```
 ___________________________________.
|;;|                          |;;||
|[]|--------------------------|[]||
|;;|                          |;;||
|;;|    -------------         |;;||
|;;|    | COMP6447 |          |;;||
|;;|    -------------         |;;||
|;;|    intel x64             |;;||
|;;|                          |;;||
|;;|__________________________|;;||
|;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;||
|;;;;;;_______________________ ;;;;;||
|;;;;;|  ___                 |;;;;;||
|;;;;;| |;;;|                |;;;;;||
|;;;;;| |;;;|                |;;;;;||
|;;;;;| |;;;|                |;;;;;||
|;;;;;| |;;;|                |;;;;;||
|;;;;;| |___|                |;;;;;||
\_____|______________________|______||
 ~~~~~^^^^^^^^^^^^^^^^^^~~~~~~
```

further reversing

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

But first

How were first real wargames?

Will run in-person help session during some weeks, to walk through solutions of wargames if people want.

Potentially after lecture, more details later

Some admin

Bsides

Wednesday 6pm tute will be cancelled due to tutor availability. Please go to a different tutorial (See WebCMS).

Next Monday is a public holiday. Lecture is optional, I will still give it in person and record it + release it before tutorials. watch the lecture before your tutorial.

Intel x86 documentation has more pages than the 6502 has transistors

x86 is a family of instruction set architectures[a] initially developed by Intel

x86-64 (AMD64) is the 64 bit variant of x86.

x86 is commonly used to refer to the 32 bit variant

x86 is different to arm, mips, etc.

- Programs in this course will only work on x86 family of CPUs (most pc's other than new macs)

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);
}
```

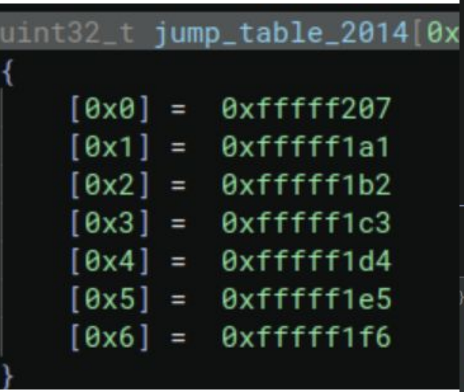
- Loops are just conditionals with a **goto**
  - Do the comparison
  - If false jump to end of loop
  - Else do stuff in Loop then jump back to top
- Loops are usually compiled backwards (easier?)
  - while(x) {} -> if (x) { do {...} while(x) }

