基于符号执行的缓冲区溢出漏洞自动化利用

李 超 胡建伟 崔艳鹏
(西安电子科技大学网络与信息安全学院 陕西 西安 710071)

摘 要 软件规模与数量的快速增长给软件安全研究带来了严峻的挑战,以人工方式分析漏洞已难以完成漏洞危害性的评估。分析缓冲区溢出漏洞的形成原理,提出一种缓冲区溢出漏洞自动化利用方法。该方法采用符号执行检测漏洞,为缓解符号执行中状态爆炸问题,使用危险函数切片减少状态数量。对于检测到的漏洞,通过构建约束表达式和约束求解自动生成 exploit。针对进程中不存在空间足够的可控内存块的情况,以 shellcode 分段存放的方式利用漏洞。实验结果表明,该方法可有效缓解符号执行路径爆炸问题,自动检测漏洞并生成适用性较好的 exploit。

关键词 缓冲区溢出 符号执行 程序切片 自动化利用

中图分类号 TP311 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2019.09.056

AUTOMATIC EXPLOIT GENERATION FOR BUFFER OVERFLOW BASED ON SYMBOLIC EXECUTION

Li Chao Hu Jianwei Cui Yanpeng
(*School of Cyber Engineering, Xidian University, Xi'an 710071, Shaanxi, China*)

Abstract The rapid growth of the software scale and quantity has brought severe challenges to software security research. It is difficult to perform vulnerability assessment by analyzing the vulnerability manually. This paper analyzed the principle of buffer overflow and proposed an automatic exploit generation method. The method used symbolic execution to detect vulnerabilities. To alleviate the state explosion of symbolic execution, we pruned the states according to the slice of unsafe function calls. For detected vulnerabilities, exploit was automatically generated by constructing constraint expressions and constraint solving. In the case that there was no enough controllable memory block in the process, we segmented shellcode to construct the exploit. The experimental results show that the method can detect vulnerabilities and generate exploit with good applicability automatically.

Keywords Buffer overflow Symbolic execution Program slicing Automatic exploit generation

0 引 言

信息技术的快速发展使得计算机软件在社会活动与工业生产中起着越来越重要的作用。同时,软件规模与数量的快速增长给信息安全带来了严峻的挑战。信息系统中存在软件漏洞是导致信息安全问题的重要原因。软件漏洞通常指软件系统在设计、实现、配置、运行等过程中,由操作实体有意或无意产生的缺陷、瑕疵或错误,它们以不同形式存在于信息系统的各个层次与环节中。为确保信息系统的安全,众多研究人员对漏洞分析与防护问题进行了大量的研究工作。然而,由于冯·诺依曼计算机体系自身的缺陷,以及当前软件系统的代码规模和技术复杂度的急剧提升,并且在软件生命周期的每个阶段都需要人工参与,难免会引入一些错误,导致无法彻底清除软件中存在的漏洞。

在不能完全杜绝漏洞存在的情况下,需对其进行分析与研究,以最小化漏洞所带来的损失。由于大多数商业软件都不公开源码,并且二进制代码是软件的最终表现形式,分析二进制代码可以更加全面和直接地找到软件中存在的漏洞,因此,二进制漏洞分析技术更具有实用性。文献[1]的数据显示,2017 年新增漏

收稿日期:2019-01-20。**李超**,硕士生,主研领域:软件安全。**胡建伟**,副教授。**崔艳鹏**,副教授。

洞中,缓冲区溢出漏洞为数量最多的漏洞类型,占新增漏洞总量的 18.06%,远远高于其他漏洞类型。因此,研究缓冲区溢出漏洞具有重要意义。

面对各类信息系统中存在的大量漏洞,CNNVD[2]等组织对漏洞进行统一的分类管理,评估漏洞的危害性并将其标记为不同危害等级,指导软件厂商采取相应的修复措施,从而减少漏洞带来的威胁和损失。通常可利用的软件漏洞具有很高的危害性,攻击者往往通过这些漏洞控制目标系统。因此,在漏洞响应过程中,需要快速甄别大量软件错误中的可利用漏洞。虽然使用模糊测试技术发现软件错误具有很好的效果,但由于漏洞类型的多样性和漏洞形成机理的复杂性,漏洞可利用性的评估和利用数据的构造通常需进行动态调试分析漏洞形成的细节,这个过程由分析人员以手工方式完成,并且要求分析人员熟悉汇编语言等计算机底层原理。随着软硬件产品和应用的快速增长,漏洞数量急剧攀升,2017 年共发布漏洞信息 13 417 条,漏洞数量达到 2016 年的近 2 倍[1]。因此,传统漏洞分析方式已难以应对上述挑战。

