基于符号执行的格式化字符串漏洞自动验证方法研究

王瑞鹏，张　旻，黄　晖，沈　毅
（国防科技大学电子对抗学院，合肥，230037）

摘要　格式化字符串漏洞是一种常见的危害较大的软件漏洞。现有格式化字符串漏洞自动验证系统未充分考虑参数存储位置位于栈以外空间的情况，造成对该部分漏洞可利用性的误判。针对该问题，论文设计实现了一种基于符号执行的格式化字符串漏洞自动验证方法，首先根据参数符号信息检测当前格式化字符串函数漏洞，然后分别构建参数存储于不同内存空间情况下的漏洞验证符号约束，最后利用约束求解自动得到漏洞验证代码，实现了格式化字符串漏洞的自动验证。在 Linux 系统下对不同类型测试程序进行了实验，验证了方法的有效性。

关键词　格式化字符串漏洞；漏洞验证；符号执行；漏洞自动验证
DOI　10.3969/j.issn.1009-3516.2021.03.013
中图分类号　TP309.2　　**文献标志码**　A　　**文章编号**　1009-3516(2021)03-0082-07

Research on Automatic Exploit Generation Method of Format String Vulnerability Based on Symbolic Execution

WANG Ruipeng, ZHANG Min, HUANG Hui, SHEN Yi
(College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China)

Abstract　Format string vulnerability is a common and harmful software vulnerability. The misjudgment of the exploitability posed by software vulnerability is as much about some of the existing format string vulnerability automatic exploit generation system as the parameter storage location is outside the stack. In view of this problem, an automatic exploit generation method of format string vulnerabilities is designed based on symbolic execution. First, the current format string function vulnerabilities are detected according to the parameter symbol information, and then the exploit constraints with parameters stored in different spaces are constructed respectively, and finally the exploit code is obtained by using the constraint solution. The automatic verification of format string vulnerability is realized. The experiments with different test programs under Linux system verify the effectiveness of the method.

Key words　format string vulnerability; exploit; symbolic execution; automatic exploit generation

收稿日期：2020-12-25
基金项目：国家重点研发计划重点专项项目(2017YFB0802905)
作者简介：王瑞鹏(1997—)，男，陕西渭南人，硕士生，研究方向：二进制软件漏洞挖掘与验证。E-mail：wangruipeng@nudt.edu.cn
通信作者：张　旻(1966—)，男，安徽合肥人，教授，研究方向：信号与信息处理、计算智能。E-mail：zhangmin@nudt.edu.cn

引用格式：王瑞鹏，张旻，黄晖，等.基于符号执行的格式化字符串漏洞自动验证方法研究[J]. 空军工程大学学报(自然科学版)，2021，22(3)：82-88. WANG Ruipeng, ZHANG Min, HUANG Hui, et al. Research on Automatic Exploit Generation Method of Format String Vulnerability Based on Symbolic Execution[J]. Journal of Air Force Engineering University (Natural Science Edition), 2021, 22(3): 82-88.

随着信息化程度的不断提高，信息系统被广泛应用于金融、医疗、国防、工控等重要领域。高度信息化带来便利的同时也对网络空间安全提出了更高的要求。漏洞是威胁网络空间安全的重要因素，其中格式化字符串漏洞是一种常见的软件漏洞，它由 Tymm Twillman 首次发现于 1999 年[1]，其主要成因是程序未对来自外部的输入内容做出严格过滤，导致格式化控制符参数能够被外部输入所影响[2]，进而对信息系统产生较大威胁。例如 CVE-2012-0809 漏洞[3]，该漏洞可以提升 Linux 用户的权限，危害较大。因此对格式化字符串漏洞的挖掘和验证意义十分重大。

