COMP9447 15s2 C2

INTRODUCTION TO ASM AND REVERSE ENGINEERING

Assignment

- Assignment selections are due by Monday 11am next week (11am 17th August)
- One person from each group has to email comp9447@gmail.com with:
  - The names and student numbers of everyone in the group (if you are working solo, a really good reason why is required and I have to agree – you should have emailed me by now)
  - A list of at least three targets, in order of preference

Assignment

- Sample list

A: GCC

B: OpenSSH

C: Multiple SSH Servers

D: Firefox browser

Software must be written in C/C++ or similar or must be a compiled binary only. Can be linux, windows, mac, Atari, I don't care.

Assignment

- Your group needs to keep a blog on open-learning with your details of what you’ve done.
- If more than one group is doing the same project, we’ll play with permissions so only Fionnbharr, Richard + I can see your work.
- Marks will be allocated to how well you maintain your blog.

Assignment

- By midnight on Monday of week 8, you need to send a 3-5 page powerpoint presentation which is the “halfway” point of your work. Each team will need to talk for 5 minutes either in week 8 or 9 (chosen at random.. ;)).
- Include a quick summary of what you’re testing, what you are doing, how much success you have had and what the future half of the semester looks like.

Assignment

At the end of the semester (before the exam period – dates are online in 1A.pdf) your group needs to submit a written document in paper style detailing:

- The approaches you took
- The success/failures you had
- What you would do differently in future / had more time
- Details of any vulnerabilities you found and exploits written
- Source code is required for all bugs, fuzzers, tools and exploits – unless you get prior preapproval from me
- Assignments will be marked on technical method, dedication, success, and engineering quality (if you write exploits)

REVERSE ENGINEERING

- Mostly for man or machine made things.
- You take something you don't know, and work out how it works.
- In this context we’re talking about computer reverse engineering. Taking an unknown binary, without source code, and either working out how it works, or turning it back into source code.
- Engineering is the opposite of science, so reverse engineering is essentially science (but studying man made things, rather than naturally occurring)

Why?

- Copyright protection/DRM
- Security Testing – To verify something fits the spec.
- Security Testing – To find vulnerabilities :D
- Malware. Work out how it works to stop it, or even just determine the threat.
- Crypto – A lot of custom crypto, once you see the source, is ridiculously easy to break.
- No documentation – frequent with drivers.
- Many many more reasons.
- Very lucrative market for pro reverse engineers.

Source code discovery

- Most machine code/assembly is generated from a higher level language (ie. C, C++) that is compiled into a binary. In a way, a lot of what we're doing is decompiling, from assembly to another language.

- Most of our examples will be C and C++. As C is turing complete, you can essentially represent all assembly code with it. But if you reverse some other language (like pascal), your C code will be very messy most of the time.

- Some information, that is not included in the binary, will be lost. Comments, defines, etc. are all gone, because they don't actually exist in the binary.

Architecture

- Reversing differs based on architecture, as we're reversing machine specific code. Be aware that a lot of what you learn applies only to x86, but a lot of it applies to other platforms, and a lot more can be reapplied or reversed to apply.

- x86 is generally a lot harder to reverse than most other architectures. (sparc is the easiest in my opinion, but risc is almost always easier than cisc) as there are many more instructions available.

- However, the vast majority of code in use falls into a small subset of instructions that'll we'll learn here today.

A quick note on display formats

AT+T which is slowly dying for intel:

```
objdump -d /bin/something

push %ebp
mov %esp,%ebp
sub $0x18,%esp
mov 0x8(%ebp),%eax
call 0x80482d4 <exit@plt>
```

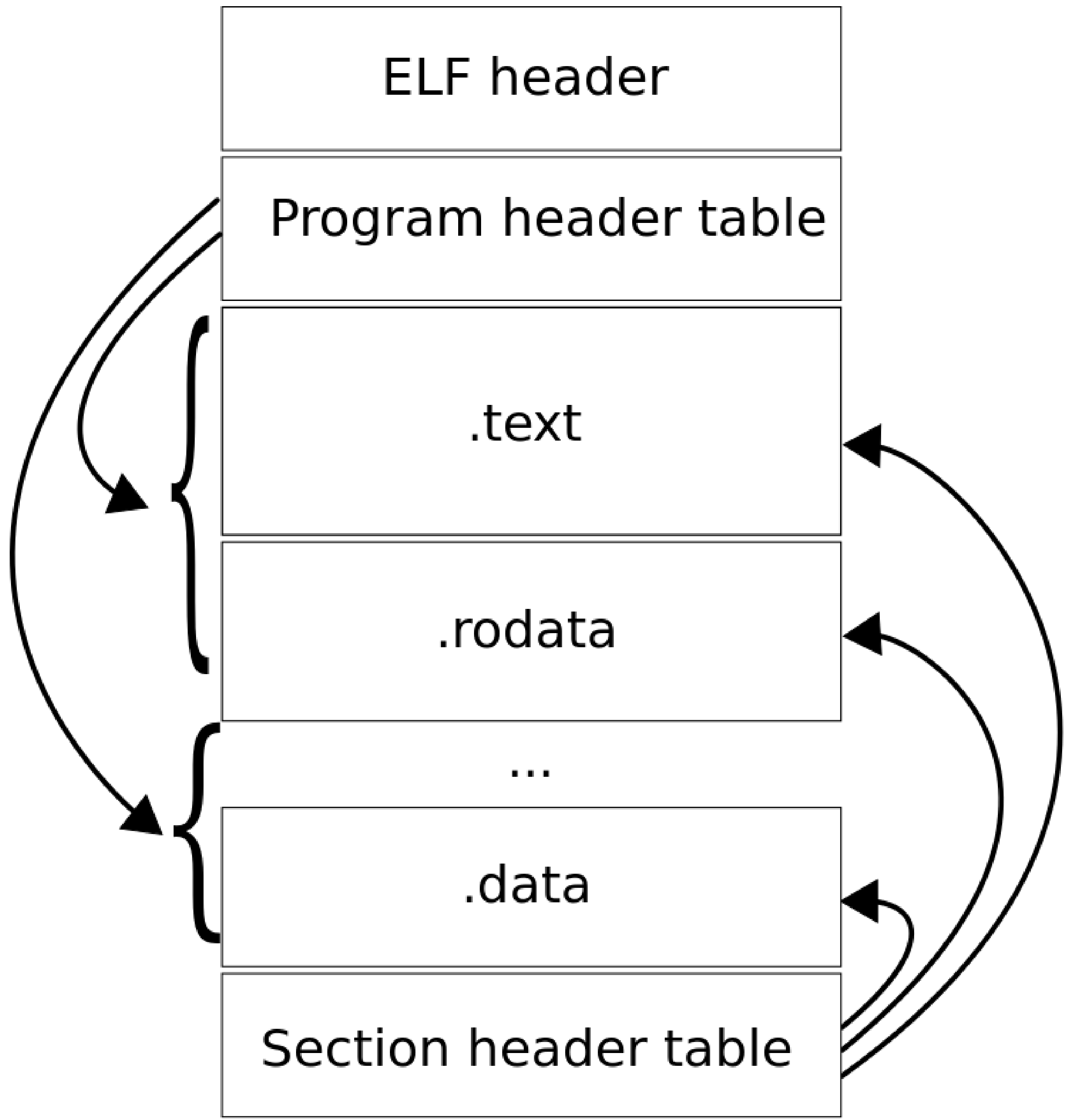
INTEL, what we'll use in all of our examples/tests and what you should use:

```
objdump -M intel -d /bin/something

push ebp
mov ebp, esp
sub esp,0x18
mov eax,DWORD PTR [ebp+0x8]
call 0x80482d4 <exit@plt>
```

For gcc, set disassembly-flavor intel

ELF

- Common executable file format on Linux and other commonly used *nix systems
- Each ELF file is made up of one ELF header, followed by file data. The file data can include:
  - Program header table, describing zero or more segments
  - Section header table, describing zero or more sections
  - Data referred to by entries in the program header table or section header table
- An ELF file is a set of segments and sections.
  - kernel (runtime) sees segments, maps them into virtual address space using mmap syscall
  - linker sees sections, combines them into executable/shared object


ELF header
Program header table
.text
.rodata
...
.data
Section header table

Important ELF sections

- **.text** - executable instructions
- **.bss/.tbss** - Block Started by Symbol, uninitialised data, zeroes
  - e.g. a global variable that is uninitialised (or set to null) would live here
  - This section normally takes no actual space in the on disk ELF
- **.data/.tdata** - initialized data / thread data
  - e.g. global/static variables that are initlaised with a non-null value.
- **.rodata** - read-only data
- **.dynamic** - dynamic linking
- **.got{,.plt}** Global Offset Table
- **.plt** Procedure Linkage Table
- **.strtab** String Table
- **.init/.fini** executable instructions, initialization code
- **.{init,fini}** array array of function pointers to init functions

x86 is a register machine

- Registers are small regions of memory (32 bits on a 32 bit architecture, generally) which are located directly on the CPU.
- Accessing registers is much faster than accessing general memory (RAM)
- Some registers have special reserved purposes, some do not and are “general purpose”, but are frequently used for specific purposes (we'll go into some of these later)

