
COMP6447
intel x86

reversing

and maybe some hacks (**shellcode development**)

But first some questions from wargame 2

- What is __x86_get_pc_thunk_bx?
  - **get** _ (**p**rogram **c**ounter) _ **thunk** _ (e)**bx**
  - A way of finding the base of the binary
  - ebx (usually) points to beginning of a data region
  - Can be disabled with compiler `-fno-pic`
  - Usually only called in main function

In computer programming, a thunk is a subroutine used to inject an additional calculation into another subroutine

```
__x86.get_pc_thunk.bx:
08048360  mov     ebx, dword [esp {__return_addr}]
08048363  retn     {__return_addr}
```

Used to find where variables are

```
080484fc  push    eax {var_10} {var_18}
080484fd  lea     eax, [ebx-0x1a00]  {data_8048600}
08048503  push    eax {var_1c}  {data_8048600}
```

How to reverse 101

- Reversing takes patience
- Look for **patterns**
  - What does a loop look like?
  - What do conditionals look like?
  - What do different variables look like (ints/shorts/floats/strings/pointers/arrays)
- Chain these patterns together to get a big picture
- Don’t spend too much time understanding **individual** instructions
  - Try to get the bigger picture

Conditional jumps

- Appeared in last weeks wargames
- Usually < 3 instructions
  - Compare 2 values
  - Jump if a condition is set

```
CMP eax, ebx
JNZ address
```

Loops

```
if(condition)
{
    do { stuff } while (condition);
}
```

- Loops are just conditionals with a **goto**
  - Do the comparison
  - If false jump to end of loop
  - Else do stuff in Loop then jump back to top
- Loops are usually compiled backwards (easier?)
  - while(x) {} -> if (x) { do {...} while(x) }