现有的自动化漏洞挖掘方法能够挖掘到大量的软件漏洞[4-8]，但不能对产生漏洞的可利用性进行验证。因此，漏洞自动验证技术在近些年成为了软件安全领域的一个研究热点。在漏洞自动验证领域，符号执行技术的准确性和可靠性使其成为自动化程序分析的重要工具。目前基于符号执行的自动化漏洞自动验证系统有很多，如 AEG[9]、Mayhem[10]、CRAX[11]等，其中 AEG 系统利用到了源码信息，其余系统都在二进制程序层面对漏洞进行自动验证。上述的系统都期望解决通用的漏洞自动验证，其漏洞验证模型中包含针对多种不同漏洞类型的验证方式，同时存在很多文献[12～17]针对特定漏洞或特定的漏洞验证方法，如黄钊等人的 FSVDTG[12] 主要针对格式化字符串漏洞进行漏洞验证方法研究，方皓[13]等人主要针对 Return-to-dl-resolve 漏洞验证方法进行研究。上述的大部分漏洞自动验证系统都是通过分析漏洞利用特点，构建漏洞验证模型，生成漏洞验证约束条件，约束求解得到漏洞验证代码。因此，漏洞验证系统的适用范围，通常由系统内漏洞验证模型适用范围决定。现有系统在针对格式化字符串漏洞自动验证时，构建了参数位于栈空间的漏洞验证模型，而忽略了参数位于其他空间时的情况。

本文提出了一种基于符号执行的格式化字符串漏洞自动验证方法，对现有的格式化字符串漏洞验证系统的漏洞验证模型进行了拓展，提高了格式化字符串漏洞自动验证系统的性能，降低了对于漏洞可利用性的误判，扩大了系统的适用范围，解决了对格式化字符串参数位于堆区及 BSS 段空间的漏洞自动验证的问题。

1　格式化字符串漏洞

1.1　格式化字符串参数

格式化字符串参数是格式化字符串函数用于将指定的字符串转换为期望的输入输出格式的控制符参数，格式化字符串参数的格式如下：

%[parameter][flags][width][precision]
[length]type

一般以%开始，以 type 结束，例如“%d”。配合 printf()等格式化字符串函数，能够将函数参数以指定格式进行输入输出。

其中 parameter 一般为 n$，指获取格式化字符串中的指定参数，flags 表示标志位，width 输出的最小字段宽度，precision 表示输出精度，即输出的最大长度，length 表示输出的字节长度。

格式化字符串中的 type 表示了预期输入输出的格式。常见的 type 类型及其使用效果如表 1 所示[18]。

表 1　常见 type 类型及其含义

类型	表示类型
d	有符号整型
u	无符号整型
x	16 进制无符号数
o	8 进制无符号数
s	字符串型
c	字符型
p	地址的格式表示对应变量的值
n	已经成功输出的字符个数写入对应的整型指针参数所指的变量

1.2　格式化字符串漏洞

格式化字符串漏洞是格式化控制符参数被程序的外部输入污染所导致的。例如 printf(a,b)语句，假设这里的 a 参数可被外部输入污染，若此时参数 a 中的内容为合法的格式化字符串内容时，变量 b 中的内容就会按照指定格式打印出来，但若 a 中内容为“%d%d”等非法格式化字符串内容，程序不仅会将变量 b 打印出来，而且会尝试将函数未定义的第 3 个参数打印出来，由于此时程序未定义第 3 个参数，所以系统便打印了非预期的内存内容[19]。如果精心构造格式化字符串漏洞中格式化控制符参数的内容，可以实现任意内存地址的读写，进而获取系统权限，引发严重的系统安全问题。