Registers

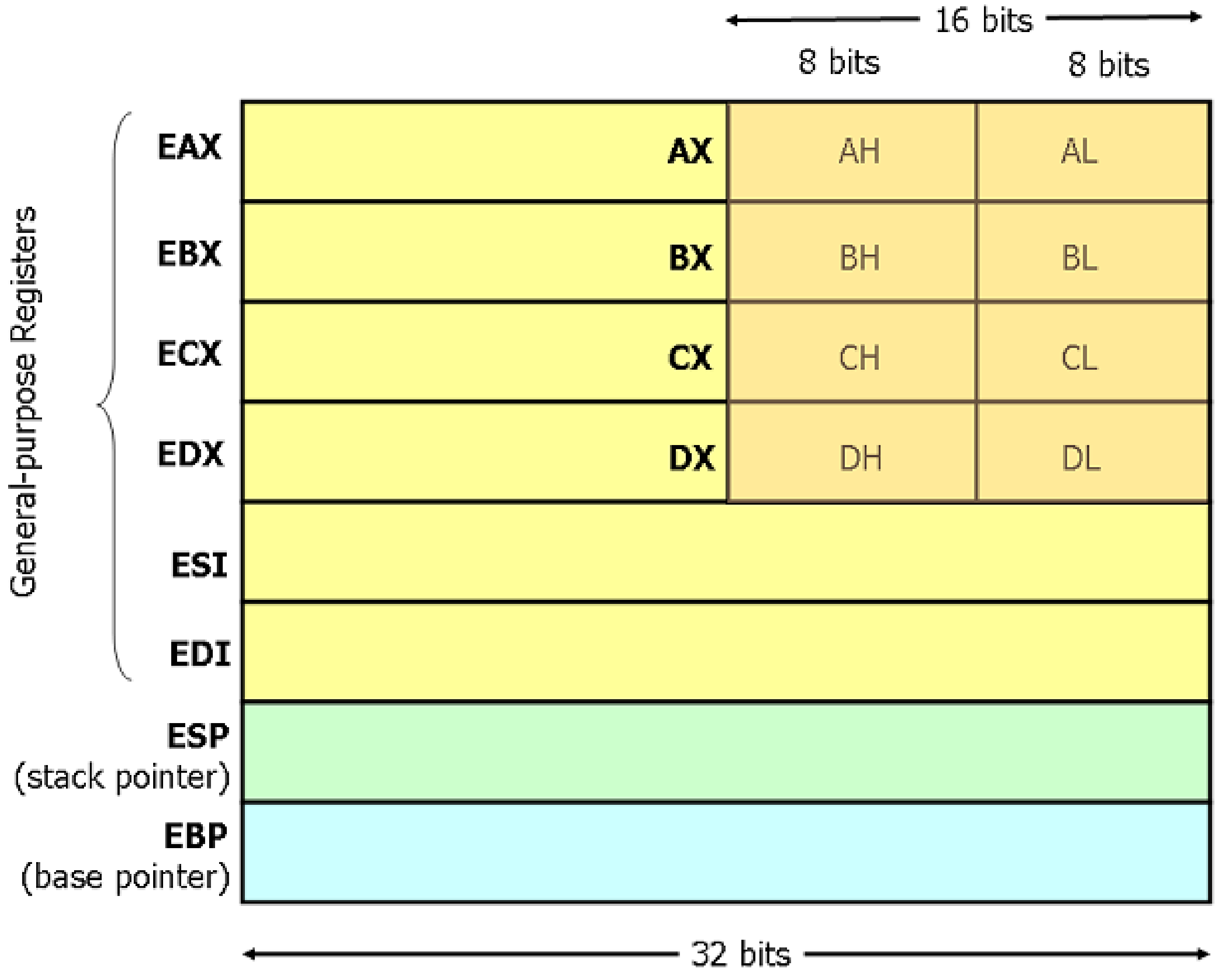


Image stolen from
http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Useless slide title

- The first four registers (eax, ebx, ecx, edx) can be accessed in 16 and 8 bit sections as well.
  - AX is the least significant (low) 16 bits of EAX.
  - AH is the most significant (high) 8 bits of AX.
  - AL is the least significant (low) 8 bits of AX.
- There is no direct way to access the top 16 bits of these registers only.
- X86 used to be sixteen bits! (the E stands for EXTENDED) And now, x86-64 (64 bit) is becoming a lot more prevalent. We'll only really cover 32 bit x86, as it's still the basis of most applications, and most 64 bit applications also come in 32 bit versions.
- X64 registers are out of scope here, but are similarly designed
  - RAX (64), EAX (32), AH(16), AH + HL (8)

Register usage

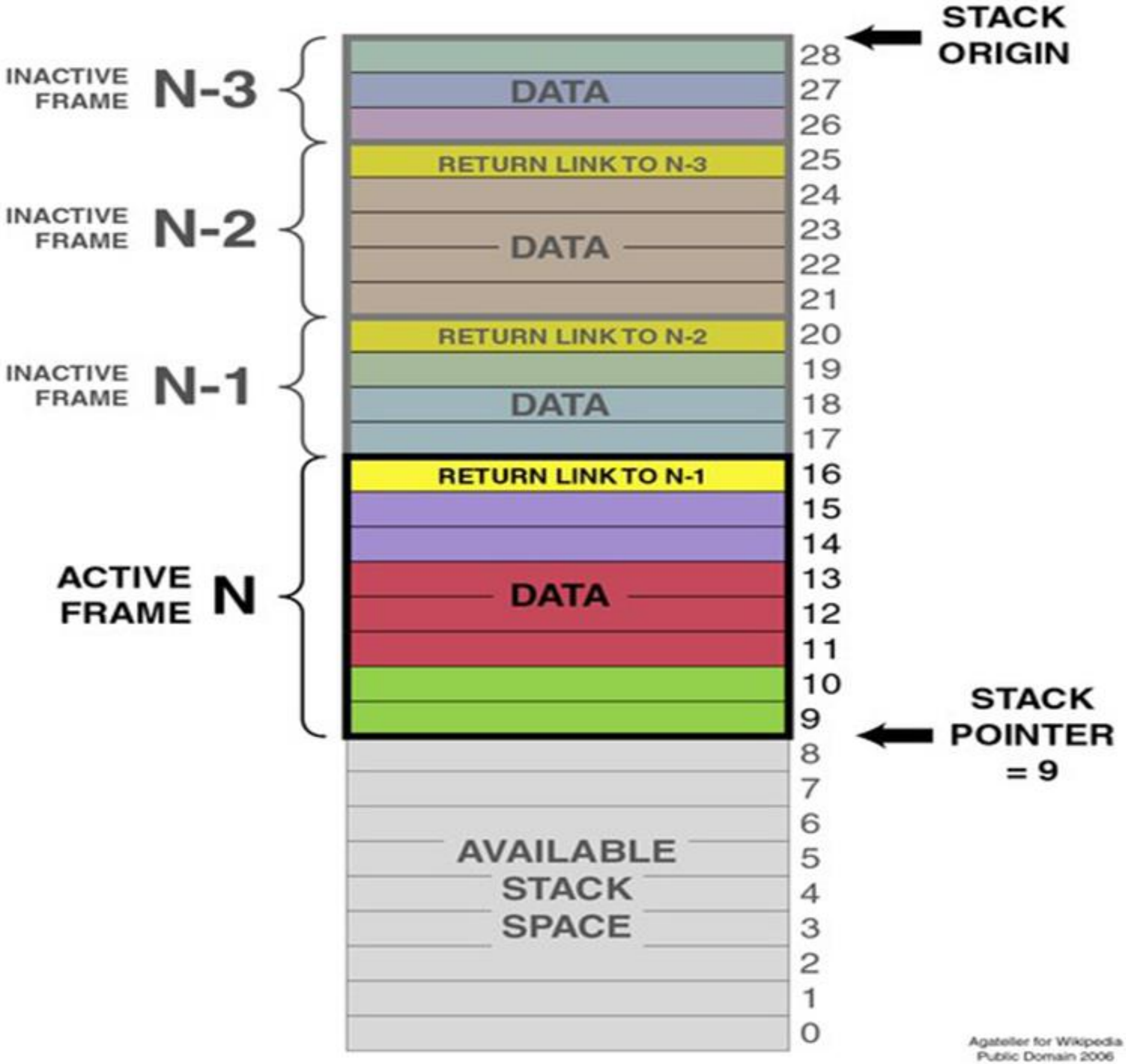
- **EAX** — Accumulator for operands and results data
  - **EAX** is frequently used as the first argument to a function call, and is frequently used as the return code from a function call.
  - **EAX** is also the return value for a function.
- **EBX** — Pointer to data in the DS segment
  - **EBX** is frequently used as a “base index”, for arrays.
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the **DS** register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the **ES** register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

ESP

- Extra Sensory Perception
  - Crush other students minds
- Extended Stack Pointer
  - ESP POINTS TO THE TOP OF THE STACK! That is, ESP contains the memory address of the top of the stack.
- The “top item” of the stack is actually at a lower address in memory than the previous item. That is, the stack grows from high addresses to low addresses. This is because 0 is the “top of the page”.

EBP

- EBP is the "Extended Base Pointer". It is also commonly referred to as the "Frame pointer".
- Stack frames are used so that each function has it‟s own area of the stack, and so that the stack can be restored to the state it was in when before a function was called.
- EBP is not necessary, but is very helpful, and is used by many compilers. It points to the bottom of the current stack frame, which is also where ESP pointed before the function was called.