为提高软件漏洞风险评估的效率,本文研究缓冲区溢出漏洞,提出一种面向二进制程序的自动化漏洞利用方法,通过构建 exploit 证明漏洞的危害性。该方法首先使用符号执行检测漏洞,然后构建路径约束表达式和利用约束表达式,最后通过约束求解器求解得到 exploit。

1 相关工作

目前已有学者对自动化漏洞利用进行了研究,并取得了一定的进展。由 Brumley 等[3]提出的 APEG 基于补丁比对的方法定位程序中已修补的漏洞,通过分析补丁中添加的过滤条件,构造不满足过滤条件的输入触发漏洞。该方法无法适用于补丁中没有添加过滤条件的情况,并且构造的输入只能进行拒绝服务攻击。相对于 APEG 对补丁的分析,Avgerinos 等[4]提出了基于源码的漏洞自动挖掘和利用方法 AEG,AEG 使用预置条件的符号执行找到程序漏洞,利用动态二进制插桩获取程序运行时信息,构建约束表达式,并求解得到可实现控制流劫持攻击的利用数据。

为了能够在无法获取程序源码的情况下自动构造利用数据,Heelan[5]提出了基于二进制程序的漏洞自动利用方法。该方法以可触发漏洞的样例作为输入,通过代码插桩定位到漏洞,并使用污点分析找到可用于存放攻击代码的可控内存,构建生成利用数据所需的约束表达式,最后求解得到控制流劫持攻击利用数据。Cha 等[6]提出的漏洞自动利用生成方法 Mayhem 使用混合符号执行技术,分析过程中符号执行引擎在离线符号执行与在线符号执行间不断切换,以减少内存消耗,缓解状态爆炸问题。此外,该方法使用基于索引的内存模型优化符号化内存的加载提高系统效率。Wang 等[7]提出自动化生成多样性漏洞利用的方法 PolyAEG,该方法以崩溃样例为输入,通过动态污点分析获得程序执行的相关信息,构建污点传播流图和全局污点状态记录获取程序中所有可能被控制的劫持点、跳板指令和内存区域,最后利用不同的跳转指令和可控制内存区域构造多样性的利用样本。Huang 等[8]提出的 CRAX 同样以崩溃样例为输入对程序进行全系统模拟的符号执行分析,分析过程中对漏洞利用不相关的库函数或内核函数进行具体执行,以优化符号执行,提高处理速度。该方法可适用于 Microsoft office word 等规模较大的应用程序。

相对于上述面向控制流的利用方法,Hu 等[9]提出了面向数据流的自动利用方法 FlowStitch,利用内存错误修改程序数据流中的关键变量,可达到敏感信息泄露或提权的攻击效果。该方法可实现敏感信息泄露,因此实用性较强。其缺点是需要能触发内存错误的输入。由于堆管理机制的复杂性,导致堆漏洞利用的难度相对较大,Revery[10]对堆漏洞自动化利用问题进行了探索,在 19 个测试程序中可成功对 9 个程序生成利用。此外,NAVEX[11]对 Web 应用漏洞自动构造利用数据,可成功利用 SQL 注入和 XSS 漏洞,该方法与二进制漏洞利用有较大的差别。

综上所述,APEG 和 AEG 分别依赖于补丁和源码检测漏洞;Mayhem 使用符号执行检测漏洞,采用具体化部分符号变量的方法减少搜索空间,但可能导致漏洞不可利用;文献[5]和文献[7-9]均依赖于已知的崩溃输入,无法自动检测程序中存在的漏洞。此外,上述方法未考虑进程中不存在空间足以容纳 shellcode 的可控内存块的情况,构造利用的适用性较差。本文所提方法使用符号执行检测漏洞,通过切片减少状态数量,并改进漏洞利用时 shellcode 存放方式,可提高系统适用性。