2　漏洞自动验证方法设计

本文提出了一种基于符号执行的格式化字符串漏洞自动验证方法。该方法核心流程如图 1 所示。

首先，监控程序的运行状态，在程序运行至格式化字符串函数时，进行漏洞的存在性检测；之后判断当前格式化字符串参数的存储位置，并根据参数的

不同存储位置,构建对应的漏洞验证约束;最后,将构建好的约束作为约束求解器输入,约束求解得到最终漏洞验证输入实例。

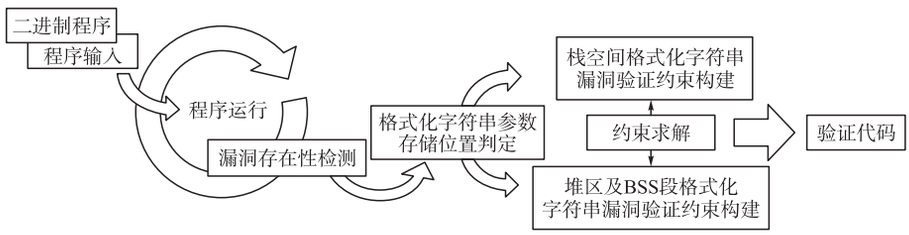

图 1 方法核心流程

2.1 漏洞存在性检测

如第 1.1 节中所述,格式化字符串漏洞是由于外部输入对格式化字符串参数的污染而导致的。本方法会监控程序中每个格式化字符串函数,判断格式化字符串参数是否被外部输入影响,从而判断漏洞的存在性。

首先对程序中目标危险函数进行挂钩操作。当测试程序运行至目标函数时对其进行拦截,并对目标函数的格式化字符串参数进行判断,若当前参数取值为符号值,则表明当前函数的格式化字符串参数已被系统引入的符号变元所污染,即该参数会被外部输入污染,从而证明格式化字符串漏洞的存在,反之,则说明当前函数不存在格式化字符串漏洞,如图 2 所示。

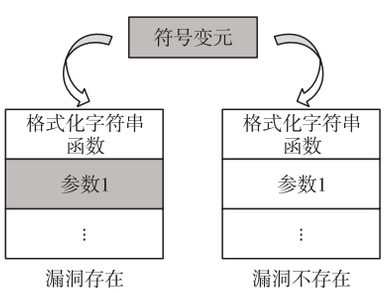

图 2 漏洞存在性检测方法

2.2 栈空间格式化字符串漏洞验证约束构建

当漏洞的格式化字符串参数存储于栈空间时,可通过格式化字符串函数在栈空间内布置任意地址,进而利用格式化字符串漏洞实现对任意地址读写。利用形如 address+%offset $s 的漏洞验证载荷进行任意地址读操作,利用形如 address + padding + %offset $n 的漏洞验证载荷进行任意地址写操作。其中 address 表示目标读写地址,offset 表示 address 与危险函数参数的相对偏移,该方法如图 3 所示。

通常期望写入 address 地址的数值远大于缓冲区大小,导致 address + padding 的长度无法满足漏洞验证条件,所以通常利用形如% num s 的格式化字符串产生长度为 num 的字符串。

为了构建上述漏洞验证载荷,实现对 address 地址的读写操作,需要首先计算 offset 的取值,当前的格式化字符串缓冲区地址保存在第 1 个参数内,因此 *(ESP+4)为栈中 address 的地址,因此 offset 的取值如式(1)所示。

$$offset=(*(ESP+4)-(ESP-4))/4 \quad (1)$$

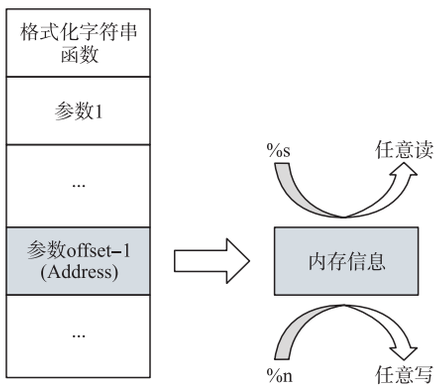

图 3 栈空间格式化字符串漏洞任意地址读写方法

在计算格式化字符串时,由于期望向目标地址写入的值往往较大,采取一次覆盖目标地址为期望值的时间效率极为低下,所以,在这种情况下,可以采用两次写入的方法,既节省缓冲区大小,又可以缩短漏洞验证时间,即利用"%hn"一次写入下 2 字节数据,上述计算格式化字符串漏洞验证载荷过程如算法 1 所示。