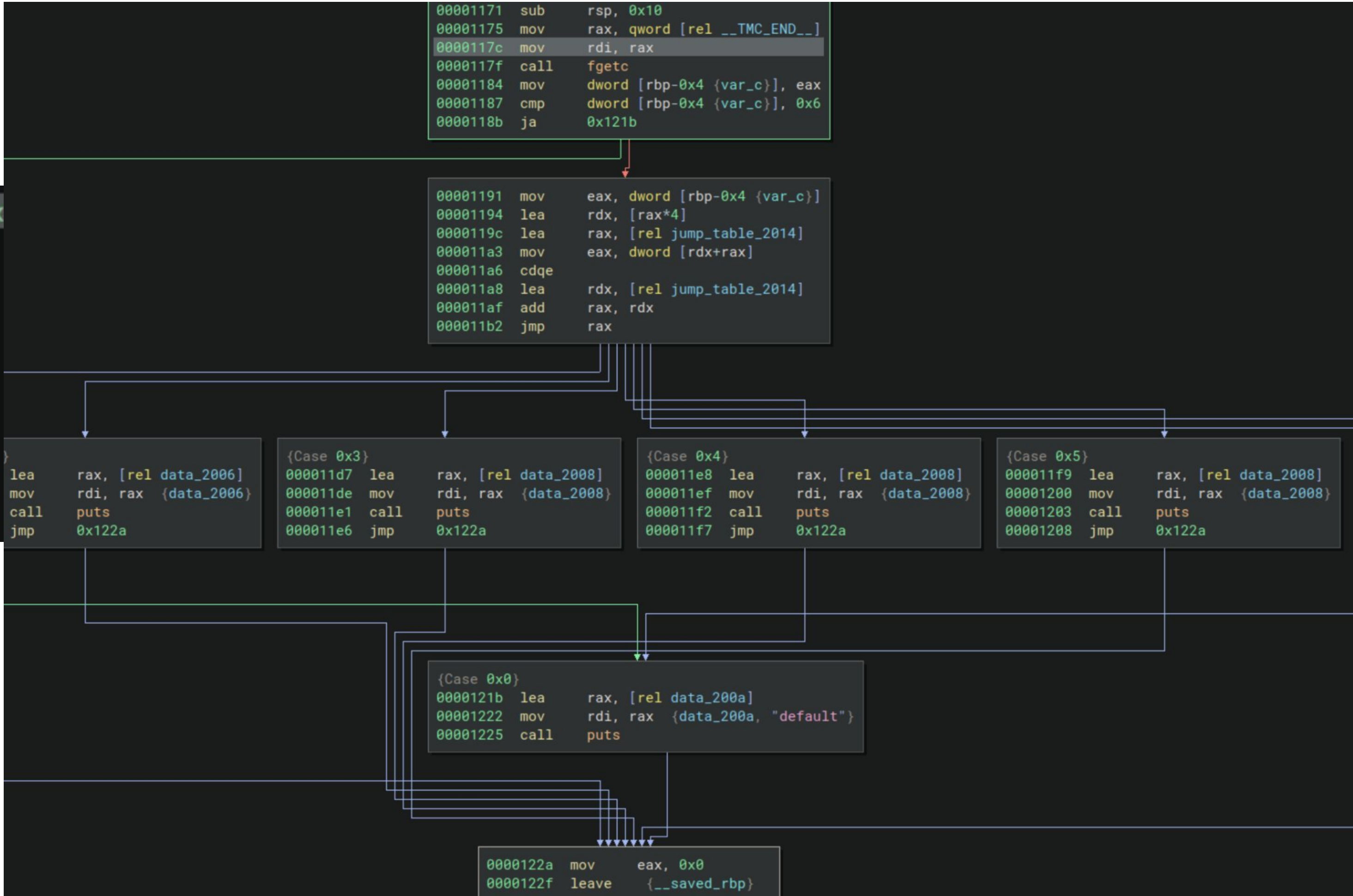
> Loop demo

What about switch statements

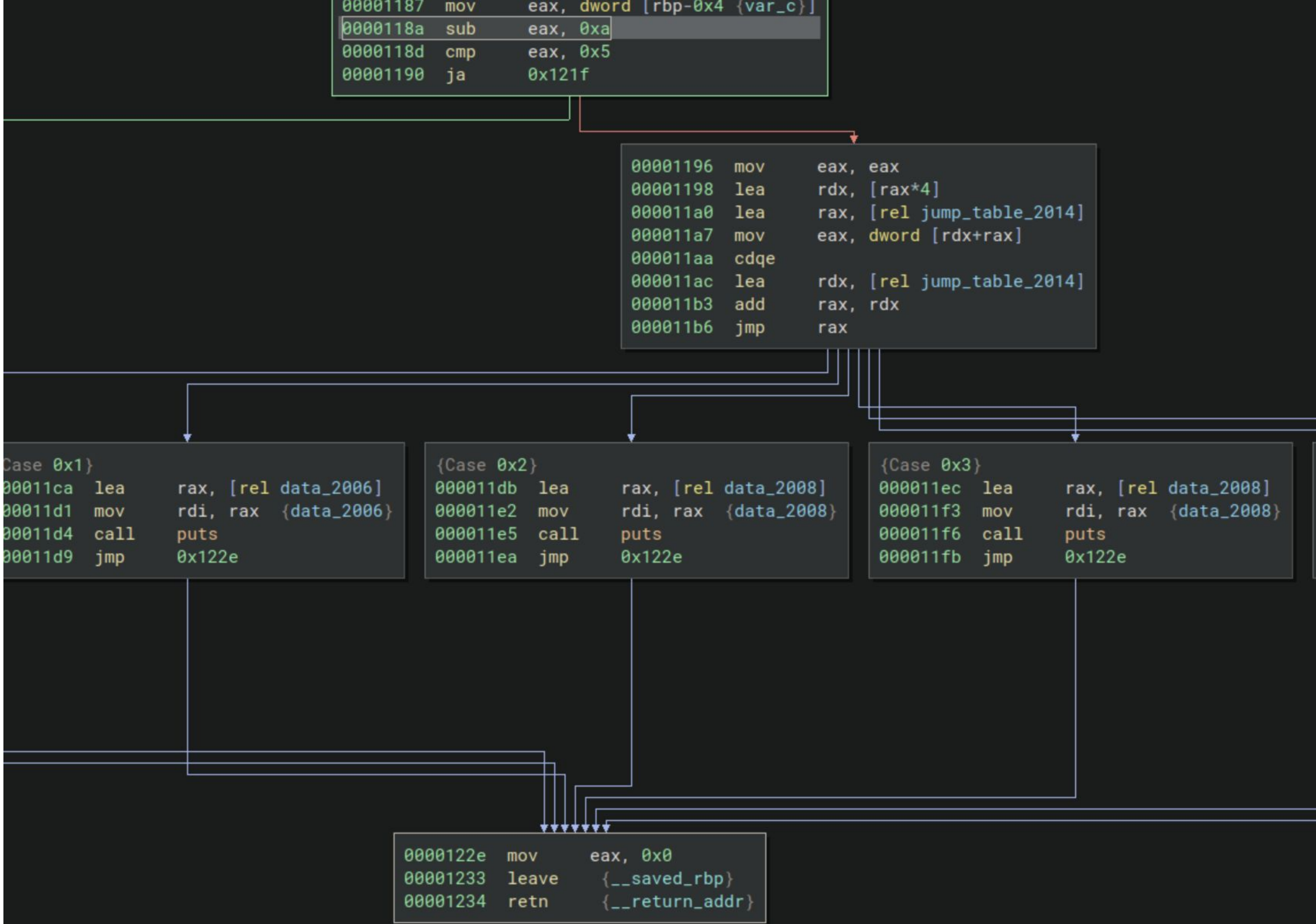
- Several cases
  - simple case of small close together numbers. ie: x = 1 or 2 or 3
  - Simple case of larger close together numbers. ie: x = 4 or 5 or 7 or 8 or 9
  - Complex case of random things

