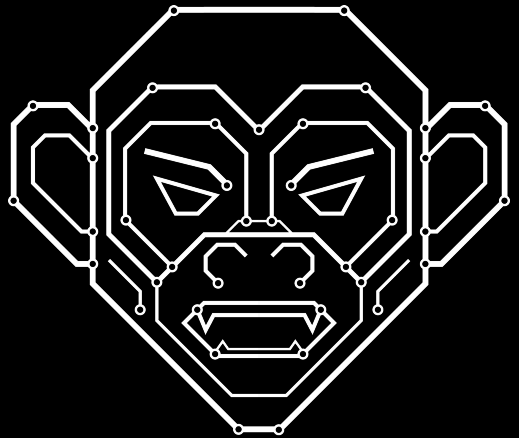
AFNOM Session 5

Binary Exploitation/PWN



A·F·N·O·M

Society Recap

- We learn **hacking** skills and do **CTFs**
- No membership — just **turn up**!
- Come to the **pub** after sessions + occasional **events** (films / games)
- Ask **questions**!! We're all here to learn from each other :-)

AFNOM Values

- **Inclusive** — everyone is welcome, regardless of race, ethnicity, gender, gender identity, sexual orientation, age, class, physical ability, nationality, and text-editor preference.
- **Supportive** — we're not here to compete with each other; we've all got things to learn and teach.
- **Ambitious** — we all want to advance our cybersecurity skills.
- **Respectful** — let's be kind and humble.

Preface

- Some things we cover could be interpreted as or used immorally or illegally
- We are here to learn cool stuff and have fun!

“Don’t do crimes”
- Dr Ian Batten

- Good lecturer
- Sound advice
- We are the **ethical** hacking society

“Don’t be evil”
- Google

- Somewhat ironic
- They stopped using it in 2015
- But we are not multi conglomerate mega empire so we can reserve the right to morals

Housekeeping: **AI usage**

- We **do not encourage** the use of **generative AI** in sessions
- However we will not stop you from using them as a **tool**
  - Particularly if used as a **learning method** (as much as that is possible)
- **Vibesolving** (i.e. "claude solve this") is **strongly discouraged**
- Asking for help without **sincerely attempting** it yourself first will not be fruitful
- Our aim and purpose here is to **have fun** learning **hacking skills**, which we **do not** believe is achievable by just getting the AI to do it for you!
- Give it a **proper go**, **look stuff up**, **ask people** around you, and **have fun** with the process of hacking :)

Upcoming schedule

- **Today** — Binary Exploitation/PWN
- **18th November** — Cryptography
- **25th November** — Reverse Engineering
- **2nd December** — Network Forensics
- **Future** — Even more hacking!… Any suggestions? Let us know!

What is PWN?

- Exploiting the **functionality** of a binary (**native program**) to circumvent protections, gain access/permissions, etc.
- Redirecting **control flow**, running your own code, or dropping into a **shell**
- One of the things you might imagine when you think of **‘hacking’**

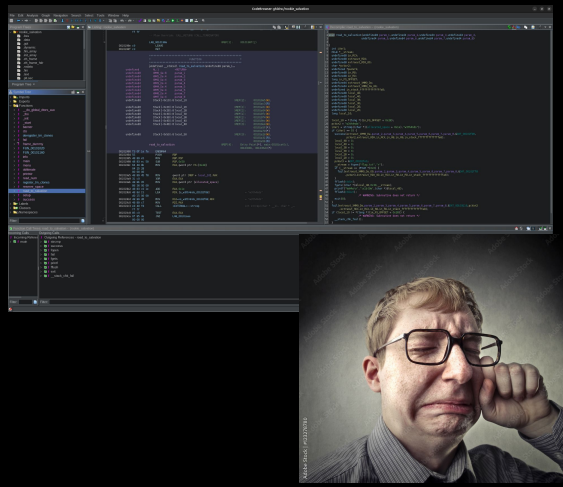
Expectation



“01110101011010100 1…..Im in”

Reality





Today's Session

- Intro to **PWN**
- Deep and **technical** category
- Want to get you started with enough to **give it a go!**

- **Covered today**
  - **Assembly** & the **stack** - how code runs at a low level (on the **x86** architecture)
  - Smashing the stack
  - Taking control
  - …and a little more!

- **Not today**
  - Other **architectures** (ARM, RISCV, etc)
  - How to do harder stack attacks (**ROP**)
  - The **heap** + heap attacks 🤮
  - Maybe in a **future session** though!