1)算法 1:计算格式化字符串漏洞验证载荷。

输入:目标地址 Ta,目标值 Tv,参数偏移 Offset

输出:目标格式化字符串 Fs

1. Tv_high=Tv<<16;

2. Tv_low=Tv mod 65536;

3. if Tv_high<Tv_low then

4. Fs=(Ta+2)+(Ta)+"%"+(Tv_high - 8)+"s";

5. Fs +="%"+(Offset) +"$n%"+(Tv_low - Tv_high) +"s%"+(Offset+1)+"$n";

6. else

7. Fs=(Ta)+(Ta+2)+"%"+(Tv_low - 8)+"s" ;

8. Fs + ="%" +(Offset)+"$n%"+(Tv_high-Tv_low)+"s%"+(Offset+1)+"$n";

9. end if

10. return Fs；

接下来，会构建符号内存内容与格式化字符串漏洞验证载荷相等的约束：

(Eq Formatstring$_0$(Read w8 0 buf))

(Eq Formatstring$_1$(Read w8 1 buf))

(Eq Formatstring$_2$(Read w8 2 buf))

⋮

若未指定 Ta 或 Tv，默认 Ta 为当前栈空间中存储的 EIP 值，并在栈空间内布置 shellcode，使 Tv 等于 shellcode 的地址。但由于格式化字符串漏洞验证载荷长度受到 Ta 和 Tv 的影响，所以无法直接断定 shellcode 布置位置，进而无法确定 Tv 具体数值。为了解决这一问题，本文采用了启发式算法来探寻 shellcode 的合理布置位置。从符号值的起始位置开始解尝试，并利用约束求解判断当前解是否为可行解，从而得到 shellcode 的合理布置位置，其过程如算法 2 所示。

2)算法 2：启发式算法计算 shellcode 的合理布置位置。

输入：符号区域起始位置 SymStart，符号区域结束位置 SymEnd，Shellcode，格式化字符串 Fs

输出：shellcode 布置位置 ShellcodeLocal

1. for i = Symstart to SymEnd-len(Shellcode) do

2. Set(Shellcode,SymStart + i)；

3. Fs= CalcFs(SymStart + i)；

4. Set(Fs,SymStart)；

5. if getSymSolution() then

6. return ShellcodeLocal；

7. end if

8. end for

在确定了 shellcode 的布置位置后，构建 shellcode 布置的约束条件为：

(Eq Shellcode$_0$(Read w8 ShellcodeLocal buf))

(Eq Shellcode$_1$(Read w8 ShellcodeLocal+1 buf))

(Eq Shellcode$_2$(Read w8 ShellcodeLocal+2 buf))

⋮

2.3　堆区及 BSS 段格式化字符串漏洞验证约束构建

当漏洞函数的格式化字符串参数存储于堆区及 BSS 段时，栈空间无法被直接写入地址。由于无法向栈空间写入目标地址，所以 2.2 节中的方法在当前情况下失效。目前大多的格式化字符串漏洞自动验证系统在遇到该情况时，会判定当前的漏洞无法被攻击者利用，从而给予该漏洞以中低危风险等级。但事实上，这种漏洞仍有可能通过间接方式，达到对目标地址的任意读写。

在程序运行时，程序的栈空间一般不会存在期望修改地址的指针，所以需要一个指向栈地址的指针，间接向栈空间填入期望修改的地址。而 EBP 寄存器在程序运行中的作用是将各个函数调用串联起来，所以 EBP 指针就是一条存在于栈空间内的指针链，因此系统会利用栈下存储的 EBP 指针，间接实现任意地址读写，如图 4 所示。