Case 1

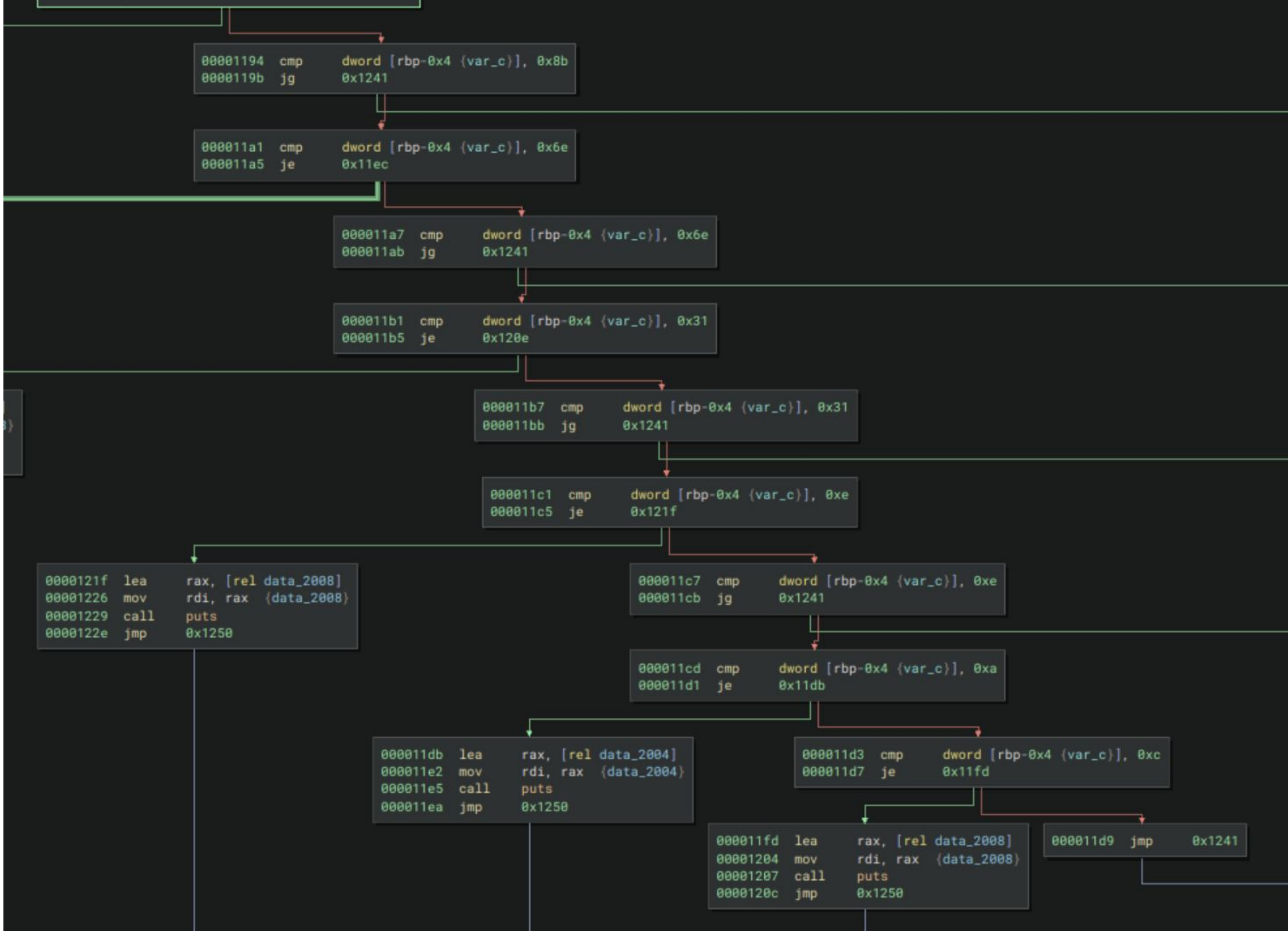

uint32_t jump_table_2014[0x
{
[0x0] = 0xffffff207
[0x1] = 0xfffff1a1
[0x2] = 0xfffff1b2
[0x3] = 0xfffff1c3
[0x4] = 0xfffff1d4
[0x5] = 0xfffff1e5
[0x6] = 0xfffff1f6
}


00001171 sub rsp, 0x10
00001175 mov rax, qword [rel __TMC_END__]
0000117c mov rdi, rax
0000117f call fgetc
00001184 mov dword [rbp-0x4 {var_c}], eax
00001187 cmp dword [rbp-0x4 {var_c}], 0x6
0000118b ja 0x121b
00001191 mov eax, dword [rbp-0x4 {var_c}]
00001194 lea rdx, [rax*4]
0000119c lea rax, [rel jump_table_2014]
000011a3 mov eax, dword [rdx+rax]
000011a6 cdqe
000011a8 lea rdx, [rel jump_table_2014]
000011af add rax, rdx
000011b2 jmp rax
lea rax, [rel data_2006]
mov rdi, rax {data_2006}
call puts
jmp 0x122a
{Case 0x3}
000011d7 lea rax, [rel data_2008]
000011de mov rdi, rax {data_2008}
000011e1 call puts
000011e6 jmp 0x122a
{Case 0x4}
000011e8 lea rax, [rel data_2008]
000011ef mov rdi, rax {data_2008}
000011f2 call puts
000011f7 jmp 0x122a
{Case 0x5}
000011f9 lea rax, [rel data_2008]
00001200 mov rdi, rax {data_2008}
00001203 call puts
00001208 jmp 0x122a
{Case 0x0}
0000121b lea rax, [rel data_200a]
00001222 mov rdi, rax {data_200a, "default"}
00001225 call puts
0000122a mov eax, 0x0
0000122f leave {__saved_rbp}

Case 2


00001187 mov eax, dword [rbp-0x4 {var_c}]
0000118a sub eax, 0xa
0000118d cmp eax, 0x5
00001190 ja 0x121f
00001196 mov eax, eax
00001198 lea rdx, [rax*4]
000011a0 lea rax, [rel jump_table_2014]
000011a7 mov eax, dword [rdx+rax]
000011aa cdqe
000011ac lea rdx, [rel jump_table_2014]
000011b3 add rax, rdx
000011b6 jmp rax
ase 0x1}
00011ca lea rax, [rel data_2006]
00011d1 mov rdi, rax {data_2006}
00011d4 call puts
00011d9 jmp 0x122e
{Case 0x2}
000011db lea rax, [rel data_2008]
000011e2 mov rdi, rax {data_2008}
000011e5 call puts
000011ea jmp 0x122e
{Case 0x3}
000011ec lea rax, [rel data_2008]
000011f3 mov rdi, rax {data_2008}
000011f6 call puts
000011fb jmp 0x122e
0000122e mov eax, 0x0
00001233 leave {__saved_rbp}
00001234 retn {__return_addr}