STACK ORIGIN
INACTIVE FRAME N-3
DATA
RETURN LINK TO N-3
INACTIVE FRAME N-2
DATA
RETURN LINK TO N-2
INACTIVE FRAME N-1
DATA
RETURN LINK TO N-1
ACTIVE FRAME N
DATA
STACK POINTER = 9
AVAILABLE STACK SPACE
28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Agateller for Wikipedia
Public Domain 2006

MOV

- **MOV eax, ebx** : moves the content of ebx to eax.h
- **MOV eax, [ebx]** : moves 4 bytes of content of memory from the address in ebx to the eax Register.
  - [ ] are like * (or ->) in C
  - eax = *ebx;
- **MOV eax, [ebx+4]** : moves 4 bytes of content of memory from the address in ebx plus 4, to the contents of EAX.
  - eax = ebx->offset_4;
  - eax = *(ebx+4);
- **MOV ax, dx** : moves the content of dx into ax. (lower halves of edx and eax)

ADD + SUB

- **ADD ebx, ecx** : Adds ebx and ecx and stores the results in ebx
- **SUB ebx, ecx** : Subtracts ecx from ebx and stores the result in ebxs

LEA

- LEA - Load Effective Address
- Can be confusing at first
- Loads the address of the source operand into the destination!
- Think of it as ‘&’ in C.
- Often used to do instructions like **LEA EBX, [ECX+4]** which is the same as the following two instructions:
  - MOV EBX, ECX
  - ADD EBX, 4
- Can be used for multiplication in a single instruction **LEA edx, [edi+edx*4]**
- tl;dr, think of it like a fancy **MOV** and ignore the [ ]'s

JMP/CALL/RET

- **JMP** is
  - **MOV EIP, <addr>**
- **CALL** is
  - **PUSH EIP**
  - **JMP <ADDR>**
- The difference being call is for saving state to return back to in the future.
- **RET** is
  - **POP EIP**
- This is used for returning from function calls.
- It can take an optional argument where it cleans up some arguments on the stack before popping. This only happens when the callee is responsible for cleaning up the stack
- **RETN 8** (probably cleaning up two arguments)
  - **POP EIP**
  - **ESP -= 8**

TEST and CMP

- Basis of most ‘if’ statements
- **TEST** is a bitwise **AND**, but doesn’t save the
- value just sets appropriate flags in EFLAGS
- **CMP** is a **SUB**

Conditional jmps

- There are a lot of these, so you may have to look them up..
  - JZ = jump if zero flag set
  - JNZ = jump if zero flag not set
  - JGE = jump if greater or equal
  - JLE = jump if less than or equal
- Etc. many many combinations

CMP EAX, EBX
JNZ address

- Means: if EAX-EBX is not equal to 0, jump.
- One thing to remember: JNE and JE are pseudonyms for JNZ and JZ

Loops and comparisons are backwards

- Because of how assembly works, conditions are normally checked backwards, because we're jumping code.
  - if (a == b) { **something()** }; *second_thing();*

When translated to assembly, is saying, if a != b, jump over the brace.

; eax contains a, b is in ebx

CMP eax, ebx

JNZ past_the_if

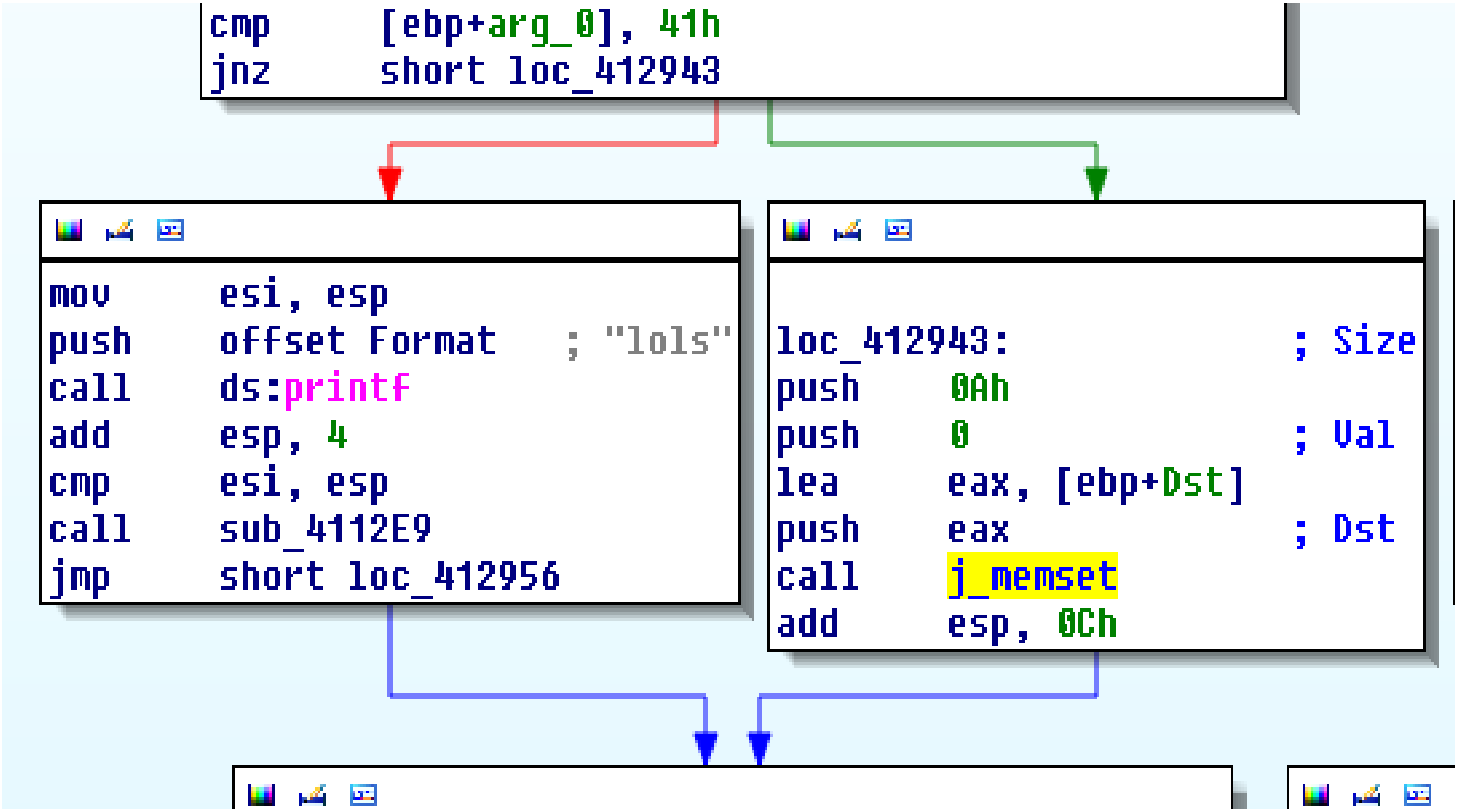
CALL something

past_the_if:

CALL second_thing;

Another example

```
if(argc == 0x41) // 'A'
{
    printf("lols");
}
else
{
    memset(szCmdline, 0, 10);
}
```



Function prologues and epilogues

- As we just learnt, **EBP** points to the base of a stack frame, and **ESP** to the top (newest) element.
- When a function is entered, it must save the stack frame(if stack frames are being used) so the previous function can restore it, and then make the top of the stack the bottom of the new frame.
  - **PUSH EBP**
  - **MOV EBP, ESP**
  - **SUB ESP, 8** // It will then reserve space for local arguments
- 8 bytes of arguments - meaning generally two local vars (32 bits * 2 == 8 bytes).

Calling Conventions

- **CDECL**, most common on linux
- Arguments are pushed right to left
- function(a, b, c); in c looks like:

push c

push b

push a

call function

- The caller cleans up the stack. After that function call we might see:
  - **add esp, 0Ch**
- which cleans three arguments (12 bytes)

It is good to make small examples in C and look at output

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

noinline int
add(int one, int two)
{
return one + two;
}

int
main(int argc, char *argv[])
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    printf("%d\n", add(a, b));
}
```

```
$ gcc -m32 samp2.c -o samp
$ objdump -M intel -d samp | less
..
 804842e:       89 44 24 18              mov
DWORD PTR [esp+0x18],eax

mov    DWORD PTR [esp+0x18],eax
mov    eax,DWORD PTR [esp+0x18]
mov    DWORD PTR [esp+0x4],eax
mov    eax,DWORD PTR [esp+0x1c]
mov    DWORD PTR [esp],eax
call   80483f4 <add>
..

