
ceyx@sectalks:~# 0x01.shellz.for.days

> from zero to z3r0c00l...

08..16

# $ whoami

> UNSW Alumnus

> Team Captain / Member of 9447 CTF Team

> Malware Engineer, Red Team


# $ scope

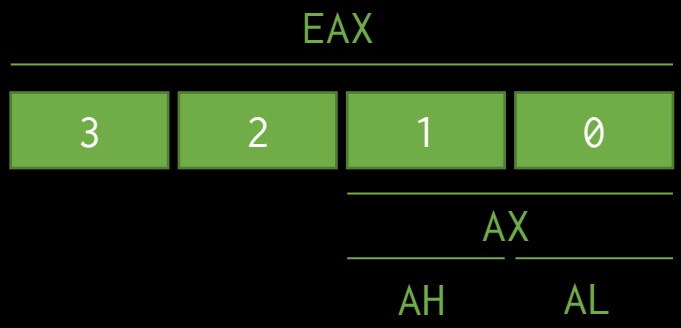
- theory
  - what is shellcode, what problem does it solve?
  - basic intel x86 assembly
  - shellcode 101
  - tools of the trade
  - limitations / restrictions
- practical
  - tools - Binary Ninja / gdb (peda) / objdump / nasm
  - guided example - reversing + exploit development

# $ 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!

# $ intel.x86.assembly._registers_

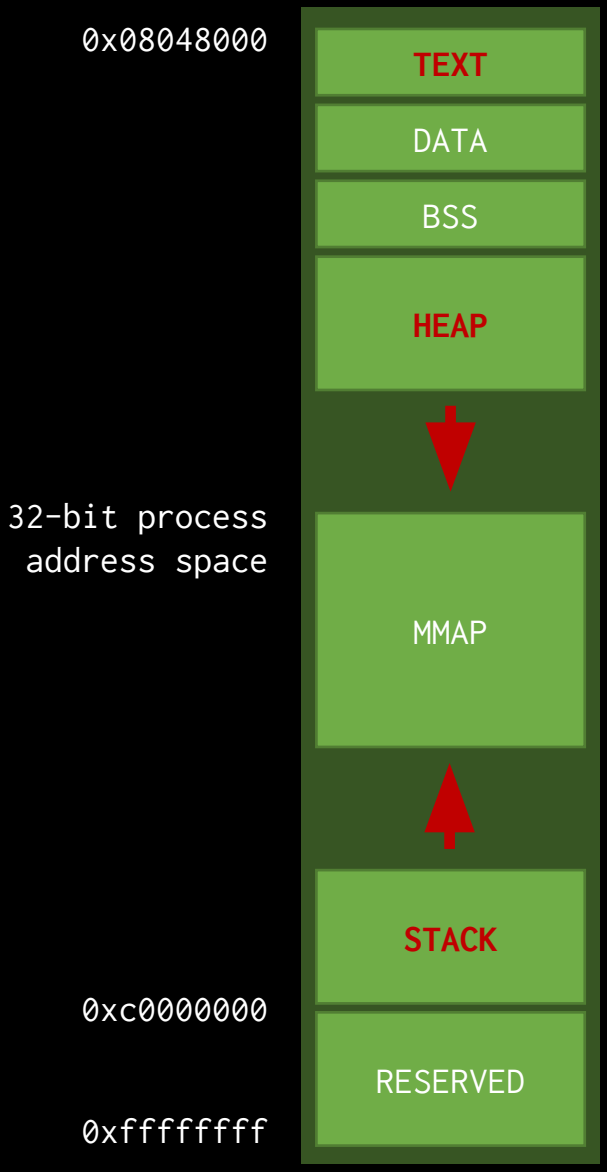
- EIP/PC - what is being executed
- ESP/EBP - reference to data
- EAX/ECX/EDX/EBX - general purpose
- ESI/EDI - source / destination
- EFLAGS - processor state




```
<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>
```