2 缓冲区溢出漏洞自动化利用

缓冲区溢出漏洞产生的原因是程序未正确检查用户输入数据的长度是否超过目标缓冲区的大小,向缓冲区写入过多数据覆盖了内存中其他数据,可能导致控制流劫持。通过缓冲区溢出劫持控制流的常见方法包括覆盖栈中函数返回地址和覆盖函数指针。利用代

码注入或代码复用[12]可实现执行任意代码。代码注入将一段攻击代码写入进程空间,之后劫持控制流到攻击代码执行;代码复用将内存中已有的代码片段拼接成可实现特定功能的攻击链进行攻击。本文主要研究代码注入攻击的自动化。

本文基于二进制分析框架 angr[13]设计并实现缓冲区溢出漏洞自动利用原型系统 AutoExp(Automatic Exploitation),该系统以漏洞程序为输入,使用符号执行[14]检测漏洞,通过构建约束表达式和约束求解生成 exploit。以 exploit 作为程序输入可触发漏洞,并利用漏洞达到获取系统控制权、运行任意代码或窃取数据等目的。

如图 1 所示,自动化生成 exploit 包括 4 个步骤:1)预处理。为了减小漏洞检测过程中符号执行的状态空间,首先扫描目标程序中危险函数调用位置,然后通过程序切片技术获取危险函数调用位置到程序入口点的代码切片。2)漏洞检测。针对上一步得到的切片进行符号执行,记录每个状态的路径约束、寄存器和符号内存信息。同时,每运行一步均检测是否存在包含漏洞的状态。3)构建利用约束。找到漏洞后,判断漏洞的可利用性,通过构建 shellcode 约束将可控内存区域的值约束为 shellcode 以实现攻击代码注入,构建 EIP 约束将 EIP 寄存器的值约束为 shellcode 存放地址以实现控制流劫持。4)约束求解。使用约束求解器求解路径约束和利用约束,若有解则成功生成 exploit。

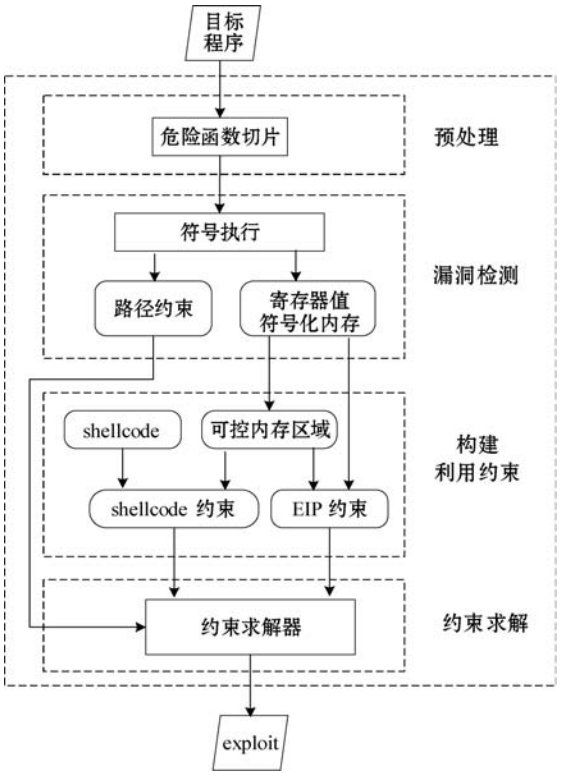

图 1　漏洞自动利用系统设计

2.1　基于符号执行的漏洞检测

漏洞自动化利用的前提条件为找到程序中存在的漏洞,本文采用符号执行检测漏洞。符号执行以符号变量代替具体值作为程序输入,并动态模拟执行程序中的指令,在执行过程中记录寄存器和内存状态。当遇到分支语句时,复制程序状态以便继续分析所有分支,并构建路径约束表达式记录到达不同分支的路径信息。符号执行过程中,根据不同漏洞模型设置违例断言可检测程序中存在的漏洞。

由于符号执行分析过程中每一个分支语句都可能导致新增一条路径,所以路径数量可能按指数级别增长,即存在状态爆炸问题。为了缓解状态爆炸问题,并且使分析过程更具有针对性,本文提出基于危险函数切片的方法获取包含危险函数调用的程序切片,符号执行时根据切片剔除无关路径。