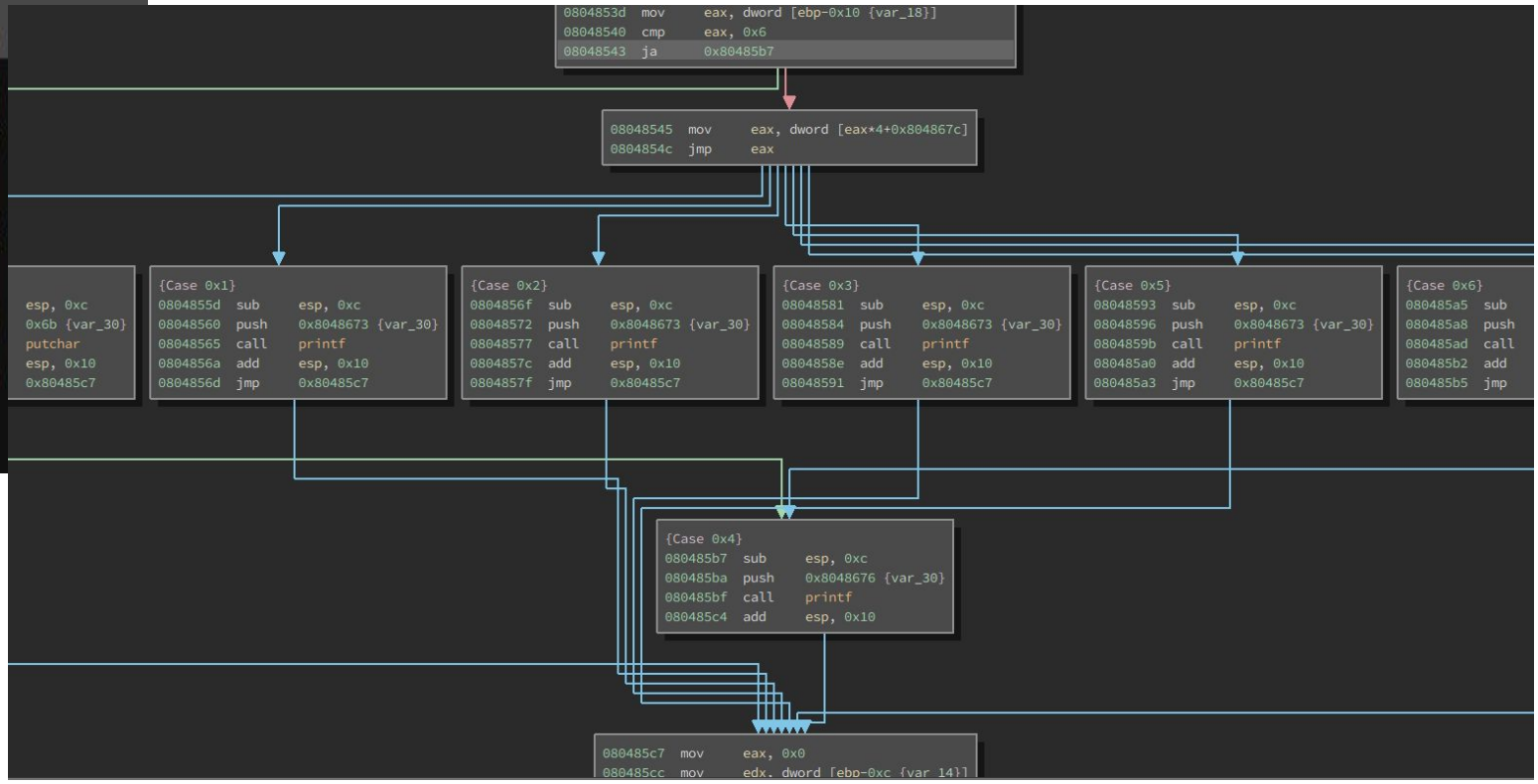
> Loop demo

What about switch statements

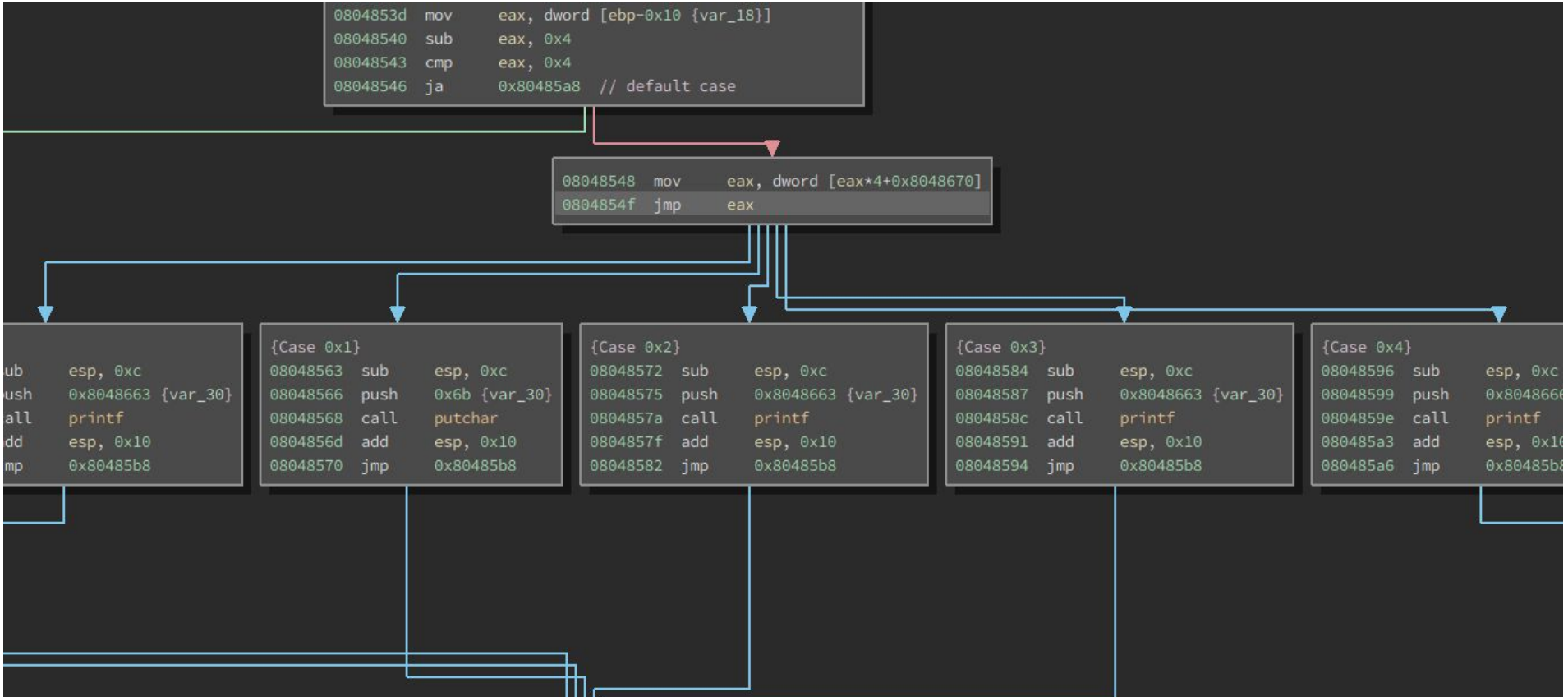
- Several cases
  - simple case of small close together numbers. ie: x = 1 or 2 or 3
  - Simple case of larger close together numbers. ie: x = 4 or 5 or 7 or 8 or 9
  - Complex case of random things

Case 1

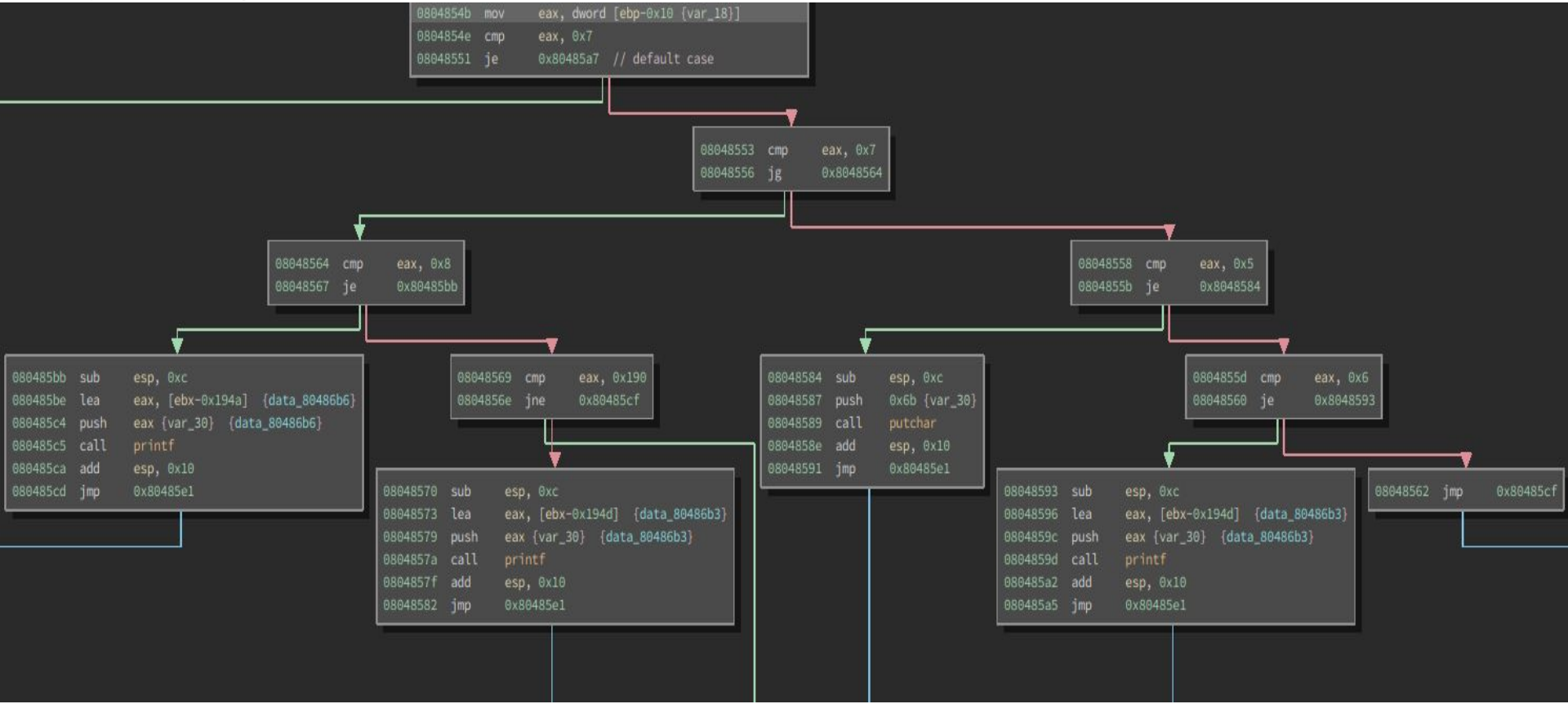
```
0804867c  uint32_t jump_table_804867c[7] =
0804867c  {
0804867c      [0x0] = 0x804854e
08048680      [0x1] = 0x804855d
08048684      [0x2] = 0x804856f
08048688      [0x3] = 0x8048581
0804868c      [0x4] = 0x80485b7
08048690      [0x5] = 0x8048593
08048694      [0x6] = 0x80485a5
08048698  }
```