$ gcc -m32 samp2.c -Os -fno-inline-small-
functions -o samp

 804836b:       push   eax
 804836c:       push   esi
 804836d:       call   8048444 <add>
```

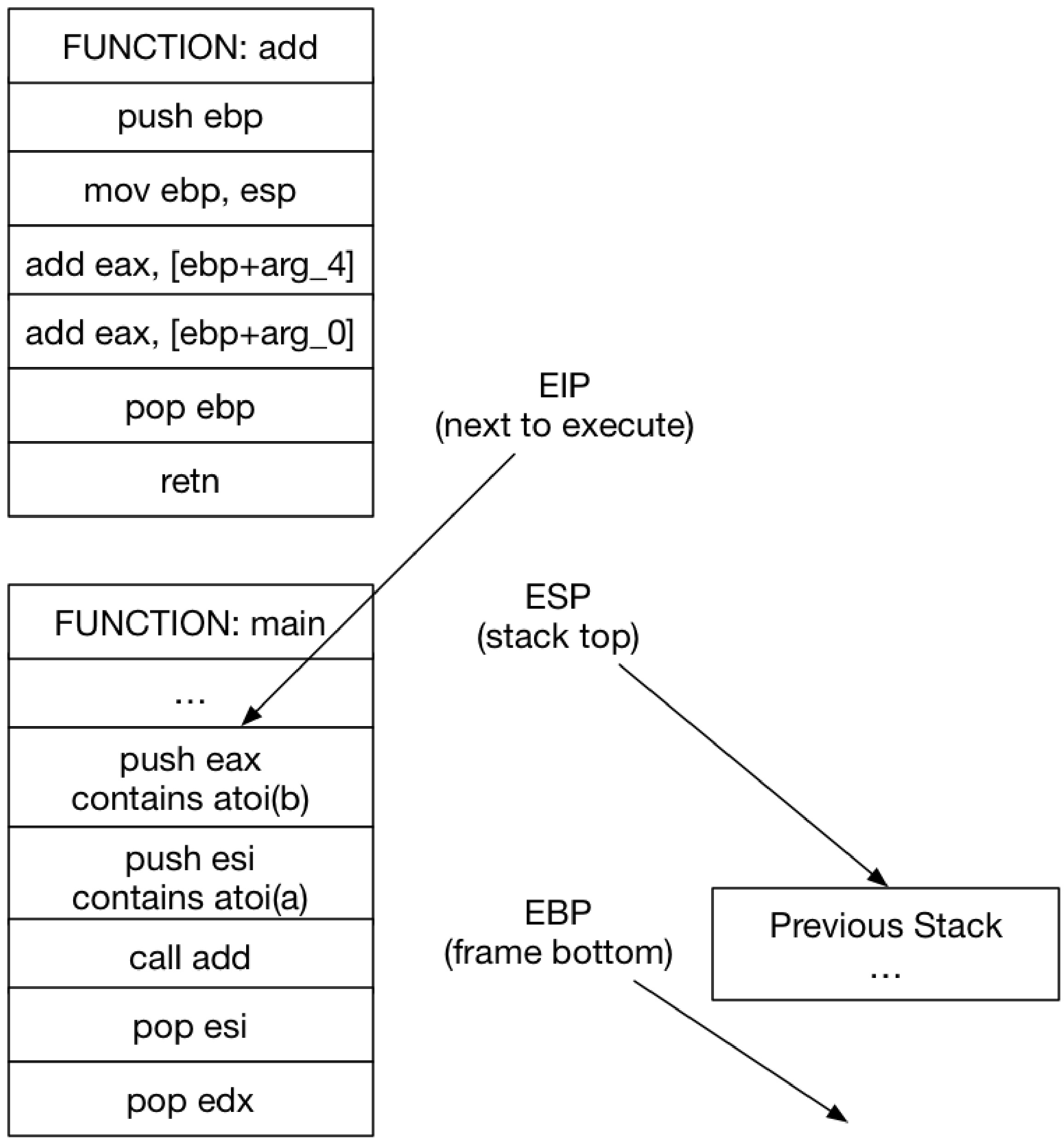
It is good to make small examples in C and look at output

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

noinline int
add(int one, int two)
{
return one + two;
}

int
main(int argc, char *argv[])
{
	int a = atoi(argv[1]);
	int b = atoi(argv[2]);
	printf("%d\n", add(a, b));
}
```

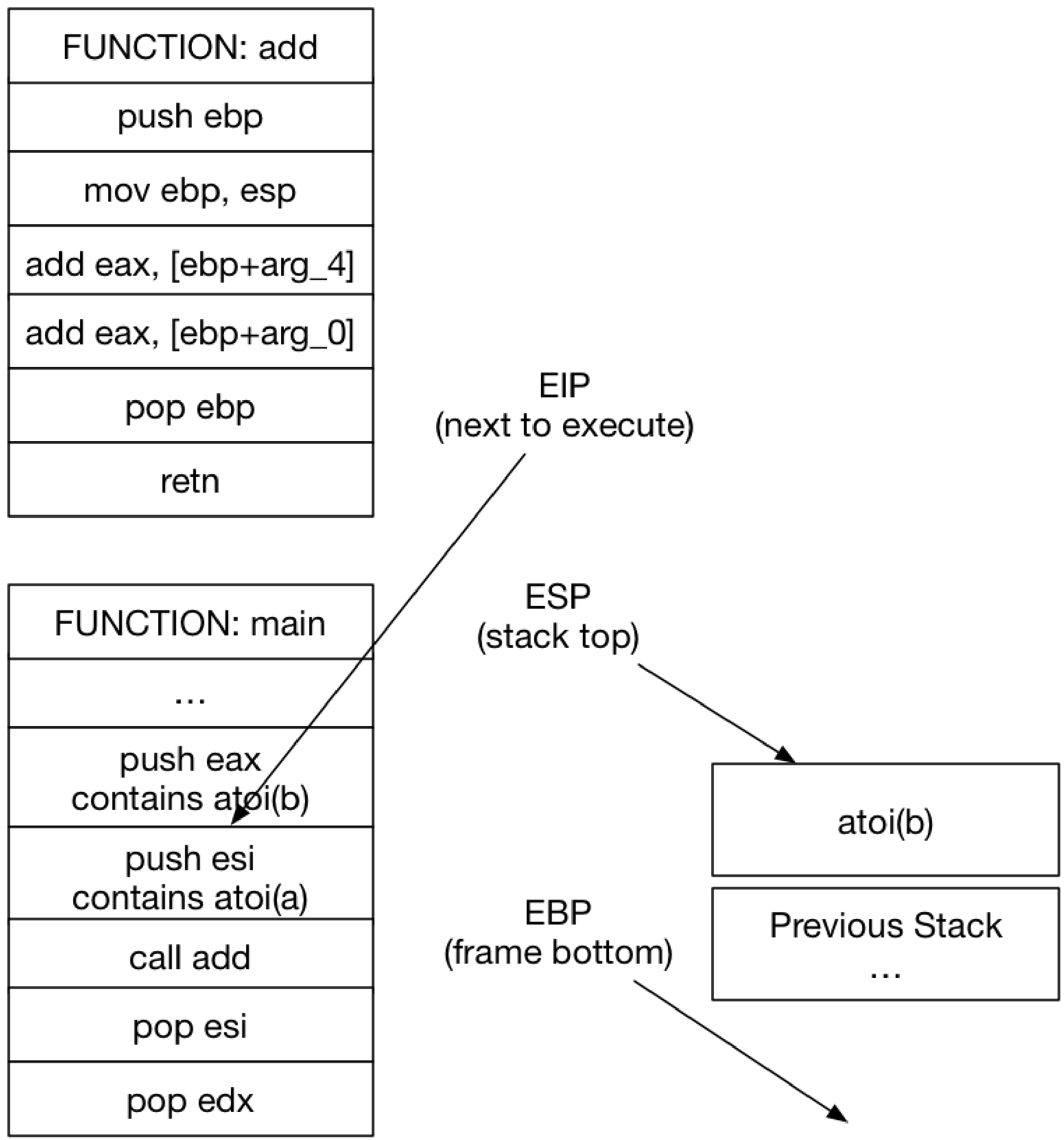
```
; Attributes: bp-based frame

public add
add proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_4]
add     eax, [ebp+arg_0]
pop     ebp
retn
add endp
```

```
call    _atoi
pop     edx
push    dword ptr [ebx+8] ; nptr
mov     esi, eax        ; Save the first atoi
                        ; - atoi(a) in esi
call    _atoi
pop     ecx
pop     ebx
push    eax             ; return value was placed
                        ; into eax by called function (atoi)
push    esi
call    add
pop     esi
```