2.1.1　预处理

控制流劫持漏洞的利用主要关注漏洞脆弱点和控制流劫持点[15],漏洞脆弱点指导致漏洞产生的函数或指令,而控制流劫持点指程序控制流被输入数据控制的指令。缓冲区溢出漏洞多是由于程序中使用了危险函数,并且未对用户输入数据进行严格的检查所导致的。因此,缓冲区溢出漏洞的脆弱点往往为危险函数调用位置。常见危险函数如表 1 所示。

表 1　危险函数列表

函数功能	函数名称
获取输入	gets、scanf、fscanf、vscanf、vfscanf、vfwscanf、vwscanf
内存拷贝	strcpy、stpcpy、wcscpy、memcpy、wmemcpy
其他	sprintf、getwd、realpath、strcat、wcscat、wcxscat

预处理过程如算法 1 所示,首先通过静态分析获取程序中的脆弱点位置。具体方法为,根据预先定义的危险函数名列表 unsafe_func_name 查找程序链接表 PLT(Procedure Linkage Table)得到危险函数地址 unsafe_func_addr;根据地址查找控制流图 CFG(Control Flow Graph)得到所有危险函数节点 unsafe_nodes,获取危险函数节点的前驱节点即可得到危险函数调用点地址列表 unsafe_callsites。接着对危险函数调用点进行程序切片[16],得到从程序入口点到危险函数调用点的切片。具体方法为,分析程序数据依赖关系和控制依赖关系构建数据依赖图 ddg(Data Dependence Graph,DDG)和控制依赖图 cdg(Control Dependence Graph,CDG),根据 ddg 和 cdg 使用轻量级污点分析[17]从危险函数调用点 target 进行后向切片得到切片

bk_slice。

算法 1 预处理

输入:目标程序 program, 危险函数名列表 unsafe_func_name

输出:程序切片 bk_slice

```
plt = get_plt(program)                    // 获取程序 plt 信息
cfg = create_cfg(program)                 // 构建程序 CFG
/* 获取每个危险函数的调用点 */
for fname in unsafe_func_name:
    unsafe_func_addr = plt[fname]
                  // 根据 CFG 得到程序中所有危险函数节点
    unsafe_nodes = cfg.get_all_nodes(unsafe_func_addr)
  // 获取危险函数节点的前驱节点得到危险函数调用点
    for node in unsafe_nodes:
        unsafe_callsites.append(node.predecessors.addr)
/* 根据危险函数调用点进行切片 */
ddg = create_ddg(cfg)                     // 构建 DDG
cdg = create_cdg(cfg)                     // 构建 CDG
for target in unsafe_callsites:
bs = create_backward_slice(cfg, ddg, cdg, target) //切片
    bk_slice.append({'target': addr, 'slice': bs})
return bk_slice
```

2.1.2 漏洞检测

符号执行引擎[13]在模拟运行程序时,以状态(state)表示程序的执行过程,其中记录了程序的执行路径和内存、寄存器等运行时信息;使用模拟管理器(SimulationManager)控制符号执行过程,可管理不同类型的状态和使用搜索策略探索程序状态空间。SimulationManager 通过 stash 管理 active、found、unconstrained 等不同类型的状态,active state 为当前正执行的状态,found state 为通过设定探索目标所找到的状态 unconstrained state 为不受约束的状态。

符号执行以符号值替换用户输入,如果程序中存在缓冲区溢出漏洞,当程序运行到漏洞劫持点时 EIP 寄存器将被符号化,由于符号化变量不是具体值,符号执行引擎不能确定下一步需执行的指令,导致无法继续运行,此时状态类型为 unconstrained。因此,通过判断符号执行过程中是否存在 unconstrained 状态即可检测缓冲区溢出漏洞。

算法 2 描述了漏洞检测方法。符号执行过程中,在程序入口点与脆弱点间运行时根据预处理得到的切片进行状态修剪,剔除切片之外的路径,以减少状态数量。具体方法为,符号化用户输入并创建模拟管理器 simgr,接着获取切片 bk_slice 中脆弱点地址(即危险函数调用点)target 作为符号执行的探索目标。同时设定状态修剪策略函数 drop_states_not_in_slice,该函数判断 active stash 中的状态是否在切片范围内,若是则返回 False,即保留该状态;否则返回 True,丢弃该状态。当找到脆弱点状态后,丢弃 active 中所有状态,并把脆弱点状态从 found stash 移动到 active stash 以便从脆弱点继续运行;找到脆弱点之后继续执行,当 unconstrained stash 非空时则表明存在控制流劫持点,即找到漏洞状态 vul_state。

