏洞, tipxd 是发现被用户控制的格式化字符串, 这里没有产生利用, 只是报出这个潜在的威胁. 本论文除了测试 AEG 和 MAYHEM 都测试过的软件外, 还测试了 gif2png、hsolink 和 unrar, 并且成功生成利用, 这是 AEG 和 MAYHEM 都没有做的.

表 2 显示了 BAEG 和 AEG 以及 MAYHEM 在成功生成同一程序利用上时间的对比. 值得注意的是, 表中的横线表示当前软件并未被工具测试. 例如 AEG 并未测试 htpasswd.

表 2 利用生成时间对比表

程序	AEG耗时(分)	MAYHEM耗时(秒)	BAEG耗时(秒)
iwconfig	1.5	2	1.8
nCompress	12.3	11	9.8
Htget	57.2	7	10.4
Glftpd	2.3	4	2.6
exim	33.8	–	10.2
Htpasswd	–	4	4.3
PSUtils	–	46	42.3

图 5 是一个比较直观的体现, 对于没有被测试的软件, 我们这里不做对比. 例如, MAYHEM 没有测试 exim, 那么 exim 在耗时上的对比就只有 AEG 和 BAEG.

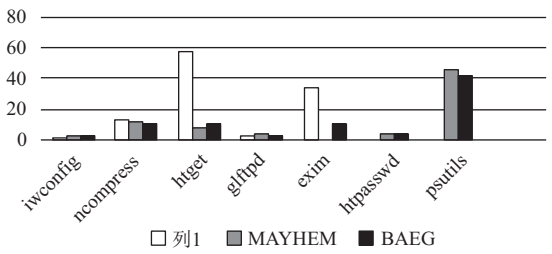

图 5 MAYHEM 和 BAEG 时间对比图

3) 实验结果分析. 通过表 2 和图 5 的对比可以看出, MAYHEM 和 AEG 能处理的大多数漏洞, 我们也能处理, 说明本系统 BAEG 具有实用性. 此外, BAEG 在利用产生的耗时上普遍快于 AEG; 小部分慢于 MAYHEM, 这是由于 MAYHEM 的对输入的预处理模块, 使得 MAYHEM 在部分软件的测试上耗时较少. 总体来说, 我们的系统 BAEG 除了实用性外, 效率上也是比较可观的.

5 相关工作

D.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[5]. APEG 通过查找补丁程序中添加的过滤条件确定程序缺陷的位置, 同时构造不满足过滤条件的“违规”输入来产生利用. APEG 的局限性体现在两个方面: 首先, 该方法无法处理补丁程序中不添加过滤判断的情况; 其次, 从实际利用效果来看, APEG 构造的利用类型大多属于拒绝服务攻击, 无法造成直接的控制流劫持. BAEG 主要是产生控制流劫持利用, 并且不依赖程序补丁.

Heelan[2]首次提出基于二进制的控制流劫持利用自动生成方法. 该方法利用符号执行和污点传播[11]来产生触发漏洞的约束条件, 判断利用的主要依据是 EIP 的值是否被污点数据感染. Heelan 的方法和我们的工作类似, 都是以崩溃输入作为实验前提, 但是我们除了判断 EIP 被符号化外, 还判断了符号读和符号写情况下的利用情况. 此外, BAEG 对崩溃路径深层次代码进行探索, 观察是否存在潜在利用.

T.Avgerinos 等人首次提出了一种有效的漏洞自动挖掘和利用产生方法 AEG[3], 缺点是利用产生过程依赖源码和受限于编译器和运行时环境影响. BAEG 基于二进制程序进行分析, 适用范围更加广泛.

S.K.Cha 等人提出基于二进制程序的漏洞利用自动生成方法 Mayhem[4], 是 AEG 方法的一个扩展. 不足之处在于只实现了部分系统和库函数的建模, 此外, 具有漏报和误报等问题. BAEG 没有误报的情况, 它所报告的每一个利用都具有可重放性.

6 结语

在本论文中, 我们基于 angr 和 qemu 实现了一个二进制程序漏洞利用生成系统 BAEG. 为了能够产生控制流劫持, 我们观察分析符号化指令指针的情况, 此外, 对于能够间接产生控制流劫持的攻击, 例如符号写

和符号读操作, 我们也作出分析. 我们提出并完成路径导向算法, 基于程序崩溃建立利用生成模型. 我们完成了利用输入的自动构造, 将 shellcode 插入到用户输入中, 使程序跳转执行我们的 shellcode, 实现控制流劫持. 对于 BAEG 找到的利用漏洞, 都伴随着对应的利用输入, 经过实验结果分析, 我们相信 BAEG 对于崩溃情况下利用漏洞的查找是可行的.

参考文献

1 Miller C, Caballero J, Johnson NM, *et al*. Crash Analysis with BitBlaze. USA: BlackHat, 2010.

2 Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities[Ph. D. thesis]. Oxford: University of Oxford, 2009.

3 Avgerinos T, Cha SK, Lim BTH, *et al*. AEG: Automatic exploit generation. Proc. of the Network and Distributed System Security Symposium. Reston, VA, USA. 2011. 283–300.

4 Cha SK, Avgerinos T, Rebert A, *et al*. Unleashing mayhem on binary code. Proc. of the 2012 IEEE Symposium on Security and Privacy. San Francisco, CA, USA. 2012.

5 Brumley D, Poosankam P, Song D, *et al*. Automatic patch-based exploit generation is possible: Techniques and implications. Proc. of the IEEE Symposium on Security and Privacy. Oakland, CA, USA. 2008.

6 Hu H, Chua ZL, Adrian S, *et al*. Automatic generation of data-oriented exploits. Proc. of the 24th USENIX Conference on Security Symposium. Washington, D.C, USA. 2015.

7 King J C. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394. [doi: 10.1145/360248.360252]

8 Shoshitaishvili Y, Wang RY, Salls C, *et al*. SOK: (State of) the art of war: Offensive techniques in binary analysis. Proc. of 2016 IEEE Symposium on Security and Privacy. San Jose, CA, USA. 2016. 138–157.

9 Bellard F. QEMU, a fast and portable dynamic translator. Proc. of the Annual Conference on USENIX Annual Technical Conference. Anaheim, CA, USA. 2005. 41–46.

10 Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California, USA. 2008. 209–224.

11 Kang MG, Mccamant S, Poosankam P. DTA++: Dynamic taint analysis with targeted control-flow propagation. Proc. of the 18th Annual Network and Distributed System Security Symposium. San Diego, California, USA. 2011.