Case 2


0804853d mov eax, dword [ebp-0x10 {var_18}]
08048540 sub eax, 0x4
08048543 cmp eax, 0x4
08048546 ja 0x80485a8 // default case
08048548 mov eax, dword [eax*4+0x8048670]
0804854f jmp eax
ub esp, 0xc
ush 0x8048663 {var_30}
all printf
dd esp, 0x10
mp 0x80485b8
{Case 0x1}
08048563 sub esp, 0xc
08048566 push 0x6b {var_30}
08048568 call putchar
0804856d add esp, 0x10
08048570 jmp 0x80485b8
{Case 0x2}
08048572 sub esp, 0xc
08048575 push 0x8048663 {var_30}
0804857a call printf
0804857f add esp, 0x10
08048582 jmp 0x80485b8
{Case 0x3}
08048584 sub esp, 0xc
08048587 push 0x8048663 {var_30}
0804858c call printf
08048591 add esp, 0x10
08048594 jmp 0x80485b8
{Case 0x4}
08048596 sub esp, 0xc
08048599 push 0x804866
0804859e call printf
080485a3 add esp, 0x1
080485a6 jmp 0x80485b

Case 3


0804854b mov eax, dword [ebp-0x10 {var_18}]
0804854e cmp eax, 0x7
08048551 je 0x80485a7 // default case
08048553 cmp eax, 0x7
08048556 jg 0x8048564
08048564 cmp eax, 0x8
08048567 je 0x80485bb
08048558 cmp eax, 0x5
0804855b je 0x8048584
080485bb sub esp, 0xc
080485be lea eax, [ebx-0x194a] {data_80486b6}
080485c4 push eax {var_30} {data_80486b6}
080485c5 call printf
080485ca add esp, 0x10
080485cd jmp 0x80485e1
08048569 cmp eax, 0x190
0804856e jne 0x80485cf
08048584 sub esp, 0xc
08048587 push 0x6b {var_30}
08048589 call putchar
0804858e add esp, 0x10
08048591 jmp 0x80485e1
0804855d cmp eax, 0x6
08048560 je 0x8048593
08048570 sub esp, 0xc
08048573 lea eax, [ebx-0x194d] {data_80486b3}
08048579 push eax {var_30} {data_80486b3}
0804857a call printf
0804857f add esp, 0x10
08048582 jmp 0x80485e1
08048593 sub esp, 0xc
08048596 lea eax, [ebx-0x194d] {data_80486b3}
0804859c push eax {var_30} {data_80486b3}
0804859d call printf
080485a2 add esp, 0x10
080485a5 jmp 0x80485e1
08048562 jmp 0x80485cf

More patterns to recognising

- Can’t underestimate how important this is.
  - Makes reversing quicker
- Certain code constructs occur over and over and become obvious to identify
  - Chain these easy to identify patterns together to understand what is happening

Another demo (control structures)

Integers

- Most things in C are just **ints** in **disguise**
  - Signed/Unsigned ints
  - Longs are just big ints
  - Shorts are just small ints
  - Chars are just smaller ints
- They all use the same instructions to modify them
  - All use add/sub to do maths
  - All use move/push/pop to move them
  - How can we tell the size from these instructions?
- Pointers are just **ints** that we treat **special**

Implications of instructions

ASM OPERATION	IMPLICATION	EXAMPLE
[dereference]	Operand is a pointer	cmp ecx, [edi] ; edi is a pointer
Data size [dereference]	Operand is a pointer to data values of indicated size	movzx ecx, byte ptr [eax+5Ah] ; [eax+5Ah] is a ; pointer to a byte
movsx/sal/sar/idiv	Source operand is signed	movsx edx, word ptr [eax+80h] ; [eax+80h] points ; to a signed short
movzx/shl/shr/div	Source operand is unsigned	movzx edi, di ; di is an unsigned short

jle/jge/jle/jl	Previous flag-setting operation
	was dealing with signed operands
		mov ebx, 10h
		cmp ecx, ebx
		jle short error_epilog2
		; ecx is signed

jae/ja/jbe/jb	Previous flag-setting operation was dealing with unsigned operands	cmp [esi+4], edi jbe short error_epilog2 ; [esi+4] is unsigned

Spot the difference

mov eax, **dword** ptr [esp]

mov eax, **word** ptr [esp]

mov eax, **byte** ptr [esp]

mov **ax**, [esp]

mov **al**, [esp]