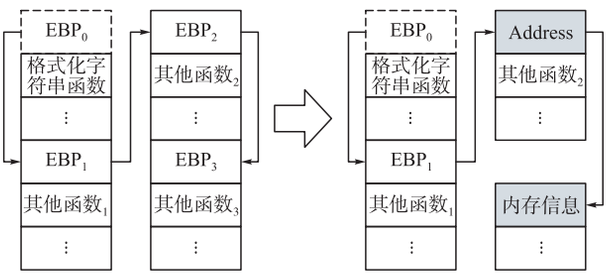

图 4　堆区及 BSS 段格式化字符串漏洞任意地址读写方法

若此时目标地址 Address 的值无法确定，默认通过修改 EBP 将函数栈迁移至堆区及 BSS 段，并在该部分内存区域构建栈内存布局，控制程序执行流程，如图 5 所示。

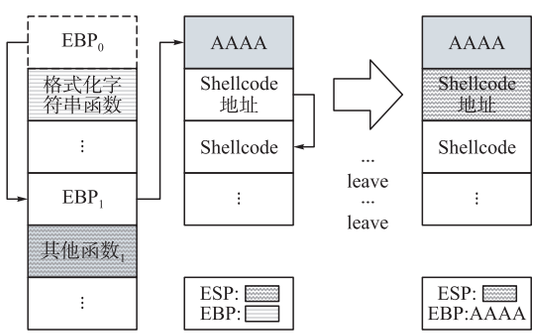

图 5　栈迁移的流程

通过格式化字符串修改栈空间内存放 EBP 内容的地址，使其指向格式化字符串参数所在空间，通过 2 次 leave 指令后，程序的 ESP 寄存器会指向该空间，此时程序的栈被迁移到该内存空间，随后的 ret 指令便会将程序执行流程劫持至 shellcode 处。此时格式化字符串漏洞验证载荷为“%”+value+“s%”+offset+“$nAAAA”+ShellcodeLocal+shellcode，其中 ShellcodeLocal 由算法 2 可得，offset 为栈中 EBP 内容存放地址的函数参数相对偏移，可由各个函数栈的 EBP 成链的特点计算得到。

2.4　约束求解

在得到不同类型的格式化字符串漏洞验证用例约束后，对其进行处理，得到期望的漏洞验证用例生成的约束。

此时得到的约束由程序运行时的符号执行路径

约束 CrashConstraints 和输入约束 InputConstraints 构成,当前约束能够触发程序漏洞路径,但其与第 2.2、第 2.3 节中最终得到的约束产生了冲突,如下式所示。

$$(\text{Eq Formatstring}_0(\text{Read w8 0 buf}))\bigcap(\text{Eq "A" (Read w8 0 buf)})$$

$$\vdots$$

因此需要将输入约束 InputConstraints 从最终的生成约束中减去,得到约束为:

$$(\overline{\text{InputConstraints}}\bigcap\text{CrashConstraints})\bigcap\text{ExploitConstraints}$$

将最终的约束作为约束求解器的输入,得到最终的漏洞验证代码。

3 实验

3.1 系统实现

依照上述方法,本文基于 S2E[20] 实现了格式化字符串漏洞自动验证原型系统 FSAEG,框架如图 6 所示。系统利用 QEMU[21] 进行全系统仿真,对目标程序运行状态进行监控,利用 KLEE[22] 实现系统的符号执行功能,利用 Z3[23] 实现约束求解。

格式化字符串漏洞自动验证插件通过监控器获取进程和模块加载的时刻以及程序运行时的状态,并利用获取到的信息构建漏洞自动验证约束条件,实现漏洞自动验证。

3.2 实验验证

本次实验的环境为 Linux 32 位系统关闭 ASLR、CANARY、NX 等漏洞利用缓解机制。实验宿主机配置为 Windows 10、Intel Core i7-9750H@2.60 GHz、32 GB 内存、虚拟机配置为 Ubuntu 16.04、7 GB 内存。