算法 2 漏洞检测

输入:目标程序 program, 切片 bk_slice

输出:漏洞状态 vul_state

```
for slice in bk_slice:
/*程序入口点到脆弱点间运行时根据切片进行状态修剪*/
sym_input = symbolic(input)          // 符号化用户输入
init_state = entry_state(program, sym_input)
                                     //创建初始状态
simgr = simulation_manager(init_state)
                                     // 创建模拟管理器
target = slice['target']             // 获取脆弱点地址
// 以脆弱点为目标进行符号执行,并设定状态修剪策略
simgr.explore(find = target, filter = drop_states_not_in_slice)
// 若找到脆弱点状态,则使 active stash 中只包含该状态
if simgr.found not NULL:
    simgr.drop(stash = 'active')
    simgr.move(from_stash = "found", to_stash = "active")
/* 从脆弱点继续探索,直到找到 unconstrained 状态 */
while simgr.unconstrained is NULL:
    simgr.step()                     // 向前执行一步
vul_state = simgr.unconstrained
return vul_state
```

2.2 基于约束求解的利用生成

2.2.1 利用约束构建

进程空间中存在可控内存块是进行代码注入攻击的必要条件。进程中可控内存区域并非都是连续的,为了找到能存放 shellcode 的可控内存块,需获取可控内存块信息,包括内存块的起始地址和大小。获取可控内存块信息的方法如算法 3 所示,首先获取漏洞状态 vul_state 中符号化内存地址列表 sym_addrs;然后根据地址是否连续来统计内存块的大小 size,并记录内存起始地址 buf_start;最后将内存块按空间从大到小的顺序排序。

算法 3 获取可控内存块信息

输入:漏洞状态 vul_state

输出:符号化内存块 sym_bufs // 获取符号化内存地址列表

```
sym_addrs = find_symbolic_addr(vul_state)
while sym_addrs not NULL:
    size = 0                         // 设定内存块初始大小
```

```
buf_start = sym_addrs[0]  // 记录内存块起始地址
/* 统计连续内存地址组成的内存块大小 */
while True:
    if not buf_start + size in sym_addrs:
        break
    sym_addrs.remove(buf_start + size)
                                    //删除已处理地址
    size += 1
sym_bufs.append({'addr': buf_start, 'size': size})
sorted_by_size(sym_bufs)    // 根据内存块大小排序
return sym_bufs
```

当进程空间中不存在足以容纳 shellcode 的可控内存块时,现有方法将无法成功构建 exploit。如图 2 所示,为提高漏洞自动利用系统的适用性,AutoExp 把 shellcode 分段存放在多个可控内存块,并使用跳转指令连接不同内存块中的攻击代码,从而完成攻击过程。

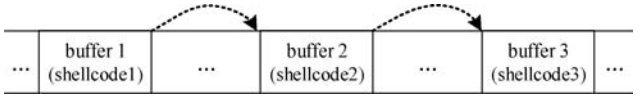

图 2　shellcode 分段存放

对 shellcode 分段时应确保指令的完整性,本文将机器码形式的 shellcode 反汇编为汇编指令,分段时以指令为基本单位。算法 4 具体描述了分段的方法,首先反汇编 shellcode 为汇编指令 asm,根据可控内存块信息与 shellcode 大小确定分段数量和每个片段的长度 segs_len;接着根据片段长度对 shellcode 进行分段;最后在除末尾片段外的所有片段后添加跳转指令 jmp_ins。