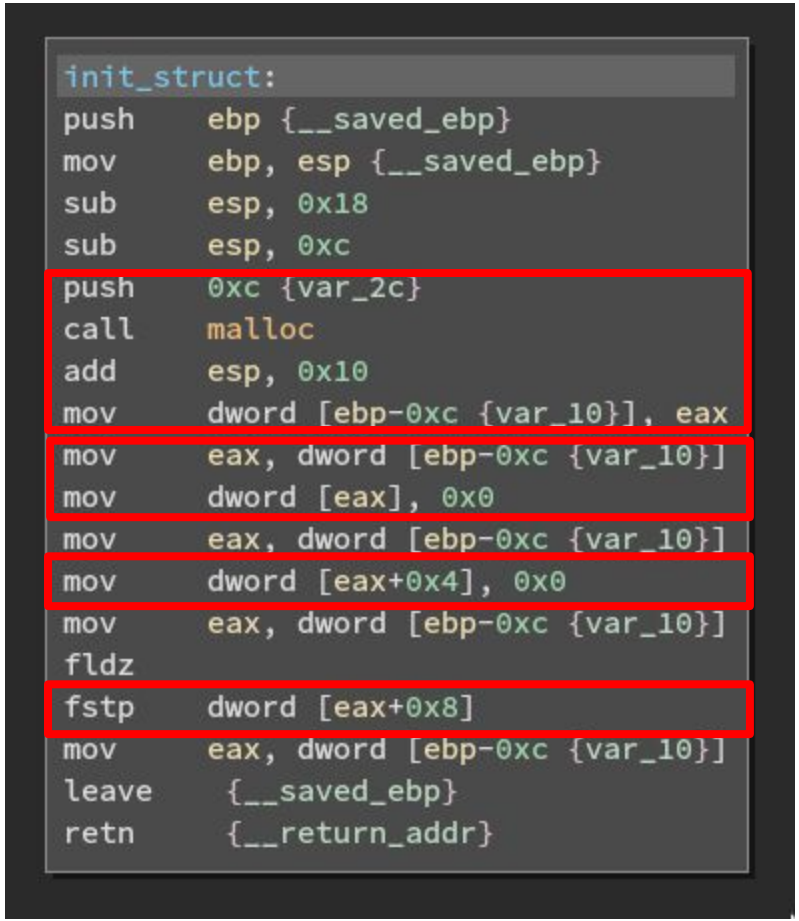
More on data types

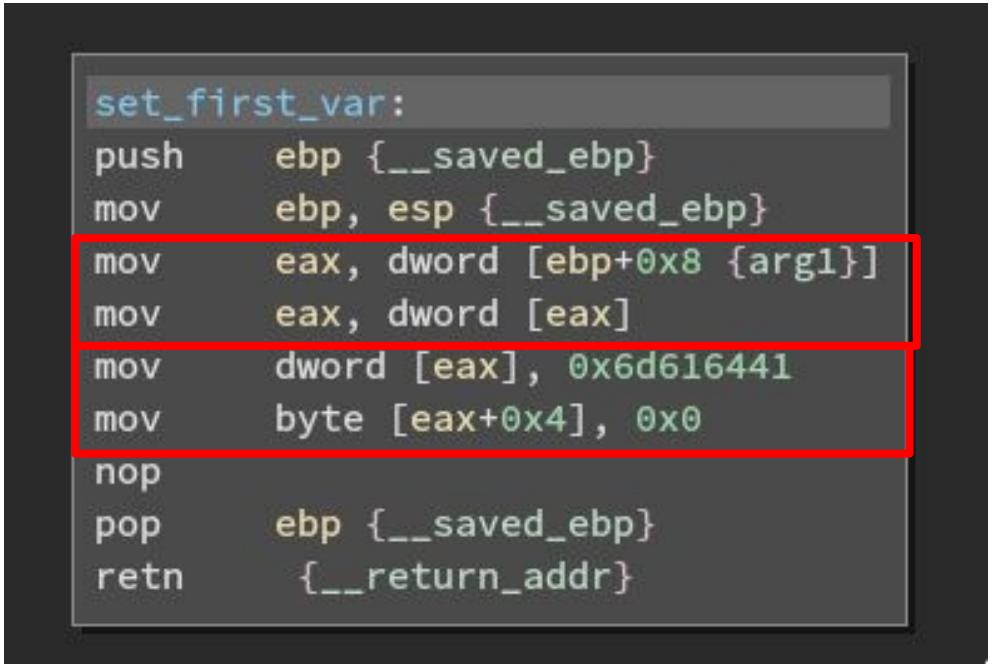
- Knowing what a data structure looks like helps know what a function does with it
- Types are obvious based on the instructions used to access them
  - The size of a variable is obvious from the instruction used
  - Pointers are obvious if they get dereferenced
  - Sign of a variable is obvious from the instructions used
- How to spot a struct
  - Allocations are of a fixed size
  - Populated using constant offsets from the base
  - Data type of each field is obvious from instruction used
  - Context of field usage lets you know what they are
    - If a field is always OR'd/AND'd with a value like 0x1,0x2,0x4,etc, it is a bit field/flags
    - If a value is compared against the number 12, it is obviously an int