Kickstart - the basics

Assembly

- Computers can’t just run **code** raw
- It needs to be processed and converted (**compiled**) into **machine code** instructions that the computer knows
- Machine code **interpreted** and **represented** as **assembly** code
- Changes between **architectures**
- Lots of **different** architectures - only looking at **x86** today!

PowerPC arm



RISC-V

```
void cool_func() {
  int foo = 6942067;
  char bar[12] = “AFNOM RULEZ”;
  other_func(1,2);
}
```




```
00011173    int32_t cool_func()

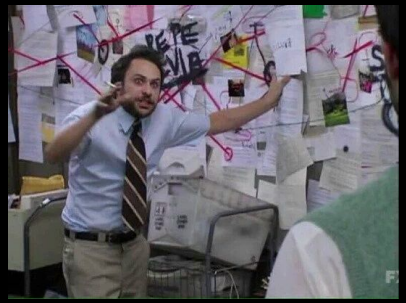
00011173  55              push    ebp {__saved_ebp}
00011174  89e5            mov     ebp, esp {__saved_ebp}
00011176  83ec10          sub     esp, 0x10
00011179  c745fc73ed6900  mov     dword [ebp-0x4 {var_8}], 6942067
00011180  c745f041464e4f  mov     dword [ebp-0x10], 'AFNO'
00011187  c745f44d205255  mov     dword [ebp-0xc {var_10}], 'M RU'
0001118e  c745f84c455a00  mov     dword [ebp-0x8 {var_c}], 'LEZ\x00'
00011195  6a02            push    0x2 {var_18}
00011197  6a01            push    0x1 {var_1c}
00011199  e8cfffffff      call    other_func
0001119e  83c408          add     esp, 0x8
000111a1  90              nop
000111a2  c9              leave   {__saved_ebp}
000111a3  c3              retn    {__return_addr}
```

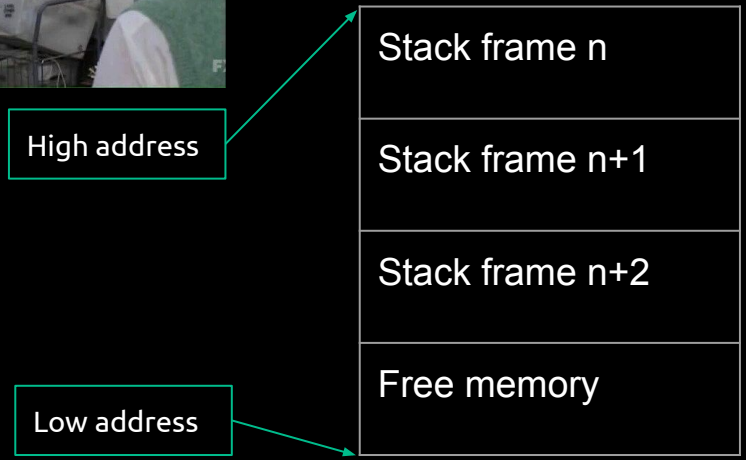
Kickstart - the basics

The stack

- Section of memory used for storing **local variables**, **working data**, etc.
- Each function gets its own **‘frame’**
- Frame **size** calculated at **compile time**
- Grows **down** memory - starts at higher memory address - frames are stacked at **decreasing** addresses
- However, frames fill ‘up’ - with the **higher address** often being shown the **bottom** of the frame

- Sometimes stack stuff is shown **low->high**, sometimes **high->low**
- Important to **understand** and identify this to avoid **confusion**!