Case 3


00001194 cmp dword [rbp-0x4 {var_c}], 0x8b
0000119b jg 0x1241
000011a1 cmp dword [rbp-0x4 {var_c}], 0x6e
000011a5 je 0x11ec
000011a7 cmp dword [rbp-0x4 {var_c}], 0x6e
000011ab jg 0x1241
000011b1 cmp dword [rbp-0x4 {var_c}], 0x31
000011b5 je 0x120e
000011b7 cmp dword [rbp-0x4 {var_c}], 0x31
000011bb jg 0x1241
000011c1 cmp dword [rbp-0x4 {var_c}], 0xe
000011c5 je 0x121f
0000121f lea rax, [rel data_2008]
00001226 mov rdi, rax {data_2008}
00001229 call puts
0000122e jmp 0x1250
000011c7 cmp dword [rbp-0x4 {var_c}], 0xe
000011cb jg 0x1241
000011cd cmp dword [rbp-0x4 {var_c}], 0xa
000011d1 je 0x11db
000011db lea rax, [rel data_2004]
000011e2 mov rdi, rax {data_2004}
000011e5 call puts
000011ea jmp 0x1250
000011d3 cmp dword [rbp-0x4 {var_c}], 0xc
000011d7 je 0x11fd
000011fd lea rax, [rel data_2008]
00001204 mov rdi, rax {data_2008}
00001207 call puts
0000120c jmp 0x1250
000011d9 jmp 0x1241

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 rax, **qword** ptr [rsp]

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

mov ax, **word** ptr [rbp-0x14]

mov al, **byte** ptr [rsp]

mov **ax**, [rsp]

mov **al**, [rsp]

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:
000011c7  endbr64
000011cb  push    rbp {__saved_rbp}
000011cc  mov     rbp, rsp {__saved_rbp}
000011cf  sub     rsp, 0x10
000011d3  mov     edi, 0x10
000011d8  call    malloc
000011dd  mov     qword [rbp-0x8 {var_10}], rax
000011e1  mov     rax, qword [rbp-0x8 {var_10}]
000011e5  mov     qword [rax], 0x0
000011ec  mov     rax, qword [rbp-0x8 {var_10}]
000011f0  mov     dword [rax+0x8], 0x0
000011f7  mov     rax, qword [rbp-0x8 {var_10}]
000011fb  pxor    xmm0, xmm0
000011ff  movss   dword [rax+0xc], xmm0  {0x0}
00001204  mov     rax, qword [rbp-0x8 {var_10}]
00001208  leave    {__saved_rbp}
00001209  retn     {__return_addr}
```

```
set_first_var:
00001169  endbr64
0000116d  push    rbp {__saved_rbp}
0000116e  mov     rbp, rsp {__saved_rbp}
00001171  mov     qword [rbp-0x8 {var_10}], rdi
00001175  mov     rax, qword [rbp-0x8 {var_10}]
00001179  mov     rax, qword [rax]
0000117c  mov     dword [rax], 0x6d616441
00001182  mov     byte [rax+0x4], 0x0
00001186  nop
00001187  pop     rbp {__saved_rbp}
00001188  retn     {__return_addr}
```

```
set_second_var:
00001189  endbr64
0000118d  push    rbp {__saved_rbp}
0000118e  mov     rbp, rsp {__saved_rbp}
00001191  mov     qword [rbp-0x8 {var_10}], rdi
00001195  mov     dword [rbp-0xc {var_14}], esi
00001198  mov     rax, qword [rbp-0x8 {var_10}]
0000119c  mov     edx, dword [rbp-0xc {var_14}]
0000119f  mov     dword [rax+0x8], edx
000011a2  nop
000011a3  pop     rbp {__saved_rbp}
000011a4  retn     {__return_addr}
```