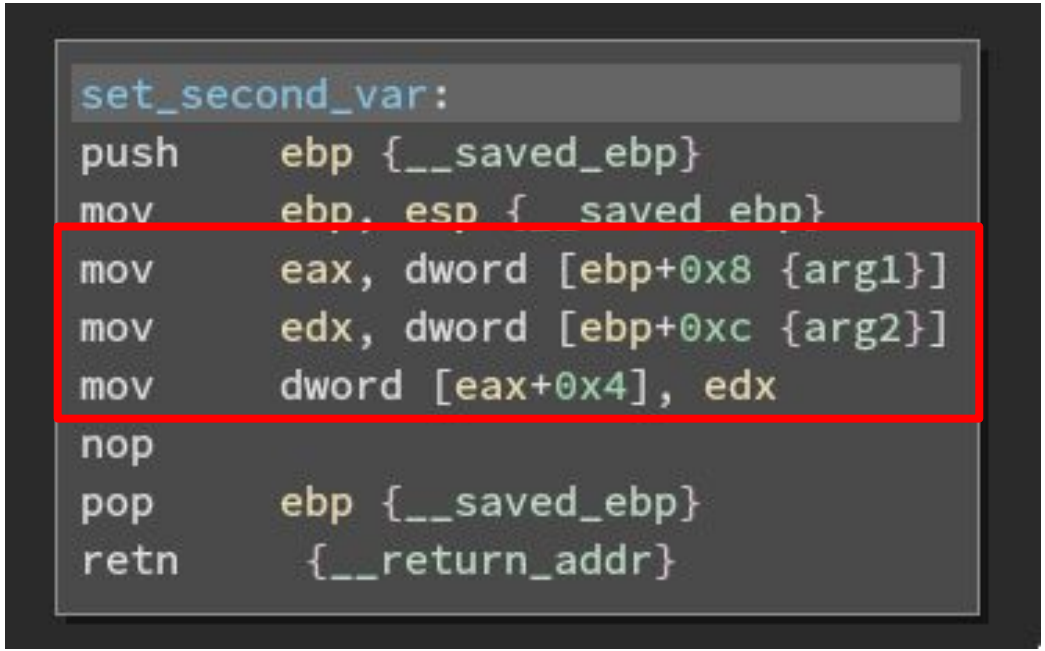
demo

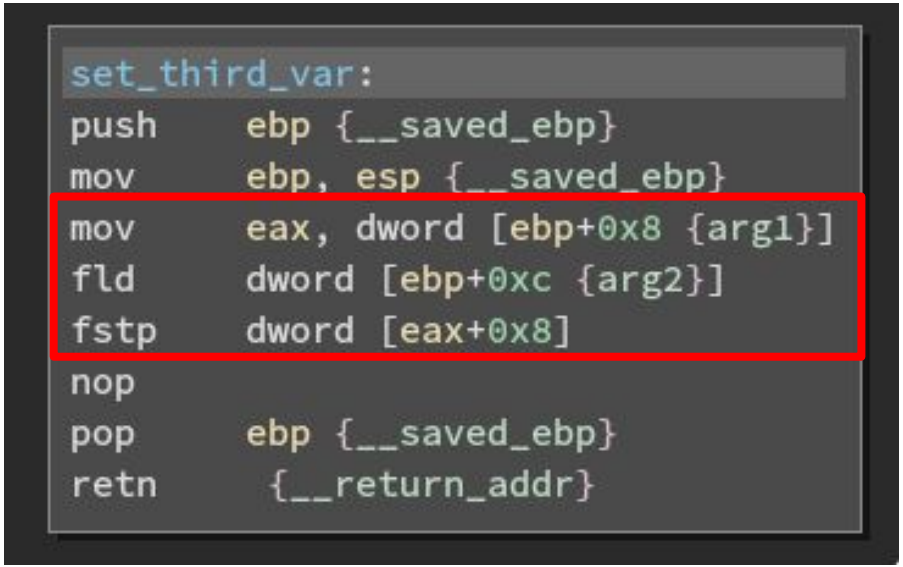
```c
typedef struct {
  char *name;
  int age;
  float money;
} Person;
```

```c
void set_first_var(Person *person) { strcpy(person->name, "Adam"); }

void set_second_var(Person *person, int age) { person->age = age; }

void set_third_var(Person *person, float money) { person->money = money; }

Person *init_struct() {
  Person *p = malloc(sizeof(Person));
  p->name = NULL;
  p->age = 0;
  p->money = 0.0;

  return p;
}
```


init_struct:
push ebp {__saved_ebp}
mov ebp, esp {__saved_ebp}
sub esp, 0x18
sub esp, 0xc
push 0xc {var_2c}
call malloc
add esp, 0x10
mov dword [ebp-0xc {var_10}], eax
mov eax, dword [ebp-0xc {var_10}]
mov dword [eax], 0x0
mov eax, dword [ebp-0xc {var_10}]
mov dword [eax+0x4], 0x0
mov eax, dword [ebp-0xc {var_10}]
fldz
fstp dword [eax+0x8]
mov eax, dword [ebp-0xc {var_10}]
leave {__saved_ebp}
retn {__return_addr}


set_first_var:
push ebp {__saved_ebp}
mov ebp, esp {__saved_ebp}
mov eax, dword [ebp+0x8 {arg1}]
mov eax, dword [eax]
mov dword [eax], 0x6d616441
mov byte [eax+0x4], 0x0
nop
pop ebp {__saved_ebp}
retn {__return_addr}


set_second_var:
push ebp {__saved_ebp}
mov ebp, esp {__saved_ebp}
mov eax, dword [ebp+0x8 {arg1}]
mov edx, dword [ebp+0xc {arg2}]
mov dword [eax+0x4], edx
nop
pop ebp {__saved_ebp}
retn {__return_addr}


set_third_var:
push ebp {__saved_ebp}
mov ebp, esp {__saved_ebp}
mov eax, dword [ebp+0x8 {arg1}]
fld dword [ebp+0xc {arg2}]
fstp dword [eax+0x8]
nop
pop ebp {__saved_ebp}
retn {__return_addr}

Structs containing arrays

```
mov     eax, [esi+edi*4+18h] ; access array which starts at [esi+18], each element is 4 bytes
```

dynamic analysis

```
3 honeypot% cat a.c
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) { printf("hello\n"); }
```

```
3 honeypot% ltrace ./a.out
__libc_start_main([ "./a.out" ] <unfinished ...>
puts("hello"hello
)                                                    = 6
+++ exited (status 0) +++
```

strace prints a lot more

Understanding programs with gdb

- Know important commands
  - break
  - step/next
  - continue/finish
  - **attach**
- Know how to attach gdb to your pwntools scripts
  - Demo here

How to reverse larger programs

- Walk through the assembly, **slowly**.
- At first, **translate into C**, if you know it, otherwise, pseudocode or whatever you do know. Good reversers eventually don‟t bother, they just **understand the assembly**.
- If you aren’t sure if something can go two ways, **write them up and try them.**
- If using IDA/BINJA, **rename things** when you work out what they do.
    - If you have lots of var_4, var_8, etc. **rename** them things easy to remember “**pizza**, cheese, cola” **until you know what they do, then give them proper name**

Different approaches

- Similar to approaches to source code auditing
- Starting at the top:
  - Find main(), off you go son
  - **Good** for small programs, malware
  - **Bad** for large programs, can be inefficient
- Starting at user-controlled input:
  - Good for finding vulnerabilities / finding parts of program you can affect.
  - Often easy to find (e.g. find socket accept(), files read etc.)
- Finding particular strings or recognisable constructs:
  - Good for examining a particular part of the program that you might be interested in e.g. finding where the string “Please enter serial key” is used.
  - Encryption often has easily identifiable patterns of instruction usage and constants.
- Strace on linux, procmon on windows.

B

Now some shellcoding….

Break + questions


COMP6447
90 34 32 75 32
14 43 12

What is shellcode

- Historically, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability.
- Shellcode is commonly written in machine code.
- After a memory corruption based exploitation
  - You need someway of executing code
    - **Shellcode**
    - ROP/RET2CODE/RET2LIBC

Why do we need shellcode

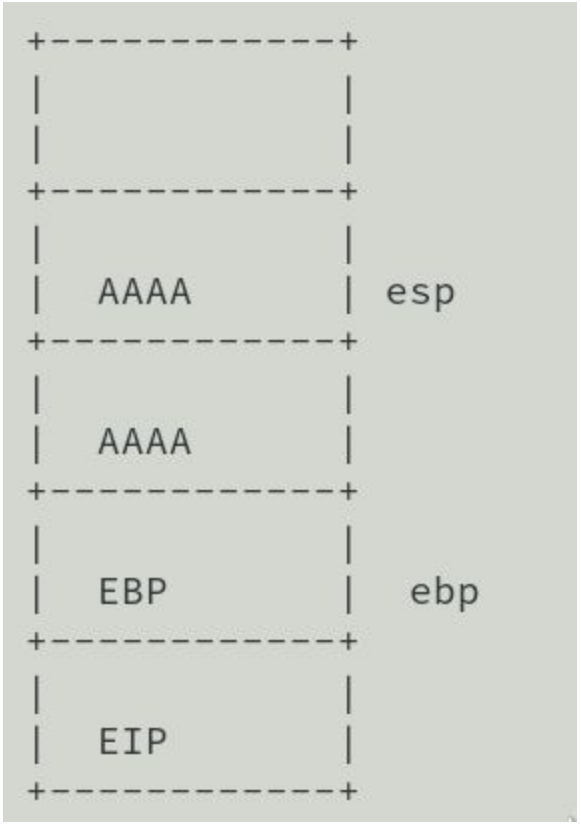
- What happens if we don’t have a win function?
  - We can upload our own program, and jump into it?
- payload = <win function> + <overwrite eip>
  - Where eip points back into our “win” function
- Functions are just assembly
  - Assembly are just bytes
    - We can send bytes to the program :)