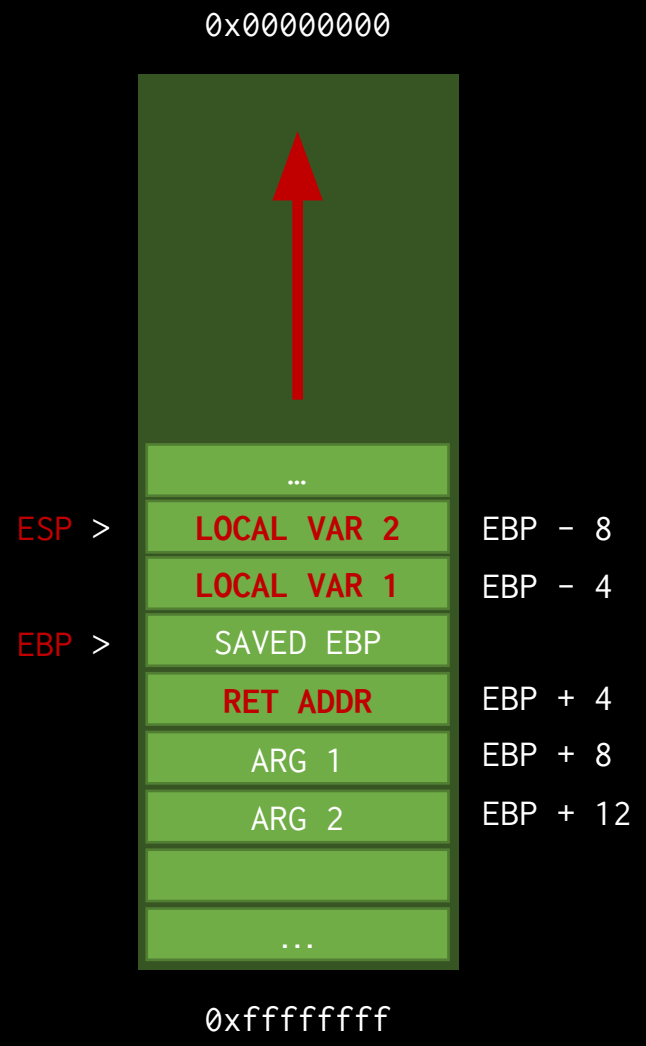
# $ virtual.memory._basics_

- 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



# $ wtf.shellcode

- it's machine code, that's all it is
- commonly used as exploit payload
- originally 'egg code' (coined by 'eggplant')
- now means to pop a shell (hence 'shellcode')
- types:
  - staged (small initial, recv more shellcode and execute)
  - egg hunter (small initial, find larger shellcode in mem)
  - omelette (small initial, recombine lots of small sections)
  - bind shell, reverse connect, open/read/send flag,
    socket reuse, download and execute, mprotect etc.

# $ intel.x86.assembly

```
>  mov eax 0x1337  - copies 2nd operand to first
>  push ebp    - pushes operand on top of stack
>  pop ebp         - pops top of stack into operand
>  xor eax ebx     - bitwise exclusive or stored in first operand
>  int 0x80    - trigger software interrupt (syscall)
>  call eax    - push eip and jmp to operand
>  ret         - pop top of stack into eip
>  jmp eax         - jmp to operand
>  cmp             - non-destructive sub, sets condition codes
>  lea esi, [ebx + 8*eax + 4] - stores calculated addr in operand
```

# $ shellcode.101

> so by what arcane magic do we fabricate shellcode?

> in theory - by channelling our inner blackhat sorcery and assembling it by hand! (priv8 shellcode collections)

> in practice - metasm / metasploit / copy pasta (shellstorm)

> important offensive security skill, consider it professional development

> real world and complex challenges will often require customisation or modification of common shellcode

> it's a building block! (userland exec)

# $ shellcode.101

> most shellcode is series of syscalls (diff calling convention)

> eax - syscall number (/usr/include/asm/unistd_32.h)

ebx / ecx / edx / esi / edi - arguments (in order)

consider, exit(0);

> cat /usr/include/asm/unistd_32.h | grep exit

#define __NR_exit 1

xor ebx, ebx- set first argument to 0 (exit value)

mov eax, 1 - set syscall number to 1 (__NR_exit)

int 0x80 - trigger software interrupt (syscall)

# $ shellcode.101._a_starting_point_

> Compile some code! gcc -static -m32 -Os shell.c -o shell

```
#include <stdio.h>
void main() {
   char *args[2];
   args[0] = "/bin/sh";
   args[1] = NULL;
   execve(args[0], args, NULL);
}
```

> Disassemble it, objdump -D shell -M intel

```
6a 00                   push   0x0
52                      push   edx
50                      push   eax
89 45 f0                mov    DWORD PTR [ebp-0x10],eax
e8 fd 4f 02 00          call   806d470 <__execve>
```

# $ shellcode.101._tips_and_tricks

> think creatively, many ways to achieve same goal

`mov eax 0` vs `xor eax eax`

`nop` vs `mov eax eax`

> as always in exploit dev, verify your assumptions

is your shellcode being executed? try this! `\xeb\xfe`

> Call and pop, puts eip on stack and then into a reg for you!