为了证明系统的有效性,本文以一个典型格式化字符串参数存储于栈空间的漏洞示例 FMT-3,来验证系统在应对格式化字符串参数位于栈空间时自动验证的有效性。

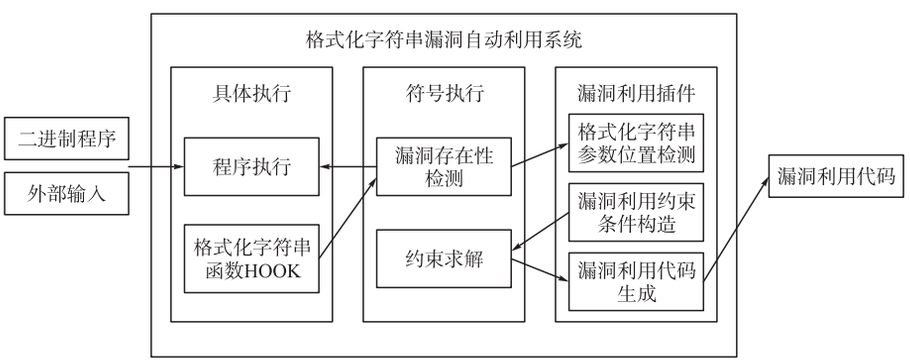

图 6 FSAEG 系统框架

FMT-3 的代码如图 7 所示。在代码的 13 行,存在明显的格式化字符串漏洞,传入 printf 的格式化字符串为可以通过用户输入改变的 s1,而 s1 的位置则处在栈空间内。

```
1  int game()
2  {
3      int result;
4      char s1;
5
6      while(1)
7      {
8          puts("test");
9      read_char(&s1,0xBE);
10         result = strcmp(&s1,"END");
11         if(!result)
12             break;
13         printf(&s1);
14     }
15     return result;
16 }
```

图 7 FMT-3 程序源码

以 2019 年 BRHG 自动化漏洞挖掘竞赛格式化字符串漏洞题目 A0012 来验证系统在应对格式化字符串参数位于栈以外空间时自动验证的有效性。

A0012 代码如图 8 所示。在代码的 23 行,存在明显的格式化字符串漏洞。传入 printf 的格式化字符串为可以通过用户输入改变的 buff,而 buff 处于 bss 段,漏洞函数调用前有 3 次其他函数调用。

```
1  char buff[128];
2  int main(int argc, char *argv[]){
3      bot();
4      return 0;
5  }
6  void bot(){
7      puts("This is Echo Bot");
8      game();
9  }
10 void game(){
11     while(1){
12         puts("Input END to exit");
13         read_char(buff, 128);
14         if(!strcmp(buff, "END")){
15             break;
16         }
19         else{
18             echo();
19         }
20     }
21 }
22 void echo(){
23     printf(buff);
24 }
```

图 8 BRHG-A0012 程序源码

在实验结果上,本系统能够对上述示例产生漏洞验证代码,如图 9、图 10 所示,并成功实现漏洞验证,获取系统权限。

为了对比不同系统之间的性能差异,本文对 HEAP-FMT、TEA-FMT 程序做了测试,其中 HEAP-FMT 为格式化字符串参数存储于堆空间的格式化字符串漏洞程序,TEA-FMT 为带有 tea[24] 加密的缓冲区位于栈空间的格式化字符串漏洞程序。在对比 FSVDTG、CRAX、AFL 和本系统后,得到结果如表 2 所示。

```
→ exp hexdump -C exploit-fmtstr-exp.bin
00000000  dc 4c c7 bf de 4c c7 bf  25 31 39 34 38 38 63 25  |.L...L..%19488c%|
00000010  36 24 68 6e 25 32 39 35  39 39 63 25 37 24 68 6e  |6$hn%29599c%7$hn|
00000020  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 50  |1.Ph//shh/bin..P|
00000030  53 89 e1 31 d2 52 b0 0b  cd 80 00 00 00 00 00 00  |S..1.R..........|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 0a 45 4e  |..............EN|
000000c0  44 0a                                             |D.|
000000c2
```