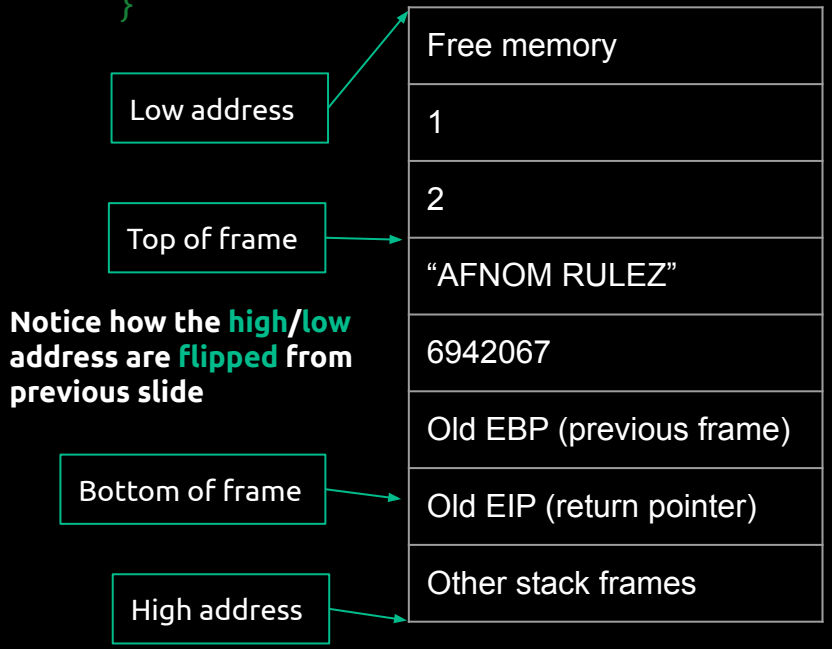


Stack Layout - Variables, parameters and control flow

- When a function is called, a **new frame** is created
- **Return address** (next instruction in caller function) and **previous frame pointer** are at the bottom of the frame
- Local variables/data written to frame from **high to low**
- Calling conventions
  - **32-bit**: Arguments are pushed to the **stack**
  - **64-bit**: Arguments put in registers: **RDI, RSI, RDX, RCX, R8, R9**, and then **stack**

All examples are **32-bit** unless specified otherwise!

```
void cool_func() {
  int foo = 6942067;
  char bar[12] = "AFNOM RULEZ";
  other_func(1,2);
}
```



Notice how the high/low address are flipped from previous slide

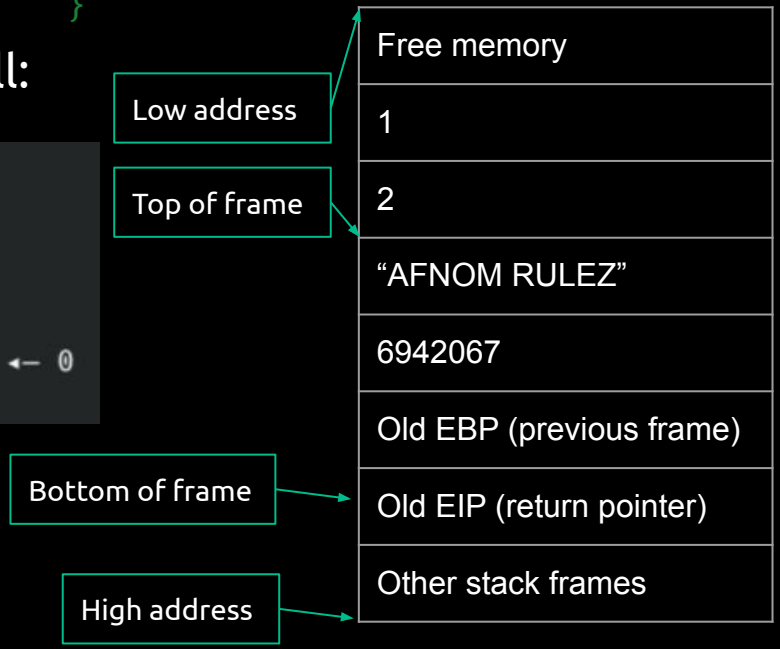
Stack Layout - Variables, parameters and control flow

```
void cool_func() {
  int foo = 6942067;
  char bar[12] = “AFNOM RULEZ”;
  other_func(1,2);
}
```

Screenshot from **GDB** just before **other_func** call:

```
 esp 0xffffcec8 ◂— 1
-014 0xffffcecc ◂— 2
-010 0xffffced0 ◂— 'AFNOM RULEZ'
-00c 0xffffced4 ◂— 'M RULEZ'
-008 0xffffced8 ◂— 0x5a454c /* 'LEZ' */
-004 0xffffcedc ◂— 0x69ed73
 ebp 0xffffcee0 —▸ 0xffffcee8 —▸ 0xf7ffcca0 (_rtld_global_ro) ◂— 0
+004 0xffffcee4 —▸ 0x565561ca (main+18) ◂— nop
```

Note: 0x69ed73 = 6942067



Diving in - Common attacks

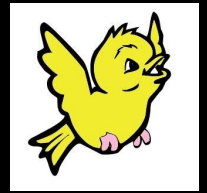
Covering today:

- **Buffer overflow**
  - Writing more data than intended to a memory location
- **Ret2Win**
  - Overwrite return pointer to jump to a specific function
- **Ret2LibC**
  - Ret2Win but specifically jumping into the C standard library
- **Leaking stack memory**
  - Finding memory addresses (stack base, code base, libc base, etc.)