> strace is very useful for quick validation of syscalls

> consider all your attack vectors (env, args, input)

> UNDERSTAND your shellcode (debugger is your friend)

# $ tools.of.the.trade._how_2_shellcode_

Raw Binary Form: (used as payload for your exploit)

```
> nasm shellcode.asm
> objdump -b binary -M intel -D shellcode
```

Executable Form: (testing your shellcode does what you want)

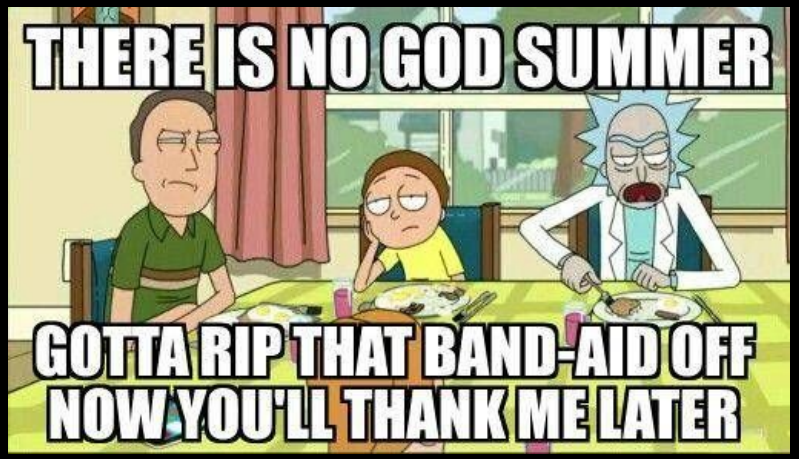
```
> nasm -f elf32 shellcode.asm
> ld -m elf_i386 shellcode.o -o shellcode
> objdump -D shellcode
```

```
global _start    ; nasm, elf file boiler plate (BITS 32)
section .text
_start:
```

# $ limitations._when_dolphins_cry_

> small buffers (limited shellcode size)
> alphanumeric (enforced by input vector)
> null-free (generalised to badchars)
> sanitisation or banned instructions (signature match AV?)
> unknown location / state

> polymorphic
> encoders
> getpc routines (metasploit)

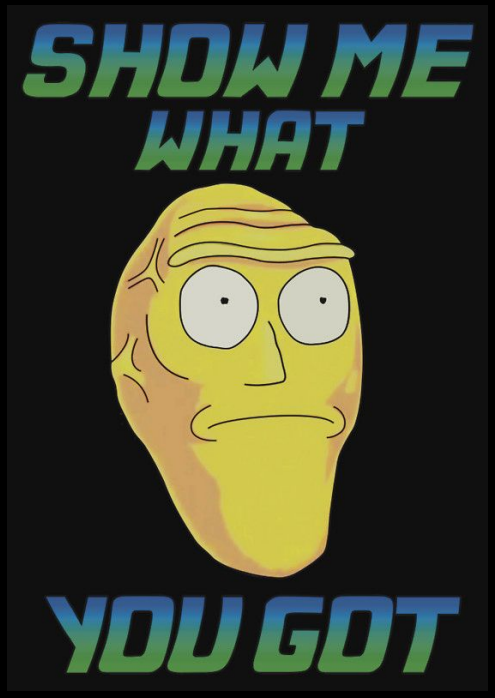


# $ shellme._like_one_of_your_french_girls_

> initial binary analysis
> run it, manual input testing
> (crash?) open up in disassembler
> reverse interesting bits
> discover/understand vulnerability
> write POC exploit to control EIP

```
python -c "print '\xeb\xfe'" | ./shellme
```