图 9　FMT-3 漏洞验证代码

```
→ exp hexdump -C exploit-fmtstr-exp.bin
00000000  80 25 31 33 35 31 38 36  39 35 39 63 25 36 24 6e  |.%135186959c%6$n|
00000010  41 41 41 41 18 ca 0e 08  6a 68 68 2f 2f 2f 73 68  |AAAA....jhh///sh|
00000020  2f 62 69 6e 89 e3 68 01  01 01 01 81 34 24 72 69  |/bin..h.....4$ri|
00000030  01 01 31 c9 51 6a 04 59  01 e1 51 89 e1 31 d2 6a  |..1.Qj.Y..Q..1.j|
00000040  0b 58 cd 80 00 00 00 00  00 00 00 00 00 00 00 00  |.X..............|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 0a  |................|
00000080  45 4e 44 0a                                       |END.|
00000084
```

图 10　BRHG-A0012 漏洞验证代码

表 2　各系统对不同类型实例测试结果

System	Testing content	FMT-3	FMT-4	BRHG-A0012	HEAP-FMT	TEA-FMT
FSVDTG	Vulnerability Detection	√	√	√	√	√
	Exploit Generation	√	√	×	×	×
CRAX	Vulnerability Detection	√	×	√	×	√
	Exploit Generation	√	×	×	×	×
AFL	Vulnerability Detection	×	√	×	√	√
	Exploit Generation	N/A	N/A	N/A	N/A	N/A
FSAEG	Vulnerability Detection	√	√	√	√	√
	Exploit Generation	√	√	√	√	×

针对该实验结果分析如下：

1）AFL 是模糊测试的常用工具之一，其采用模糊测试技术对漏洞进行挖掘和测试，能够发现程序中格式化字符串漏洞，但是由于在 FMT-3 实例和 BRHG-A0012 实例中存在可持续输入的循环结构，导致 AFL 无法生成测试用例，且其并不支持对漏洞自动验证功能。

2）CRAX 是一种典型的基于符号执行的漏洞自动验证工具，其支持对漏洞产生漏洞验证代码，但在判断格式化字符串时，需要格式化缓冲区长度大于 50，故未能成功检测 FMT-4 及 HEAP-FMT 中的格式化字符串漏洞，此外 CRAX 只采用了格式化字符串栈中任意读写的漏洞验证模型，所以其不能对缓冲区位于其他空间的实例进行漏洞自动验证。

3）FSVDTG 是专门针对格式化字符串漏洞进行漏洞测试的工具，其能够检测各种格式化字符串漏洞，并对部分程序能够进行漏洞自动验证，但是由于其同样只采用了单一的漏洞验证模型，所以不能对格式化字符串参数位于其他空间的漏洞程序进行漏洞自动验证。

本系统（FSAEG）能够能够检测格式化字符串漏洞，并在格式化字符串参数位于栈外其他空间时，成功实现漏洞验证代码的自动生成，达到其他系统所无法达到的效果。

在此次实验中，所有系统均无法对含有 tea 加密的实例 TEA-FMT 进行漏洞自动验证，主要原因是符号执行中约束求解器的性能瓶颈。

4　结语

本文总结了格式化字符串漏洞验证的相关原理，并针对现有系统无法解决的参数位于栈以外内存空间的漏洞自动验证问题，设计实现了能够适用于不同参数存储位置的格式化字符串漏洞自动验证系统 FSAEG。最后通过实验验证了方法的有效性，并与同类方法进行了对比，证实了 FSAEG 系统能够有效解决目标问题。但由于符号执行本身的路径爆炸问题，使得符号执行并不能解决过于庞大的程序，因此在下一步的工作中，拟采用模糊测试、静态分析等手段，辅助符号执行，进一步提升系统的自动验证效果。

参考文献

[1] TWILLMAN T. Exploit for Proftpd1.2.0pre6 [EB/OL]. (1999-09-01)[2020-08-17]. http://seclists.org/bugtraq/1999/Sep/328.