What can our shellcode do

- Upload our own programs, run them
- execve(“**/bin/sh**”, NULL, NULL);
- Connect back
  - The shellcode connects back to us
  - Most exploits use this since most firewalls filter ingress (bindshell won’t work)
- Socket reuse
  - Finds the socket that was used to deliver the exploit and uses that
  - Usually requires more work than that other ones
- **Egghunter**
  - Small bit of shellcode that finds a larger payload (the egg)
  - An omelette egghunter finds multiple eggs and puts them together
- Download a second stage

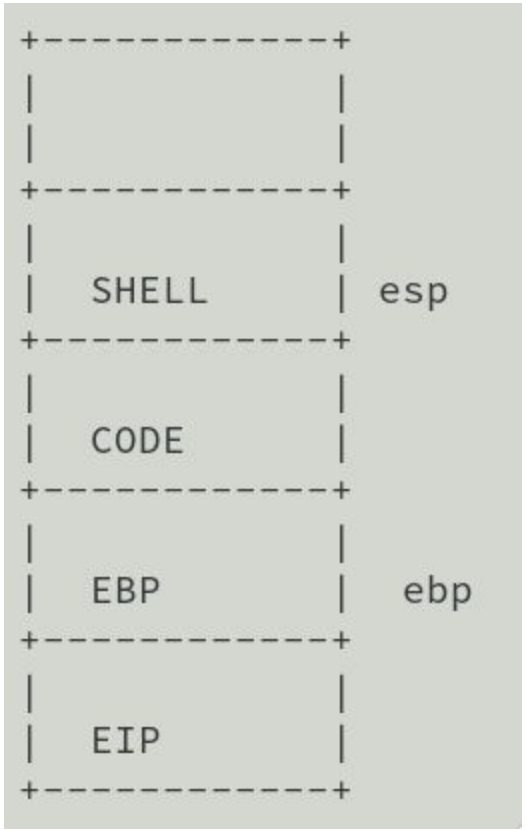
This is last week

- EIP = WIN



This is this week

- EIP = Base of shellcode



Shellcode has to be **Position Independent**

- Your shellcode won't originally know where it is in memory
- Can't **hardcode** memory addresses
- Everything has to be **relative**

Can find EIP with this stub:

```
    call stuff

stuff:

     pop eax  // eax now has eip
```

Example x86 shellcode

- char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

```
xor eax,eax
push eax
push 0x68732f2f
push 0x6e69622f          "/bin//sh" in ascii (why 2 slashes?)
mov ebx, esp


push eax
push ebx
mov ecx,esp              // Set ecx to [ "/bin//sh", NULL ]


mov al,0xb
int 0x80                 // execve /bin/sh
```

How do i get those magic bytes

- Write the ASM by hand, assemble using nasm, grab the bytes
  - hard for complex shellcode
- Write it in C, compile (probably with –static, likely with -Os) and then extract
  - With grep
- pwntools + python = win
  - asm()

Calling functions with assembly

- If you know the address, jumping to a function is trivial

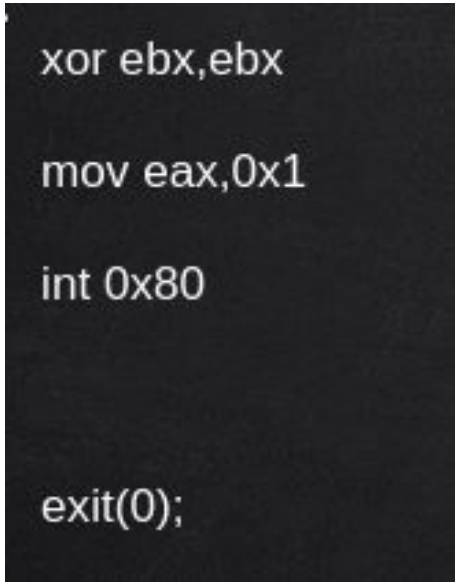
- Can pass in arguments with push <VALUE>
- Sometimes this isn’t possible (don’t know the address)

System calls

- We need to interact with the operating system (e.g. open files/exec other programs)
  - This is done with syscalls
- Syscalls are numbered
  - sys_exit == 1 (on x86), sys_fork == 2, etc
- This number is put in **eax** and then triggered either by an **interrupt** (int 0x80)
  - Arguments are passed to syscall through registers
    - 1-ebx
    - 2-ecx
    - 3-edx
    - 4-esx
    - 5-edi
- http://cgi.cse.unsw.edu.au/~z5164500/syscall/ - syscall tables exist