```
set_third_var:
000011a5  endbr64
000011a9  push    rbp {__saved_rbp}
000011aa  mov     rbp, rsp {__saved_rbp}
000011ad  mov     qword [rbp-0x8 {var_10}], rdi
000011b1  movss   dword [rbp-0xc {var_14}], xmm0
000011b6  mov     rax, qword [rbp-0x8 {var_10}]
000011ba  movss   xmm0, dword [rbp-0xc {var_14}]
000011bf  movss   dword [rax+0xc], xmm0
000011c4  nop
000011c5  pop     rbp {__saved_rbp}
000011c6  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**i**/next**i**
  - 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

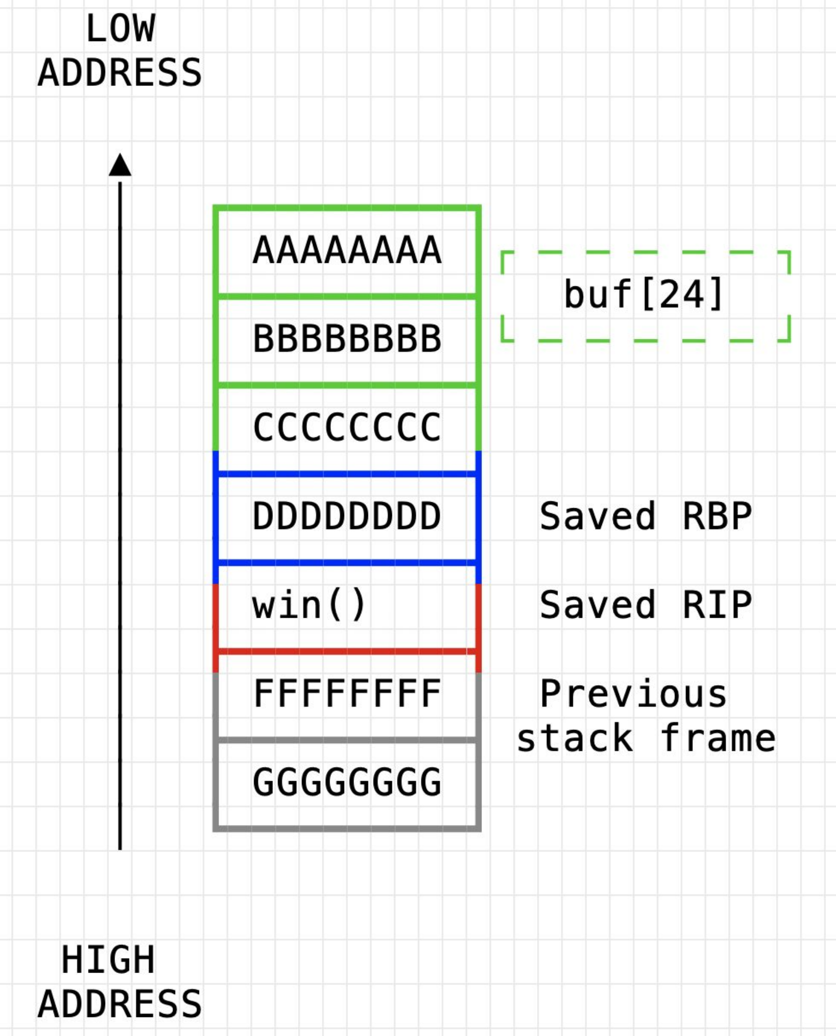
- Historically, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability.
- Shellcode is commonly written in machine code.
- After a memory corruption based exploitation
    - You need someway of executing code
        - **Shellcode**
        - ROP/RET2CODE/RET2LIBC

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 rip>
  - Where eip points back into our “win” function
- Functions are just assembly
  - Assembly are just bytes
    - We can send bytes to the program :)