[2] 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，2016：227.

[3] Red Hat. CVE-2012-0809[EB/OL]. (2012-01-19)[2020-08-17]. http://cve. mitre. org/cgi-bin/cvename. cgi? name=CVE-2012-0809.

[4] KING J C. Symbolic Execution and Program Testing[J]. Communications of the ACM, 1976, 19(7):385-394.

[5] ZALEWSKI. M (2017) American Fuzzy Lop [EB/OL]. (2017-02-25)[2020-08-17]. http://lcamtuf. coredump. cx/afl/. Accessed 25 Dec 2017

[6] STEPHENS N, GROSEN J, SALLS C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]//Proceedings of the Network and Distributed System Security Symposium. San Diego, USA: Internet Society, 2016:21-24.

[7] RAWAT S, JAIN V, KUMAR A, et al. VUzzer: Application-aware Evolutionary Fuzzing [C]//NDSS Symposium 2017. 2017: 1-14.

[8] WANG T, WEI T, GU G, et al. TaintScope: A Check-Sumaware Directed Fuzzing Tool for Automatic Software Vulnerability Detection [C]//2010 IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE, 2010: 497-512.

[9] AVGERINOS T, SANG K C, REBERT A, et al. Automatic Exploit Generation[J]. Communications of the ACM, 2014, 57(2):74-84.

[10] AVGERINOS T, REBERT A, BRUMLEY D, et al. Unleashing Mayhem on Binary Code[C]// 2012 IEEE Symposium on Security and Privacy. San Francisco, CA, USA:IEEE: 2012.

[11] 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]//2012 IEEE Sixth International Conference on Software Security & Reliability. Gaithersburg, MD, USA: IEEE, 2012.

[12] 黄钊, 黄曙光, 邓兆琨,等. 格式化字符串漏洞自动检测与测试用例生成[J]. 计算机应用研究, 2019,39(8):2464-2468.

[13] 方皓, 吴礼发, 吴志勇. 基于符号执行的 Return-to-dl-resolve 利用代码自动生成方法[J]. 计算机科学, 2019, 46(2):127-132.

[14] 黄宁, 黄曙光, 黄钊. 基于符号执行的异常处理结构体覆写攻击自动检测方法[J]. 吉林大学学报(工学版), 2020, 50(3):1024-1030.

[15] 李超, 胡建伟, 崔艳鹏. 基于符号执行的缓冲区溢出漏洞自动化利用[J]. 计算机应用与软件, 2019, 36(9):327-333.

[16] 黄桦烽, 王嘉捷, 杨轶,等. 有限资源条件下的软件漏洞自动挖掘与利用[J]. 计算机研究与发展, 2019, 56(11):2299-2314.

[17] 黄钊, 黄曙光, 邓兆琨, 等. 基于 SEH 的漏洞自动检测与测试用例生成[J]. 计算机科学,2019, 46(7): 133-138.

[18] ROBERTSON C. Format Specification Syntax: Printf and Wprintf Functions[EB/OL]. (2019-10-21)[2020-12-25]. https://docs. microsoft. com/en-us/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions? view=vs-2019.

[19] CHIPOUNOV V, KUZNETSOV V, CANDEA G. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems[C]// Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. California, USA: ACM, 2011.

[20] BELLARD F. QEMU, A Fast and Portable Dynamic Translator[C]//USENIX Annual Technical Conference. [S. l.]: FREENIX Track. 2005: 41-46.

[21] CADAR C, DUNBAR D, ENGLER D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]// Usenix Conference on Operating Systems Design & Implementation. [S. l.]:USENIX Association, 2009.

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

[23] WHEELER D J, NEEDHAM R M. TEA, A Tiny Encryption Algorithm[C]//International Workshop on Fast Software Encryption. Berlin, Heidelberg: Springer, 1994: 363-366.

(编辑:徐楠楠)