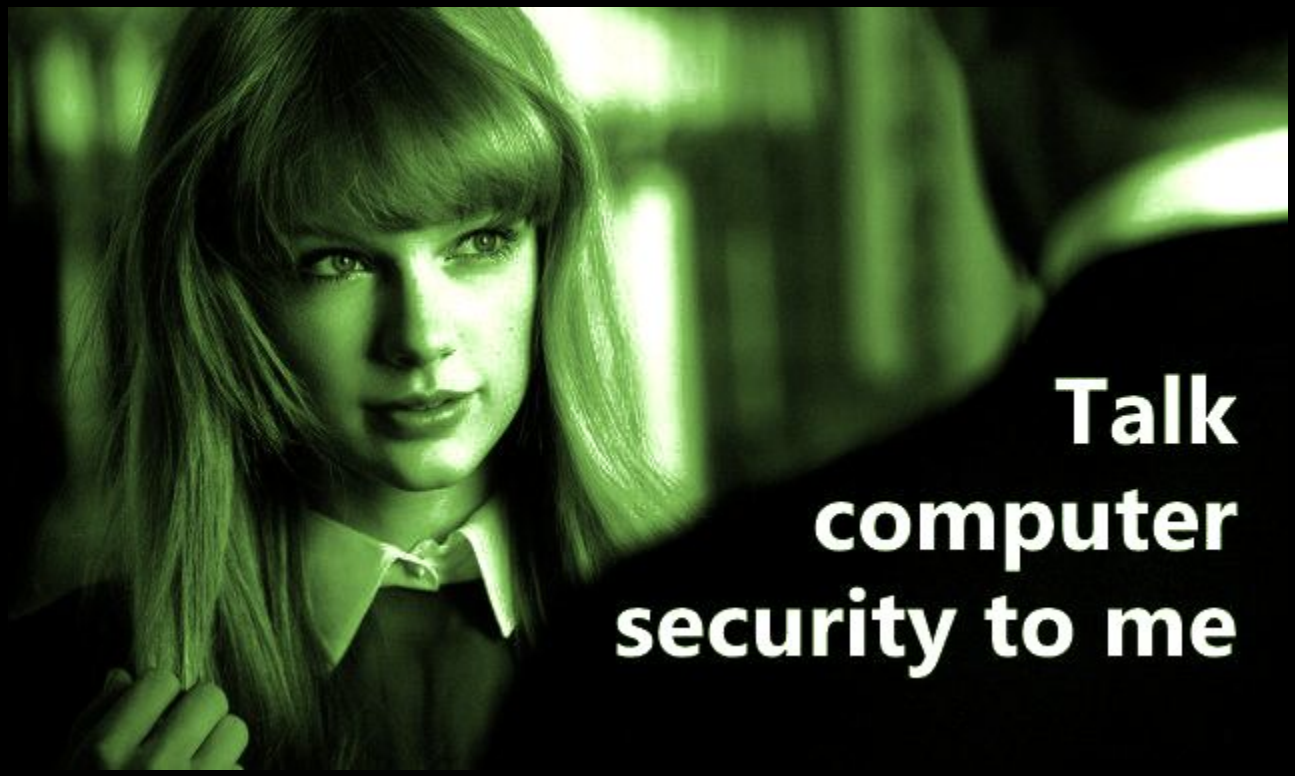
> construct payload (shellcode for us)
> iterate: test -> debug -> adjust
> pwn local -> pwn remote
> … profit



# $ references


> “Smashing the stack for fun and profit”, by Aleph One
> http://phrack.org/issues/49/14.html
> “Safely Searching Process Virtual Address Space”, by skape
> http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf
> “Writing ia32 alphanumeric shellcodes”, by rix
> http://phrack.org/issues/57/15.html
> Windows shellcoding guide, old but quality.
> https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/
> Shellstorm Database, http://shell-storm.org/shellcode/

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



$ feedback.sectalks._thank_you_

GREETINGS PROFESSOR FALKEN.
SHALL WE PLAY A GAME?

1. GOTO http://hack.Sydney
2. download src/binz - 0x01.ctf.zip, pass: #GetSchwifty!
3. exploit challenges:
   - shellme - hack.sydney:9008
   - golddigger - hack.sydney:9009
4. email flags to jcramb@gmail.com
5. share your skillz with others!