算法 4　shellcode 分段

输入:符号化内存块 sym_bufs, 攻击代码 shellcode

输出:shellcode 片段 sc_segments

```
asm = disassemble(shellcode)        // 反汇编 shellcode
/* 确定每个片段的长度 */
length = 0, n = 0
                    // 初始化片段长度 length 和内存块序号 n
for ins in asm:
// 若当前内存块还能容纳指令 ins,则划分在该内存块
    if length + ins.size <= sym_bufs[n].size - len(jmp_ins):
        length += ins.size
    else:  // 否则,当前内存块已存满,考虑下一个内存块
        segs_len.append(length)
        length = 0, n += 1
/* 在除末尾片段外的所有片段后添加跳转指令 */
for len in segs_len not last:
    sc_segments.append(shellcode[:len] + jmp_ins)
    shellcode = shellcode[len:] // 删除已处理的数据
sc_segments.append(shellcode)
return sc_segments
```

利用缓冲区溢出漏洞进行代码注入攻击需要两个步骤,分别是把攻击代码写入进程空间和劫持程序控制流到攻击代码处,该过程可通过构建 shellcode 约束表达式和 EIP 约束表达式的方法实现自动化。如算法 5 所示,首先需把 shellcode 片段写入进程中对应的可控内存块,实现方法为依次加载可控内存块 sym_bufs[n],并构建约束表达式将内存块中数据约束为对应的 shellcode 片段 sc_segments[n];接着构建约束表达式将漏洞状态的 EIP 寄存器值约束为 shellcode 存放内存的起始地址 sym_bufs[0].addr。

算法 5　构建利用约束

输入:漏洞状态 vul_state, 符号化内存块 sym_bufs, shellcode 片段 sc_segments

输出:约束表达式 constraints

```
/* 依次约束可控内存块中数据为对应 shellcode 片段的值 */
for n in range(len(sc_segments)):
                                    // 加载可控内存块
    memory = vul_state.load_mem(sym_bufs[n].addr)
                    // 将可控内存块中数据约束为 shellcode
    vul_state.add_constraints(memory == sc_segments[n])
    constraints.append(memory == sc_segments[n])
/* 约束 EIP 寄存器的值为 shellcode 起始地址 */
vul_state.add_constraints(vul_state.eip == sym_bufs[0].addr)
constraints.append(vul_state.eip == sym_bufs[0].addr)
return constraints
```

2.2.2　约束求解

对于上述构建的路径约束表达式和利用约束表达式,使用支持 SMT 求解理论的 Z3 求解器[18]进行求解。若有解,则得到一个可触发漏洞并进行代码注入攻击的 exploit;若无解,则表明检测到的漏洞无法利用。

3　实验与分析

实验运行环境为 Intel Core i7-7700HQ CPU,主频 2.8 GHz,4 GB 内存, Ubuntu 16.04 64 bits 系统,测试程序使用 gcc 5.4.0 编译。本文不考虑漏洞缓解机制的绕过,编译时不启用 NX 和 Stack Canary 保护,同时关闭系统 ASLR 保护[19]。为验证系统的有效性,本文设置两组实验,分别用于验证漏洞检测效果和测试自动生成利用数据的有效性。

3.1　漏洞检测

实验选取以下 3 个已披露漏洞作为测试样本,分

别使用 angr 和本文实现的 AutoExp 检测目标程序中存在的漏洞,并记录两种方法检测漏洞所需时间。实验结果如表 2 所示,表中第三列和第四列分别为目标程序的基本块数量和使用 AutoExp 进行预处理所得切片的基本块数量。从实验数据可知,使用切片技术对程序进行预处理可减少待分析程序的基本块数量,从而有效减小符号执行分析的复杂度。

表 2 漏洞检测结果

程序	漏洞编号	基本块数量		时间/s	
		total	slice	angr	AutoExp
Squirrel Mail	CVE-2004-0524	384	30	10.61	20.87
iwconfig	CVE-2003-0947	1 989	54	19.68	37.73
PSUtils	EDB-ID-890	11 026	95	1 920.76	127.05

表 2 使用两种方法检测漏洞所需时间。实验数据表明,当程序结构较简单且基本块数量较少时,直接使用 angr 进行符号执行分析能更快地检测到漏洞;而随着程序基本块数量的增大,angr 检测漏洞所需时间远远多于 AutoExp。具体原因如图 3 所示,AutoExp 在预处理阶段构建 CFG 需花费较多的时间,但是预处理可避免分析与漏洞无关的路径,从而在符号执行阶段花费的时间相对 angr 要少,且消耗的内存也随之减少。测试结果中,AutoExp 检测 PSUtils 中漏洞所需时间为 127.05 s,而 angr 所需时间是 AutoExp 的 15 倍,分析代码发现程序中 switch 语句会导致状态爆炸问题,该语句与漏洞路径无关,进行代码切片能避免分析该语句,使得符号执行效率得以提高。由此可见,对于结构较复杂的程序而言,本文所提方法可极大提高漏洞检测的效率。