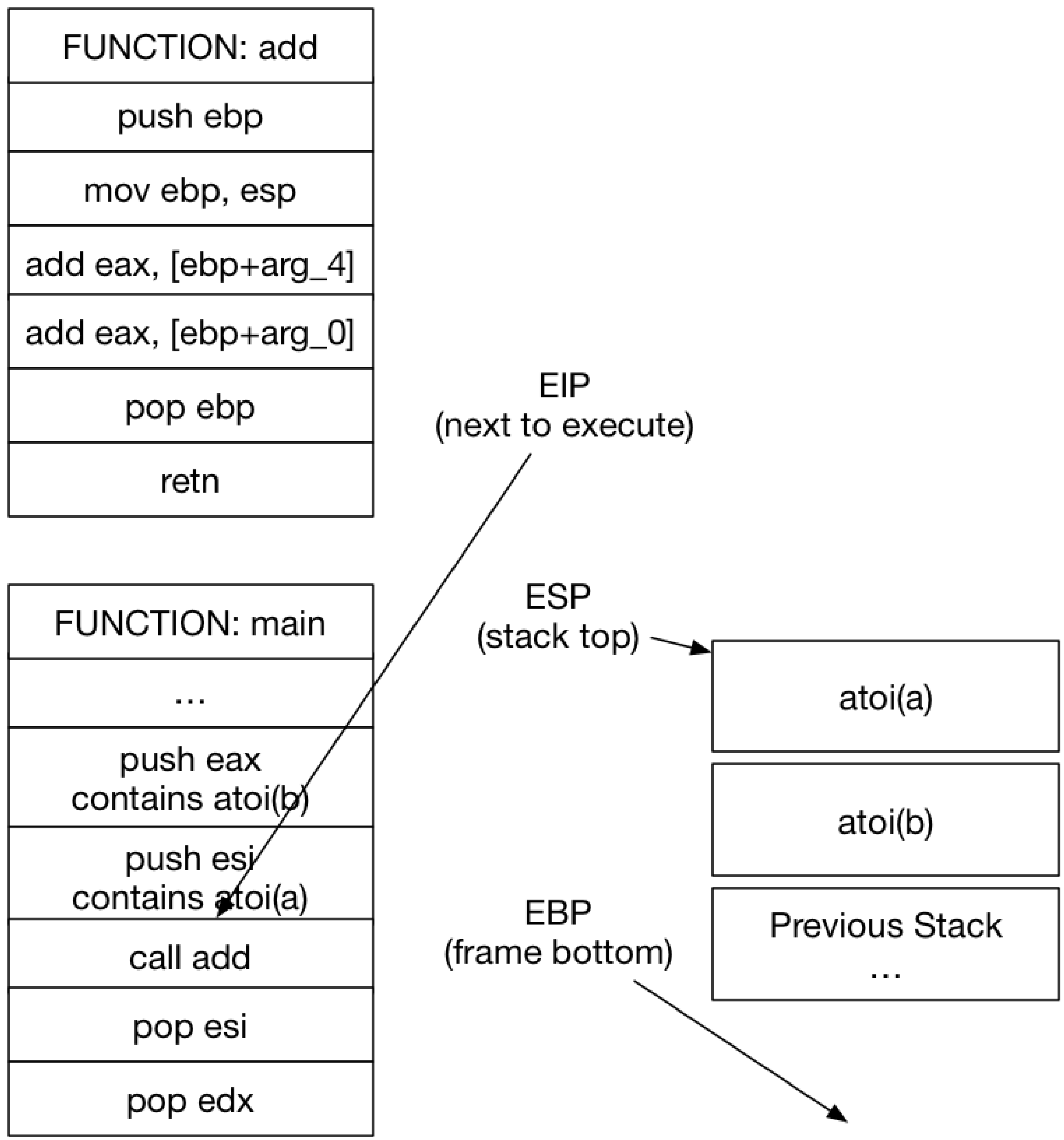
Step by step CDECL 1



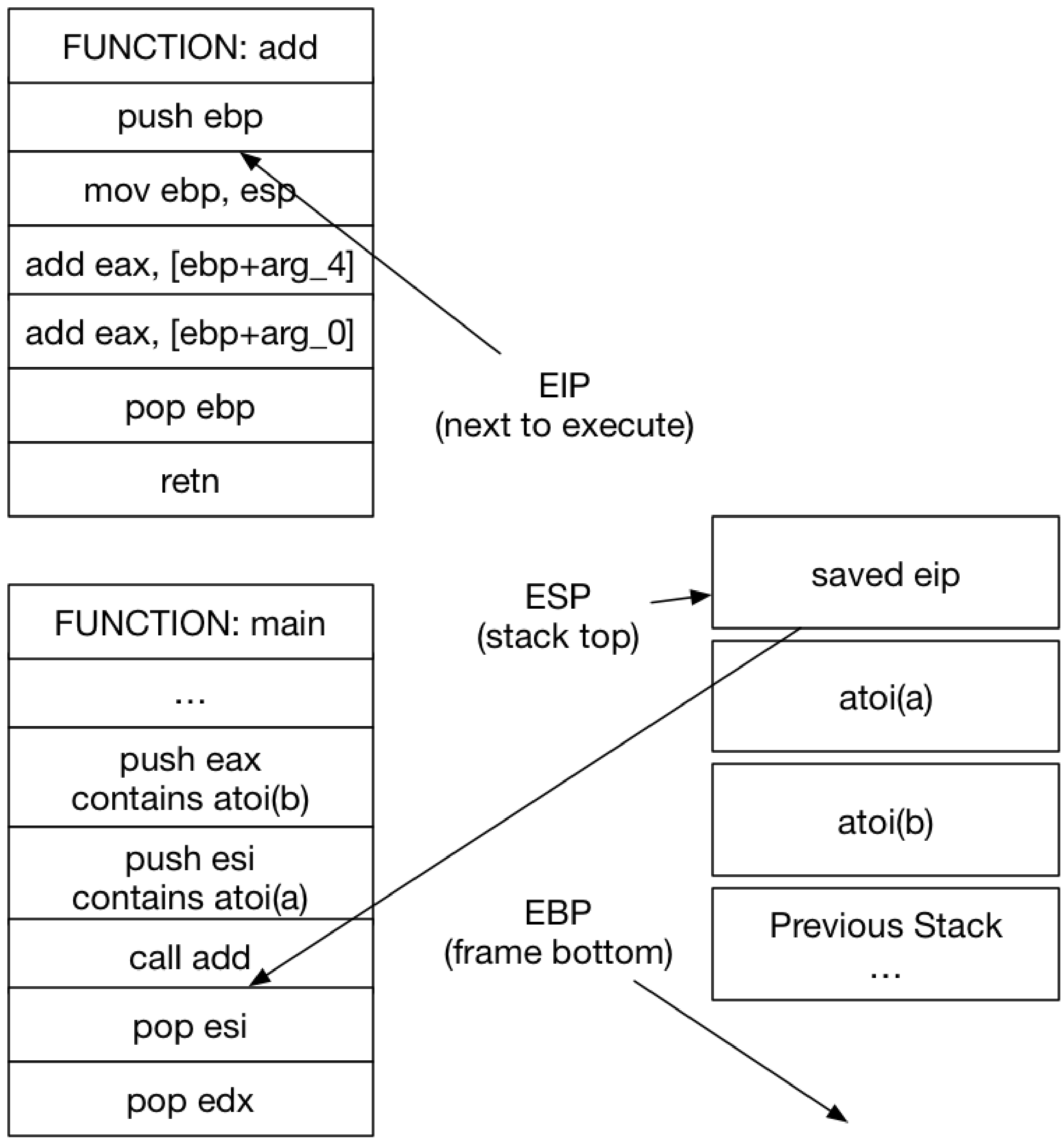
Step by step CDECL 2



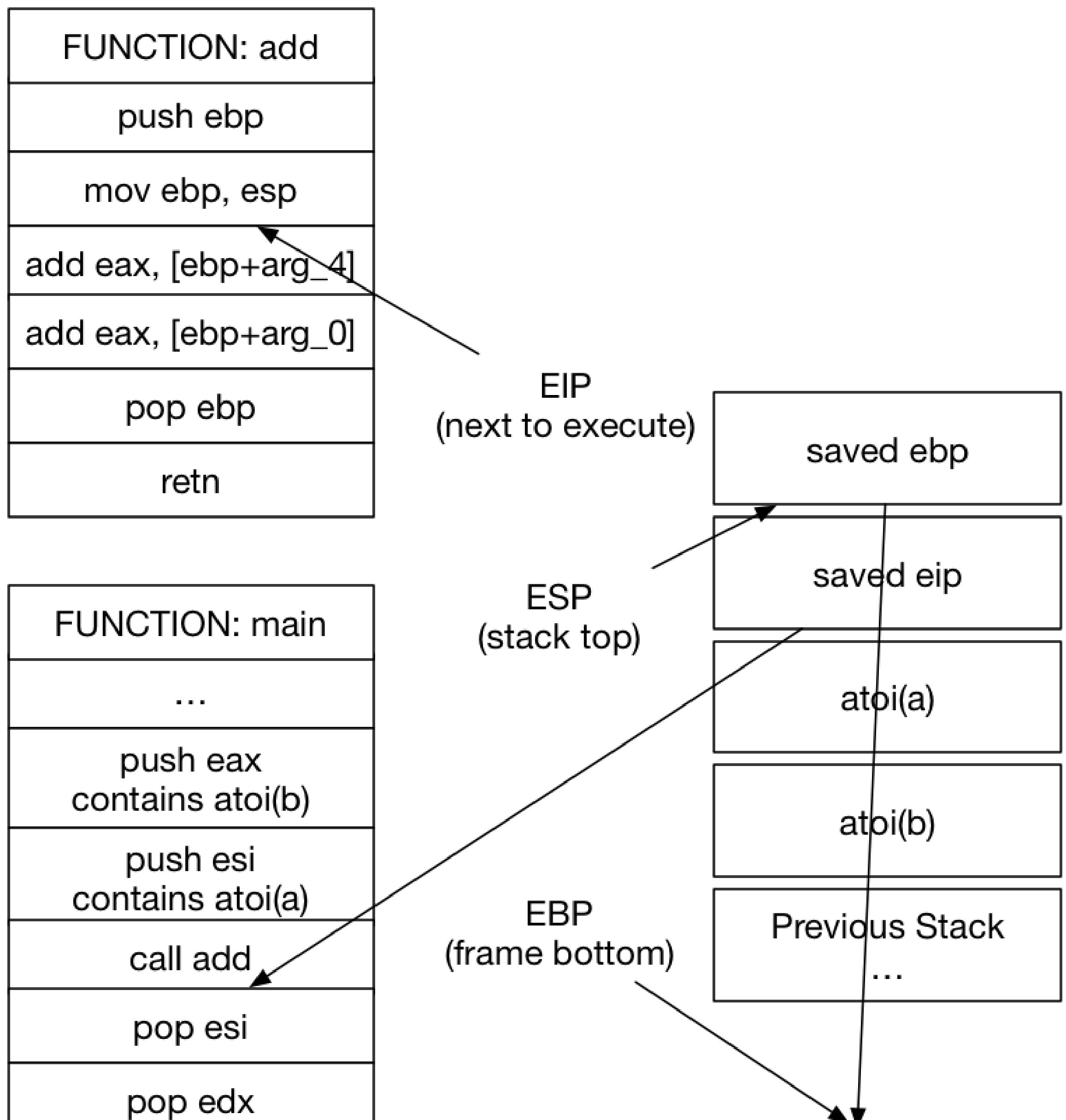
Step by step CDECL 3



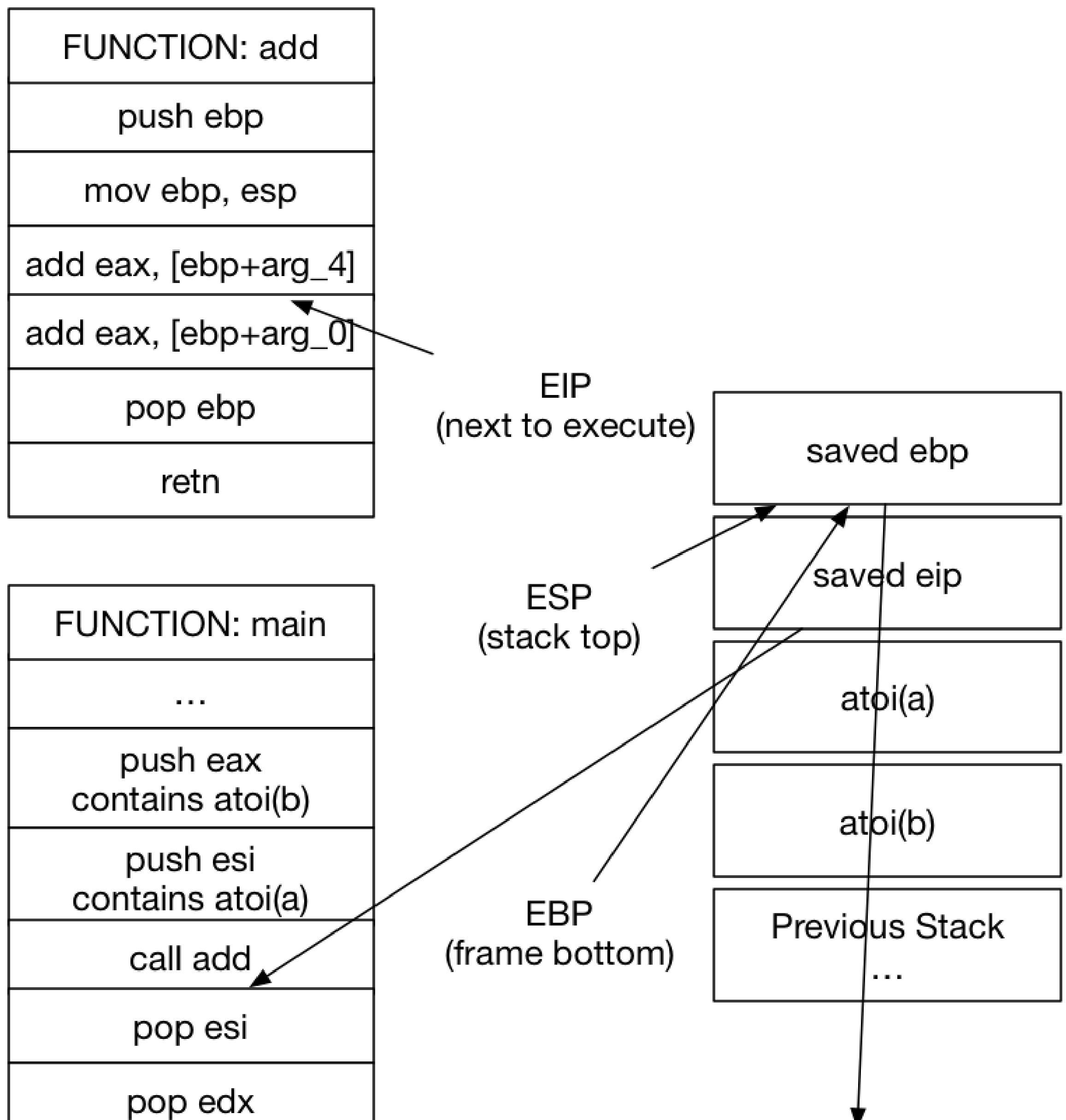
Step by step CDECL 4



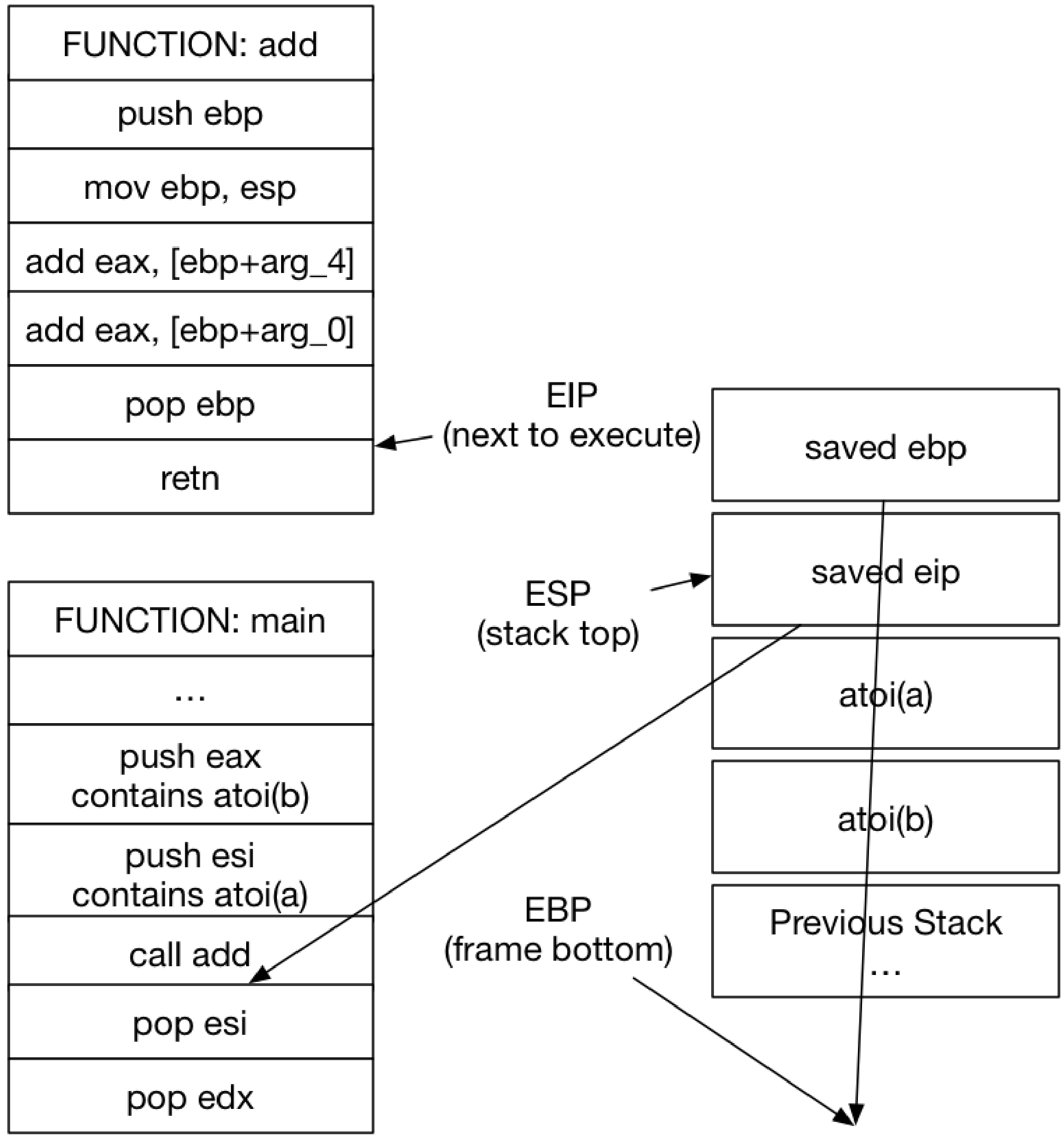
Step by step CDECL 5



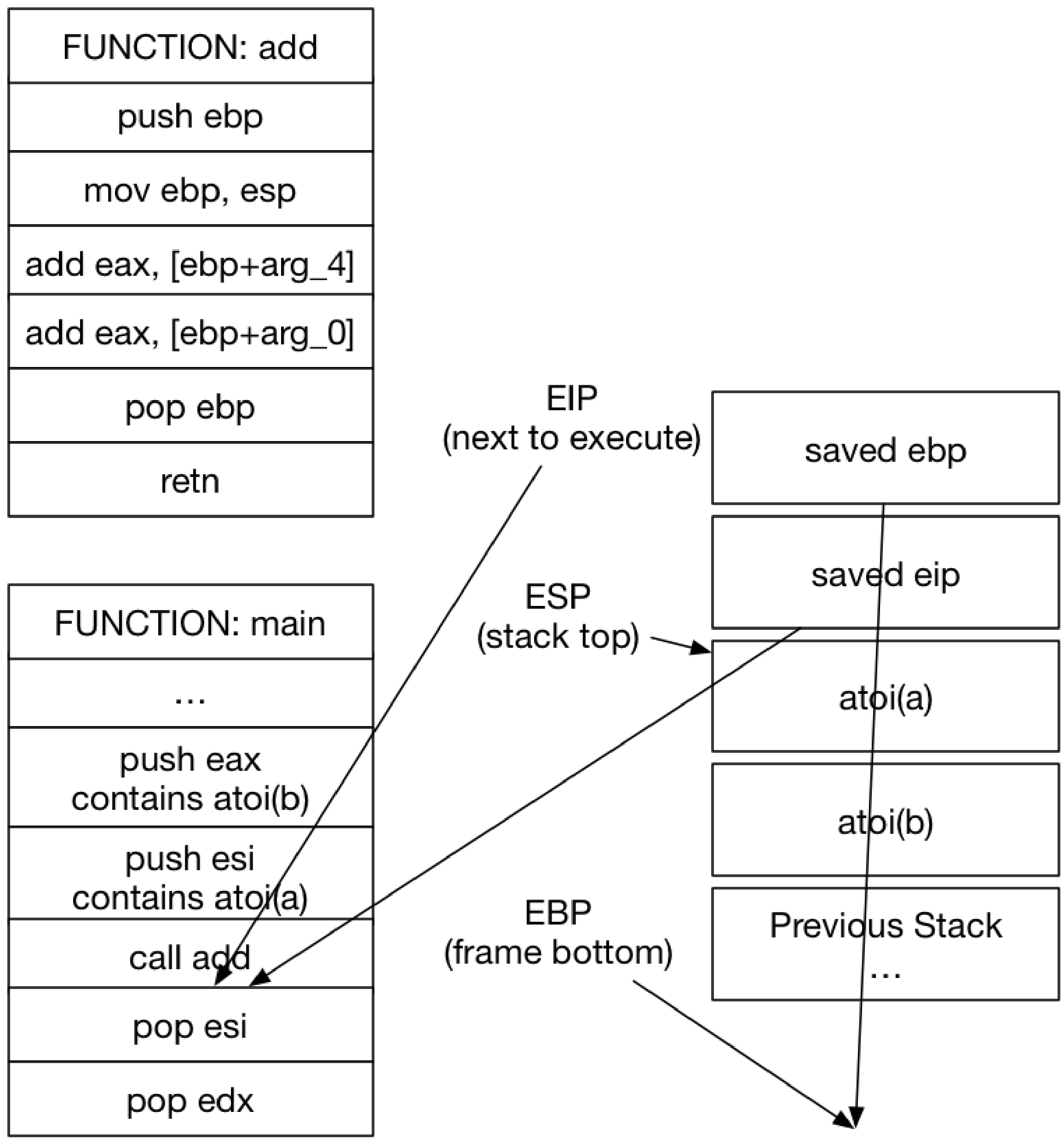
Step by step CDECL 6



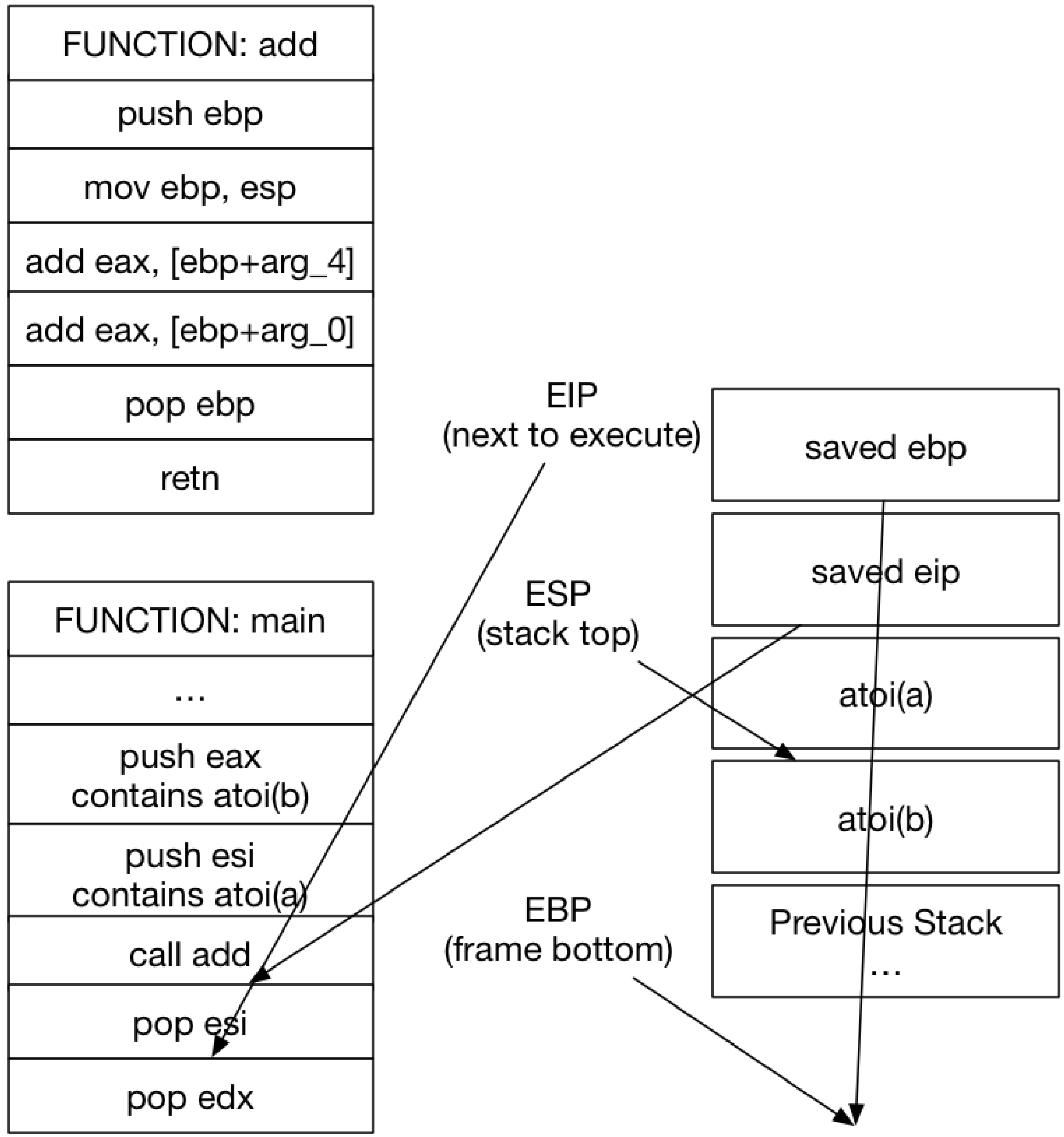
Step by step CDECL 7



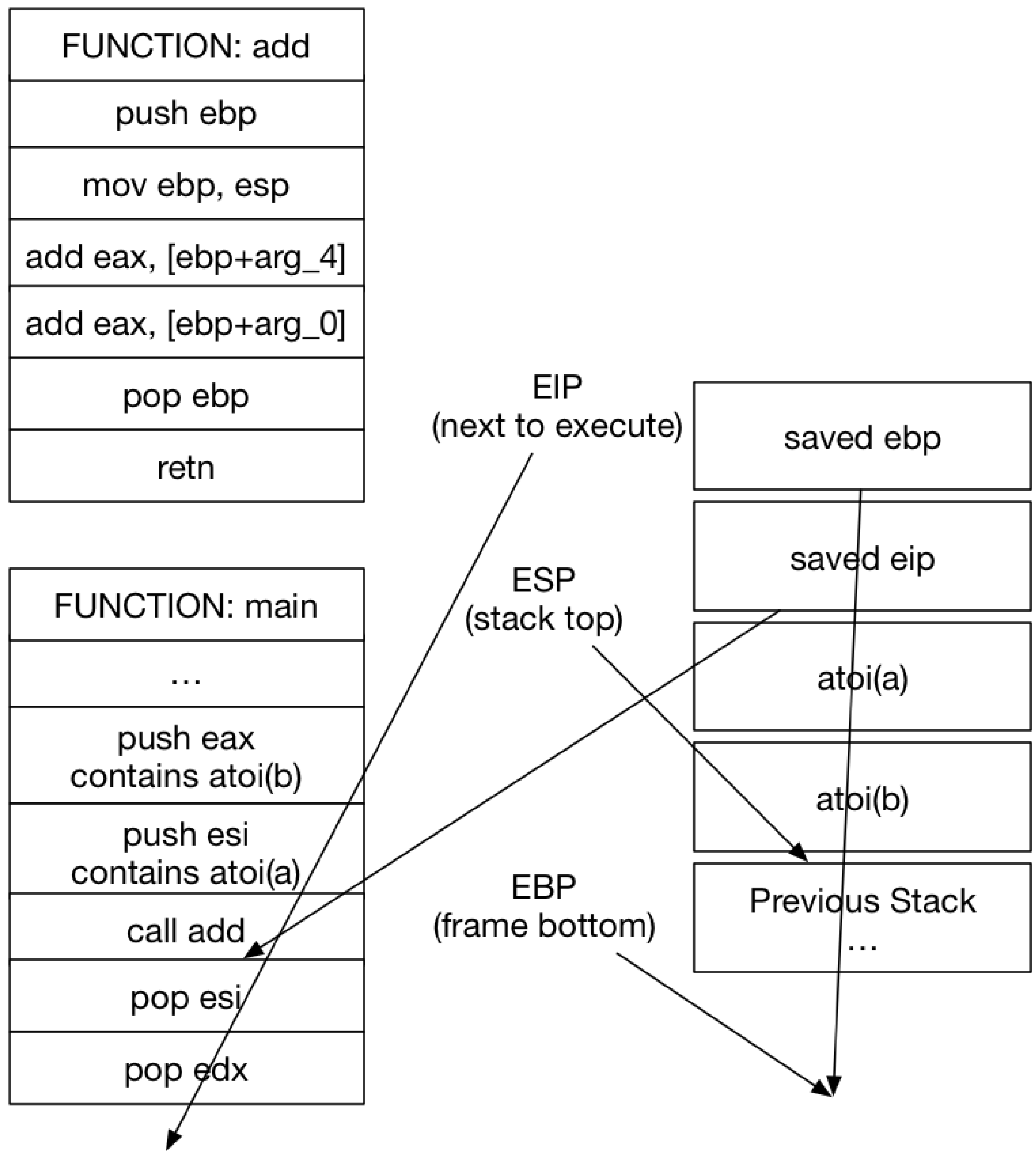
Step by step CDECL 8



Step by step CDECL 9

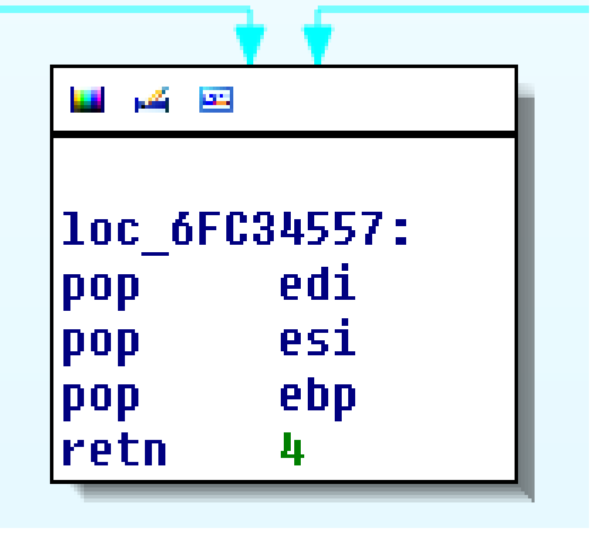


Step by step CDECL A



stdcall

- **Stdcall**, most widely used in win32 land
- Arguments also pushed right to left
- The callee is responsible for cleaning up the stack.
- You'll see something like **RET 8** at the end of the function



More calling conventions

- **Fastcall**
  - Arguments are passed in through registers
  - Not standardised
- **Thiscall**
  - ‘this’ class pointer is passed in through a register, probably **ecx**
- But really compilers do whatever they want internally, only have to be inter-operable when the functions can be called externally

Data type recognition

- It is useful to recognise data types and reconstructure structures when reversing
- In assembly code, types are apparent by how they are used
- The size of a variable is indicated by which instruction accesses it
- Pointer types also become apparent as they are dereferenced