Further attacks you might see:

- **Shellcode**
  - Inserting and executing your own code/assembly in your inputs
- **Return Oriented Programming (ROP)**
  - Reusing program code to build/execute your own

Protections (**checksec**)



- **RELRO** - RELocation Read-Only
  - Library function pointers cannot be overwritten
- **Canary** - Stack Canaries
  - A static value is placed before the stack frame to detect if the frame has been corrupted
  - Prevent buffer overflows
- **NX** / **DEP** - Not eXecutable / Data Execution Prevention
  - Memory pages cannot be loaded as both writable and executable
  - Prevent shellcode
- **PIE** - Position Independent Execution
  - The address of the binary base is randomized
  - Prevent ret2win/rop
- **ASLR** - Address Space Layout Randomisation
  - Randomizes the location of stack, heap, shared objects, etc.
  - Prevent ret2libc/rop



Smashing the stack - Buffer Overflow

- The **size** of values on the stack are **fixed** and **pre-calculated** based on the code (type, defined length, value)
- Inputting/moving data should use these bounds but aren't **enforced**
- Using functions like '**gets**' or even '**fgets**' with the wrong size can let you write past the end of the buffer
- Can **overwrite** other variables or even **return pointer**
- Basis for most PWN attacks

```
void get_input() {
  char buf[15];
  gets(buf);
}
```

Normal input:
"Hello World!"

Free memory
"Hello World!"
Old EBP (previous frame)
Old EIP (return pointer)
Other stack frames

Too long input:
"aaaaaaaaaaaaaaaaaaaaaa"

Free memory
"aaaaaaaaaaaaaaa"
"aaaa"
"aaaa"
Other stack frames

Return pointer overwritten
program crashes

Taking control - Ret2Win/Ret2LibC

- Use buffer overflows for more than just **crashing** the program
- Instead of just overflowing with random characters, why not give valid **return address**
- Can **change** what code runs after the current function!
- x86 is **little endian**, so need to be input in **reverse** order
- Not all hex values are **printable**, use a command like **whilehex** to input or **pwntools** with **python**

```
void win() {
  printf("win!\n");
}
void main() {
  char buf[10];
  gets(buf);
}
```

- Compile 64-bit with no canary + no PIE
- win() is at **0x401156**

Let's attack this program!
Step 1: overflow buffer + old frame pointer
Step 2: reverse + overwrite return pointer
Step 3: profit??
Payload - make sure to run with whilehex:
aaaaaaaaaaaaaaaa\x56\x11\x40\x00\x00\x00\x00\x00

Leaking stack values

Why do we need to leak values?

- **PIE** randomises the memory base of the program
- **ASLR** randomises the base of the stack
- **Canaries** are random values
- On a remote binary you can't attach a **debugger**
- The **printf** vulnerability can be used to leak many **stack values** (canarys, addresses, variables)
- Given there is enough space for a long enough **format string**

- Common leaking method uses '**printf**' calls that print user input
- Correct use: **printf(formatstring, userinput)**
- Vulnerable use: **printf(userinput)**
- Will treat your input as the **format string**

- Input string that would format arguments e.g. "**%p%p%p%p%p**"
- Then printf will start taking values for **arguments**
- **32-bit**: read stack values
- **64-bit**: registers then stack values

Complete freedom - Shellcode & ROP

Shellcode

- **No function** doing what you want to do already?
- Insert your own **assembly** code in your input
- Then set **return pointer** to the current **stack frame** to run it!
- **NX/DEP** generally prevents this as makes stack **non-executable** 😔
- But there are plenty of ways around this!

Return Oriented Programming (ROP)

- Similar to shellcode, build your own **assembly code**
- Build out of '**gadgets**' - **small blocks** of instructions ending in **return** instructions
- Tools like **Ropper** can auto find and filter gadgets, checking at every **byte offset**
- Given a large enough binary can build a **ROP chain** for **anything**!

Tooling

- **Ghidra/IDA/Binary Ninja**
  - Disassemblers/Decompilers for viewing and analysing binaries
- **GDB**
  - Debugging binaries
  - Good for testing your payloads and looking at memory
  - Recommend PwnDBG/GEF
- **Python pwntools**
  - Library for automating solves
  - Saves you manually copying and formatting payloads
- Our lord and saviour, **whilehex**:

```
while read -r line; do echo -e $line; done | ./[binary]
```

  - Converts hex bytes written as \xXX into the actual byte values, input non printable characters
  - Would recommending aliasing in your terminal so can easily pipe into your binary or gdb

All this + today's slides on sessions this week!

Demos!

Covering:

Buffer overflow

Ret2Win

Leaking stack values

Right Now

sessions.afnom.net

- Hang around at 5pm for **pub social**!
- Come back next week for **Cryptography!**