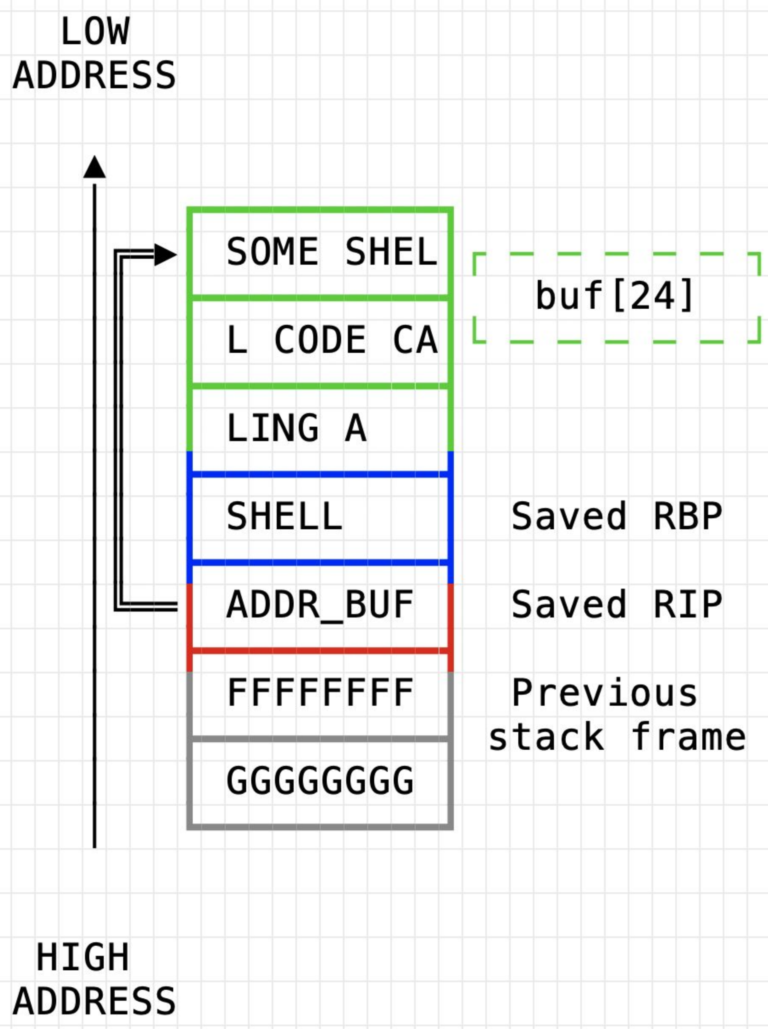
This is last week

- EIP = WIN()



This is this week

- EIP = Base of shellcode



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

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 RIP with this stub:

```
   call stuff                                  ← This call is a relative call. (call <next instruction>)

stuff:

    pop RAX  // rax now has rip
```

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";

How do i get those magic bytes

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

Calling functions with assembly

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

```
call 0x7ffe3129d3c
```

- Can pass in arguments with mov register, <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_write == 1 (on x64), sys_open == 2, etc
- This number is put in **eax** and then triggered either by an **interrupt** (syscall)
  - Arguments are passed to syscall through registers
    - 1-RDI
    - 2-RSI
    - 3-RDX
    - 4-R10
    - 5-R8
- https://x64.syscall.sh/ - syscall tables exist

SYS_EXIT

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

```
xor rdi, rdi
mov rax, 60
syscall
```

```
exit(0);
```

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 rsp to get the address of the string

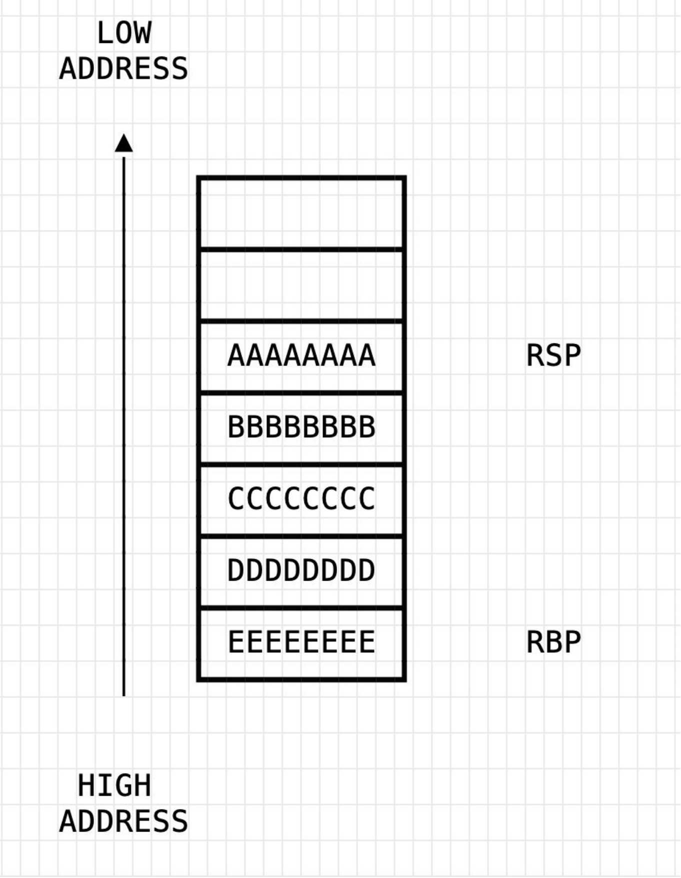
2)