- Data type size and signedness can be inferred from instructions used to access/test/compare it
- First, a quick refresher on integer types

Identifying data types

Example of unsigned char pointer, often used when processing strings or arbitrary byte streams

```
.text:76D834A6                 lea     ecx, [eax+1]
.text:76D834A9                 mov     esi, ecx
.text:76D834AB                 movzx   ecx, byte ptr [ecx] ; ecx is an unsigned char *
.text:76D834AB                                         ; moves a single byte, and uses movzx, not movsx
.text:76D834AE                 lea     eax, [eax+ecx+2]
.text:76D834B2                 cmp     eax, [ebp+arg_4]
```

```
.text:65E36DF7
.text:65E36DF7 loc_65E36DF7:                           ; CODE XREF: DisectUrl(ushort *,ushort * *,ushort * *,ushort
.text:65E36DF7                 dec     esi
.text:65E36DF8                 dec     esi
.text:65E36DF9                 cmp     word ptr [esi], ':' ; esi is a word pointer - WCHAR * in this case
.text:65E36DFD                 jnz     short skip_to_next
.text:65E36DFF                 cmp     esi, ebx
.text:65E36E01                 jbe     short end_of_search
.text:65E36E03                 dec     esi             ; decrementing esi twice = move back 1 WCHAR in the string
.text:65E36E04                 dec     esi
.text:65E36E05                 cmp     word ptr [esi], ':'
.text:65E36E09                 jz      short found_pattern ; found "::"