SYS_EXIT

sys_exit takes 1 **argument** (the exit status code) in ebx



Strings

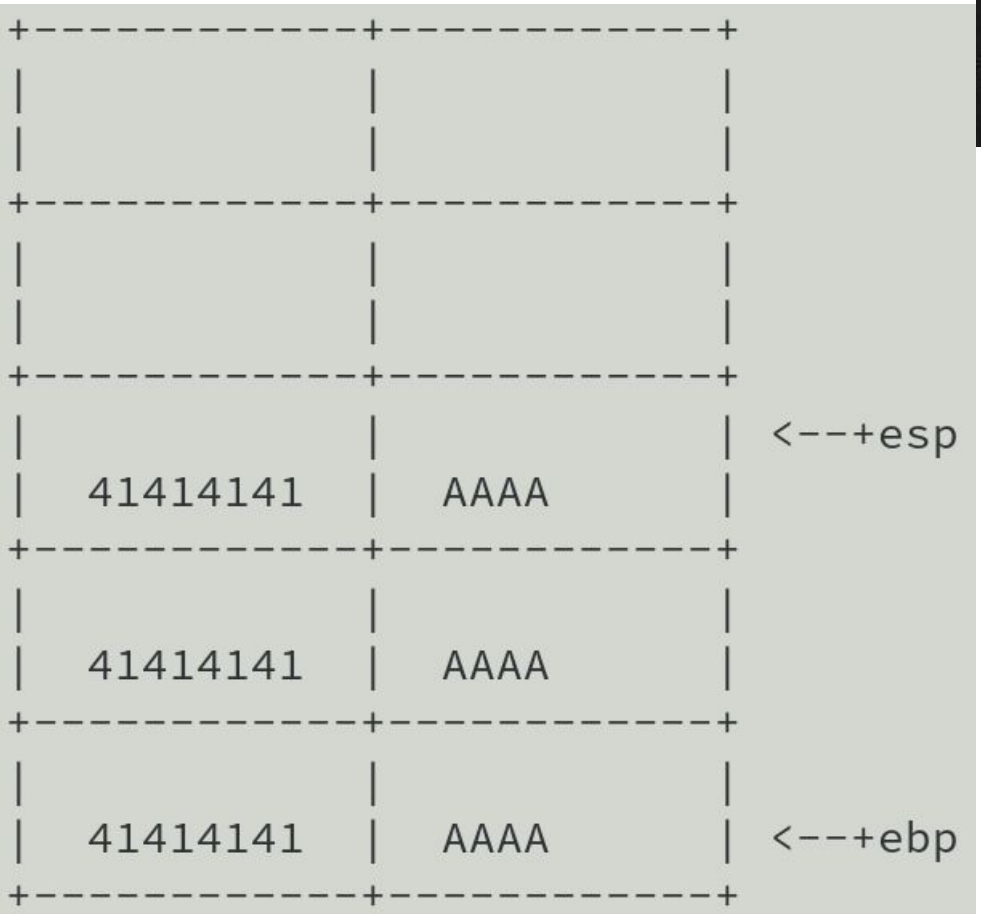
Sometimes can be useful to have strings as input to functions

- Ie: to call execve(“/bin/sh”), you need the string…
  - You need a pointer to the string?
    - But your shellcode is Position **independent**
- Two main ways to combat this

1)
   a) You can use the stack without knowing its address… pop/push
   b) Can put strings onto the stack, and then take value of esp to get the address of the string

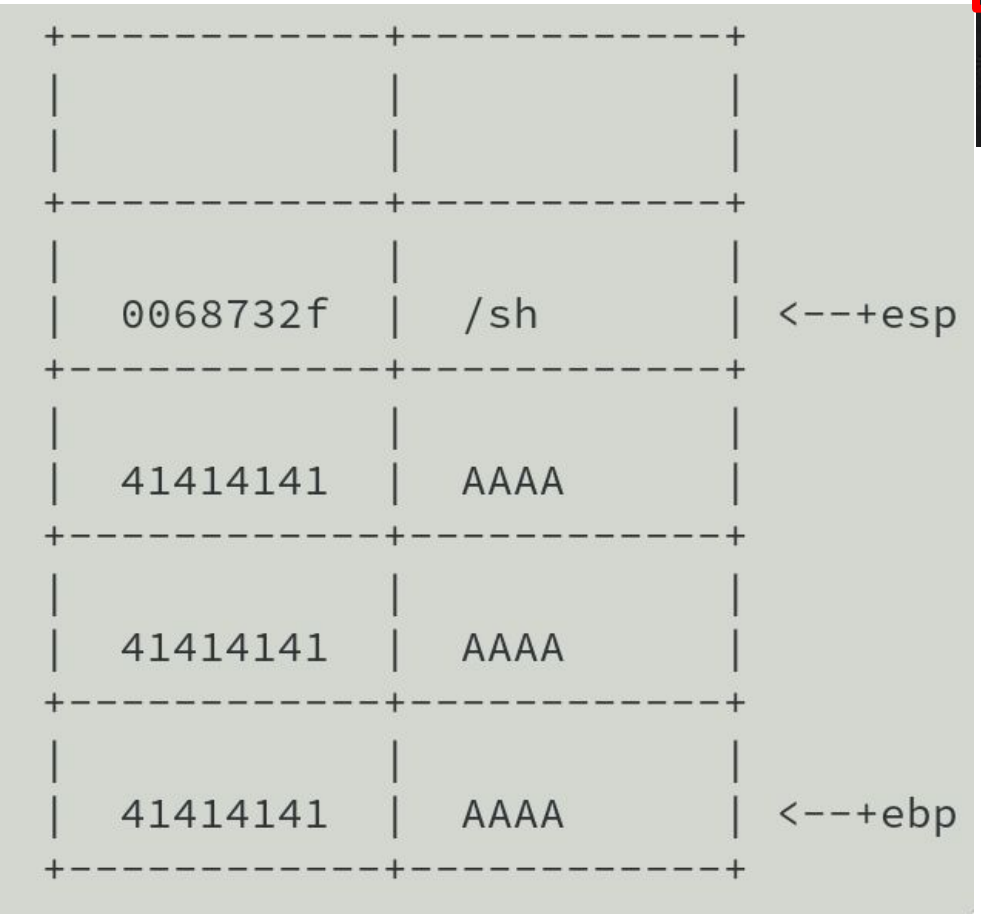
2)
   a) Add the string to the end of your shellcode
   b) Offset from the address of your shellcode

Example string usage



push 0x0068732f	push “/sh\x00”
push 0x6e69622f	push “/bin”
mov ebx, esp	esp -> /bin/sh\x00

Example string usage



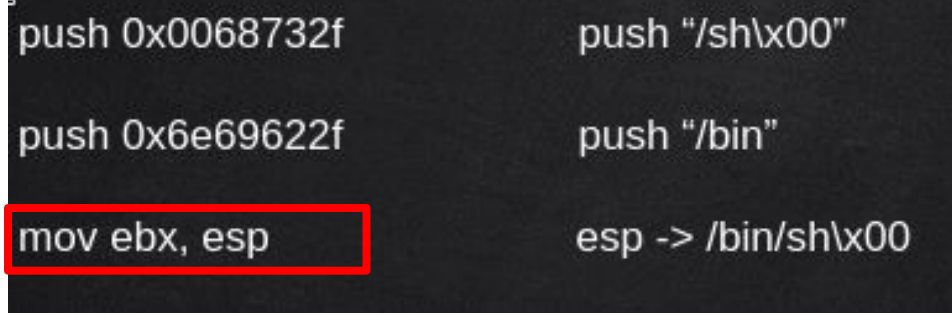
push 0x0068732f push “/sh\x00”

push 0x6e69622f push “/bin”

mov ebx, esp esp -> /bin/sh\x00

Example string usage

```
+------------+------------+
|            |            |
|  6e69622f  |  /bin      | <--+esp
+------------+------------+
|            |            |
|  0068732f  |  /sh       |
+------------+------------+
|            |            |
|  41414141  |  AAAA      |
+------------+------------+
|            |            |
|  41414141  |  AAAA      |
+------------+------------+
|            |            |
|  41414141  |  AAAA      | <--+ebp
+------------+------------+
```



Now **esp** points to the null terminated string -> “/bin/sh”.