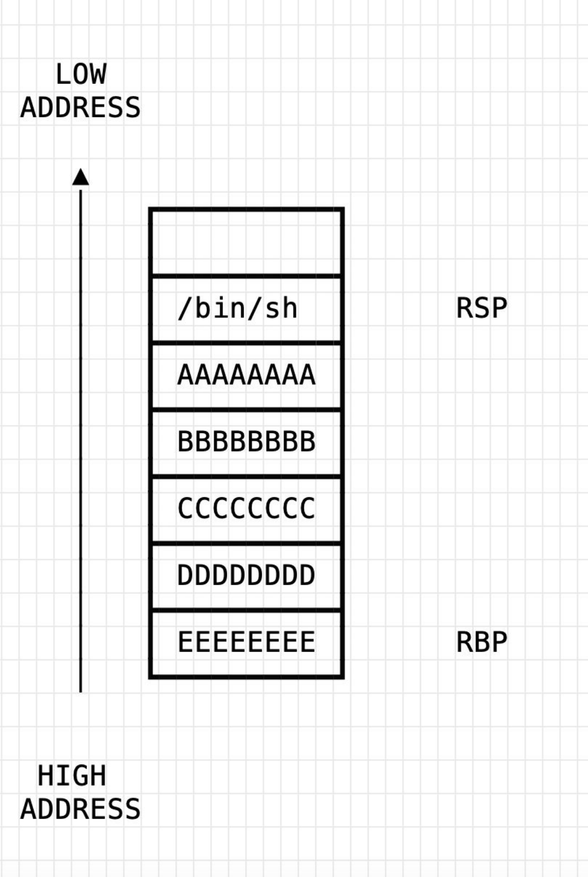
   a) Add the string to the end of your shellcode
   b) Offset from the address of your shellcode

Example string usage

```
mov rax, 0x0068732f6e69622f    mov "/bin/sh\x00"
push rax                       rsp -> "/bin/sh\x00"
mov rdi, rsp                   rdi -> "/bin/sh\x00"
```

```
mov rax, 0x0068732f6e69622f      mov "/bin/sh\x00"
push rax                         rsp -> "/bin/sh\x00"
mov rdi, rsp                     rdi -> "/bin/sh\x00"
```





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

Copy this address into another register...

NoPsLeD

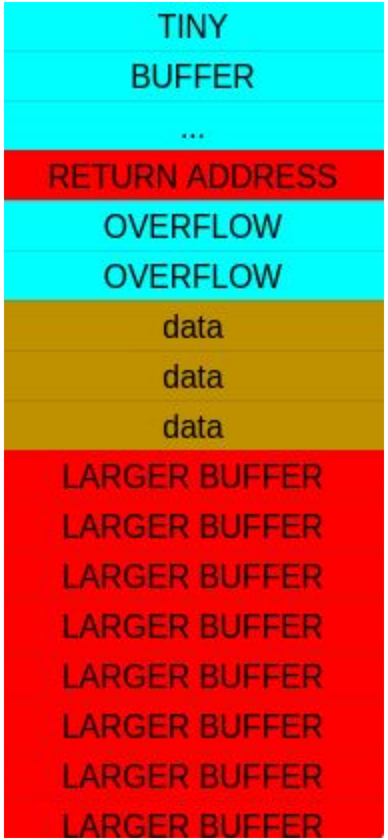
- Say you have 20 bytes of shellcode – when you specify an address, you need to land exactly at the start to execute the shellcode. – there is no margin for error.
- What if the program lets you copy in 10 megs? Seems silly that you still have to land it right on the nose.
    - 0x90 – NOP – does nothing
- **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 2020+) 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??

- Stacks are frequently marked as non-executable (As are loads of other regions)
- Some programs (like javascript engines) still often require executable stacks for performance, though, and things like java definitely do and will for the near future
- Anyway – what can we do if cannot execute shellcode? …
- 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!