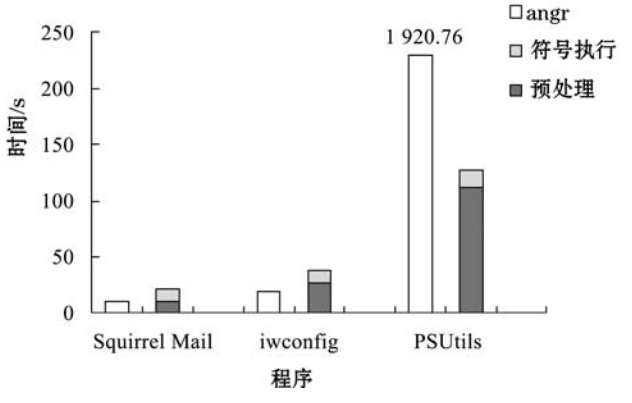

图 3 漏洞检测时间对比

3.2 漏洞利用

为验证系统自动生成 exploit 的适用性,以图 4 中漏洞程序 memo 和上述 3 个漏洞程序进行测试,测试时选取长度为 25 bytes 的 shellcode 利用漏洞。memo 程序第 14 行调用 read 函数获取用户输入到缓冲区 buf 中,由于第 15 行 strcpy 函数往数组 title 中写入过多的数据而导致缓冲区溢出漏洞,从而覆盖相邻内存中函数指针 func_ptr,利用该漏洞可劫持控制流进行代码注入攻击。

```
1 typedef struct memo{
2 char content[22];
3 time_t time;
4 char title[15];
5 int ( *func_ptr)();
6 }memorandum;
7 memorandum memo;
8 int message(){
9 printf("%s", memo.title);
10}
11int main(){
12 char buf[22];
13 memo.func_ptr = message;
14 read(0, buf, sizeof(buf));
15 strcpy(memo.title, buf);
16 read(0, buf, sizeof(buf));
17 strcpy(memo.content,buf);
18 time(&memo.time);
19 memo.func_ptr();
20}
```

图 4 缓冲区溢出漏洞程序 memo

实验结果如表 3 所示,AutoExp 可成功利用 4 个漏洞,而 Mayhem[7] 由于未考虑可控内存块不足以容纳 shellcode 的情况,因此无法成功利用 memo 中漏洞。实验结果表明,本文所提方法相对 Mayhem 具有更好的适用性。

表 3 漏洞自动利用结果对比

程序	漏洞编号	脆弱点	生成利用	
			Mayhem	AutoExp
memo	N/A	strcpy	N	Y
Squirrel Mail	CVE-2004-0524	sprintf	Y	Y
iwconfig	CVE-2003-0947	strcpy	Y	Y
PSUtils	EDB-ID-890	sprintf	Y	Y

下面具体分析 AutoExp 自动生成 exploit 的效果。程序 memo 中存在 content[22] 和 title[15] 两块连续可控内存,利用过程中选取不同长度的 shellcode,AutoExp可根据内存块能否容纳 shellcode 采取不同的 exploit 构造方法。如图 5 所示,当选取长度为 21 bytes 的 shellcode 利用漏洞时,由于存在能容纳 shellcode 的可控内存块,故选择可控内存块 content[22] 注入 shellcode 构造利用。

```
$ xxd shellcode
00000000: 6a0b 5899 5268 2f2f 7368 682f 6269 6e89  j.X.Rh//shh/bin.
00000010: e331 c9cd 80                             .1...

$ xxd exploit
00000000: 0101 0101 0101 0101 0101 0101 0101 0101  ................
00000010: 60a0 0408 1001 6a0b 5899 5268 2f2f 7368  `.....j.X.Rh//sh
00000020: 682f 6269 6e89 e331 c9cd 8010            h/bin..1....
```