Copy this address into another register...

NoPsLeD

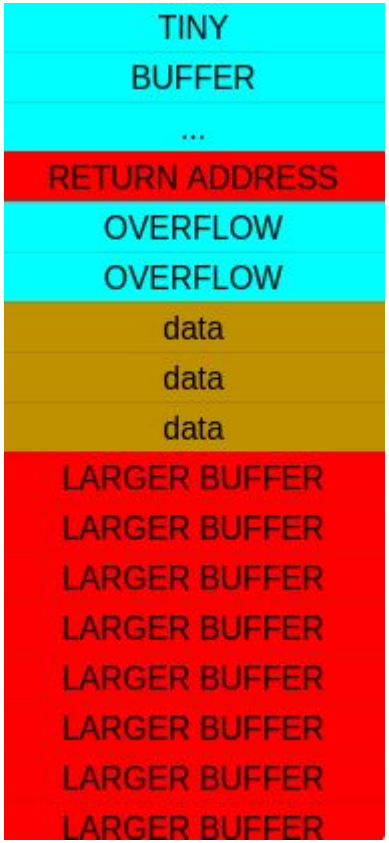
- Say you have 20 bytes of shellcode – when you specify an address, you need to land exactly at the start to execute the shellcode. – there is no margin for error.
- What if the program lets you copy in 10 megs? Seems silly that you still have to land it right on the nose.
  - 0x90 – NOP – does nothing
- **NOPNOPNOPNOPNOP * 1 million**+20 **bytes of shellcode = win**
- What if you don't know exactly where your code is, but you know the general area of it
- Some firewalls block NOP*10000, but just replace with other useless instructions, like `**xchg eax, eax**`
- The lots of NOPS thing is called a sled. **NOP SLED**.

More advanced use of shellcode

- **Egghunter**
  - If we only have a small space for our shellcode we can create an egghunter, which searches memory for a signature, and then jmp's to it
  - Omelette
- May need to fix up memory permissions (especially in 2019) or map a new page, i.e. **mprotect**()
- **Syscall Proxy** (run programs on local machine, execute syscalls on remote)
- **Mosdef** (a python compiler for remote hosts)
  - https://www.blackhat.com/presentations/win-usa-04/bh-win-04-aitel.pdf
  - Read if interested!!!

Egg hunter

- Useful when
  - The program has two buffers, one large one tiny
  - Large buffer isn't overflowable.
  - The tiny buffer is the one that overflows
    - ie: buffer of size 16, reads in 24 bytes
  - Not enough size in tiny buffer for a complete payload
- In the tiny buffer
  - Put shellcode that loops through all of memory, looking for the large buffer
  - Then execute it
- In the big buffer
  - Put a signature at the top
    - ie: (0xABCDEF1234)
  - Put your normal shellcode



Wait… How can I execute shellcode if my stack isn't executable???

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
 0x8048000  0x8049000 r--p     1000 0      /home/honeypot/moreappropiatename/comp6447/2020/lectures/3/control/complexloop
 0x8049000  0x804a000 r-xp     1000 1000   /home/honeypot/moreappropiatename/comp6447/2020/lectures/3/control/complexloop
 0x804a000  0x804b000 r--p     1000 2000   /home/honeypot/moreappropiatename/comp6447/2020/lectures/3/control/complexloop
 0x804b000  0x804c000 r--p     1000 2000   /home/honeypot/moreappropiatename/comp6447/2020/lectures/3/control/complexloop
 0x804c000  0x804d000 rw-p     1000 3000   /home/honeypot/moreappropiatename/comp6447/2020/lectures/3/control/complexloop
0xf7e00000 0xf7e1d000 r--p    1d000 0      /usr/lib/libc-2.31.so
0xf7e1d000 0xf7f47000 r-xp   12a000 1d000  /usr/lib/libc-2.31.so
0xf7f47000 0xf7fad000 r--p    66000 147000 /usr/lib/libc-2.31.so
0xf7fad000 0xf7fae000 ---p     1000 1ad000 /usr/lib/libc-2.31.so
0xf7fae000 0xf7fb0000 r--p     2000 1ad000 /usr/lib/libc-2.31.so
0xf7fb0000 0xf7fb2000 rw-p     2000 1af000 /usr/lib/libc-2.31.so
0xf7fb2000 0xf7fb6000 rw-p     4000 0
0xf7fcc000 0xf7fd0000 r--p     4000 0      [vvar]
0xf7fd0000 0xf7fd2000 r-xp     2000 0      [vdso]
0xf7fd2000 0xf7fd3000 r--p     1000 0      /usr/lib/ld-2.31.so
0xf7fd3000 0xf7ff1000 r-xp    1e000 1000   /usr/lib/ld-2.31.so
0xf7ff1000 0xf7ffc000 r--p     b000 1f000  /usr/lib/ld-2.31.so
0xf7ffc000 0xf7ffd000 r--p     1000 29000  /usr/lib/ld-2.31.so
0xf7ffd000 0xf7ffe000 rw-p     1000 2a000  /usr/lib/ld-2.31.so
0xfffdd000 0xffffe000 rw-p    21000 0      [stack]
```

NX

- NX = Non executable stack
  - A memory protection that targets shellcode developers!!
- Like other memory protections, won’t be enabled this week (will be soon though!)
-

NX - How to deal with it irl

```
NAME
       mprotect – set protection of memory mapping

SYNOPSIS
       #include <sys/mman.h>

       int mprotect(void *addr, size_t len, int prot);
```

- Real life isn't as nice as me
- Say hello to **mprotect**
  - The prot argument should be either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE, and PROT_EXEC.
- We already control the stack.
  - We already know we can call functions/syscalls
    - We already know how to pass arguments into a function
- We can just make our chunk of memory executable.
  - We will learn more about how to do this in week 6!