图 5 shellcode 连续存放

如图 6 所示,当选取长度为 25 bytes 的 shellcode

利用漏洞时,由于不存在可容纳 shellcode 的内存块,故将 shellcode 分段存放在 content[22]和 title[15]中,并用指令"eb 07"实现不同分段间的跳转。

```
$ xxd shellcode
00000000: 31c0 5068 2f2f 7368 682f 6269 6e89 e350  1.Ph//shh/bin..P
00000010: 89e2 5389 e1b0 0bcd 80                   ..S.....

$ xxd exploit_segment
00000000: 89e1 b00b cd80 0101 0180 0108 0102 0108  ..............
00000010: 60a0 0408 0000 31c0 5068 2f2f 7368 682f  `.....1.Ph//shh/
00000020: 6269 6e89 e350 89e2 53eb 0708            bin..P..S...
```

图 6　shellcode 分段存放

上述结果表明,本文所提方法可根据漏洞程序中可控内存块的大小和所选取的 shellcode 调整 exploit 构造方法,具有更好的适用性。

4　结　语

本文对漏洞自动化利用方法进行了总结,提出一种基于符号执行的缓冲区溢出漏洞自动化利用方法。该方法采用危险函数切片减少漏洞检测中符号执行的状态数量,可有效缓解状态爆炸问题,提高符号执行的效率。在漏洞利用阶段,当进程中不存在空间足够的可控内存块时,将 shellcode 进行分段存放,具有更好的适用性。本文实现了缓冲区溢出漏洞利用的自动化,后续工作可进一步研究其他类型漏洞的自动化利用,以及自动绕过程序和系统中部署的漏洞缓解机制。

参　考　文　献

[1] 中国信息安全测评中心. 2017 年国内外信息安全漏洞情况[J]. 中国信息安全, 2018(1):117 – 121.

[2] CNNVD[OL]. http://www.cnnvd.org.cn.

[3] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and implications[C]//Proceedings of the 2008 IEEE Symposium on Security and Privacy. IEEE, 2008: 143 – 157.

[4] Avgerinos T, Cha S K, Rebert A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74 – 84.

[5] Heelan S, Kroening D. Automatic generation of control flow hijacking exploits for software vulnerabilities[D]. University of Oxford, 2009.

[6] Cha S S K, Avgerinos T T, Rebert A A, et al. Unleashing Mayhem on Binary Code[C]//Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE,2012:380 – 394.

[7] 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.

[8] Huang S K, Huang M H, Huang P Y, et al. Software Crash Analysis for Automatic Exploit Generation on Binary Programs[J]. IEEE Transactions on Reliability, 2014, 63(1): 270 – 289.

[9] Hu H, Chua Z L, Adrian S, et al. Automatic Generation of Data-Oriented Exploits[C]//24th USENIX Security Symposium. Washington: USENIX Association, 2015:177 – 179.

[10] Wang Y, Zhang C, Xiang X, et al. Revery: From Proof-of-Concept to Exploitable[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2018: 1914 – 1927.

[11] Alhuzali A, Gjomemo R, Eshete B, et al. NAVEX: Precise and scalable exploit generation for dynamic web applications [C]//27th USENIX Security Symposium, Washington: USENIX Association, 2018: 377 – 392.

[12] Roemer R, Buchanan E, Shacham H, et al. Return-Oriented Programming: Systems, Languages, and Applications [J]. Acm Transactions on Information & System Security, 2012, 15(1):1 – 34.

[13] Shoshitaishvili Y, Kruegel C, Vigna G, et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis [C]//2016 IEEE Symposium on Security and Privacy. IEEE Computer Society, 2016: 138 – 157.

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

[15] 苏璞睿, 应凌云, 杨轶. 软件安全分析与应用[M]. 北京: 清华大学出版社, 2017.

[16] Kiss K, Jsz J, Lehotai G, et al. Interprocedural Static Slicing of Binary Executables[C]//Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, 2003: 118 – 127.

[17] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution[C]//IEEE Symposium on Security and Privacy. Piscataway, 2010: 317 – 331.

[18] Moura L D, BjØrner N. Z3: an efficient SMT solver[C]// Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems. Springer-Verlag, 2008: 337 – 340.

[19] Shacham H, Page M, Pfaff B, et al. On the effectiveness of address-space randomization[C]//Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25 – 29, 2004. ACM, 2004: 298 – 307.