```

Data type recognition

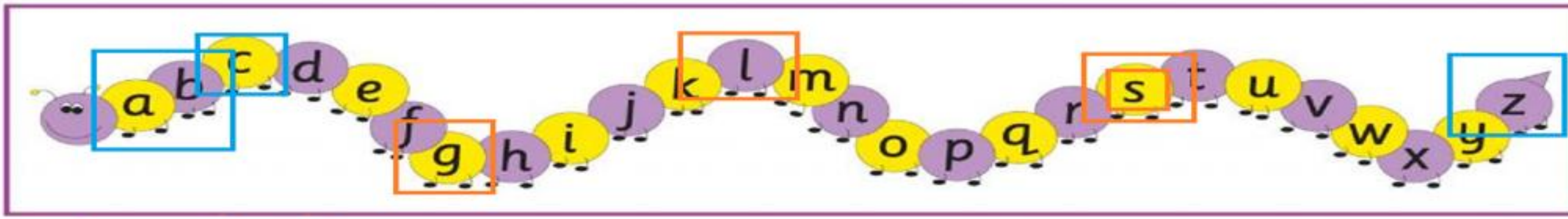
ASM OPERATION	IMPLICATION	EXAMPLE
[dereference]	Operand is a pointer	cmp ecx, [edi] ; edi is a pointer
Data size [dereference]	Operand is a pointer to data values of indicated size	movzx ecx, byte ptr [eax+5Ah] ; [eax+5Ah] is a ; pointer to a byte
movsx/sal/sar/idiv	Source operand is signed	movsx edx, word ptr [eax+80h] ; [eax+80h] points ; to a signed short
movzx/shl/shr/div	Source operand is unsigned	movzx edi, di ; di is an unsigned short

jle/jge/jle/jl	Previous flag-setting operation was dealing with signed operands	mov ebx, 10h
		cmp ecx, ebx
		jle short error_epilog2
		; ecx is signed

jae/ja/jbe/jb	Previous flag-setting operation was dealing with unsigned operands	cmp [esi+4], edi jbe short error_epilog2 ; [esi+4] is unsigned

THE SIGNEDNESS? WORM

unsigned, the original integer type :P
ja - jump above / jae - jump above equal
jb - jump below / jbe - jump below equal
movzx - move. zero extend
jc - jump if carry flag set - an alias for jb



signed, the newcomer:
jg - jump greater / jge - jump greater equal
jl - jump less / jle - jump less equal
movsx - move, sign extend

js - jump if sign flag is set

Recognising data structures

- Data structure recognition is a crucial part of understanding an application
  - User data is often used to populate a data structure, and then used later
  - Required to track what data is controllable directly or indirectly
  - Establish data flow throughout application
- Easily recognizable
  - Allocation of a fixed size (usually)
  - Populated using constant offsets from structure base
  - Data type identification used to identify types of each field
  - Context of use can give further clues to data type:
    - Constant(s) being OR'd with value (e.g. or [eax+1Ch], 8) > suggests that these values are flags/bitfields
    - Comparing against immediate constants (e.g. cmp [eax+1Ch], 10h) suggests we're dealing with an integer

Arrays

- Arrays are usually pretty easy to spot
  - Normally have a base pointer
  - And an offset * size of each object in the array
  - If it’s part of a larger structure (like a class or a struct) it might have an offset to the start of the array which is a static number

mov eax, [esi+edi*4+18h]

- Here **esi** has the base pointer of the object, the array of **4** byte values starts at offset **0x18** and **edi** is the index into that array.

C++ classes

- Classes are just structures in memory
- Usually identifiable as classes by calling conventions (eg. passed as ecx in thiscall function)
- Immediately after allocation, function called that initializes the structure (constructor)
- Polymorphism for classes shows up in binaries using vtables
  - Vtables – arrays of function pointers to virtual functions
  - Usually located at offset 0 of the class structure
  - Functions called indirectly
  - Vtables initialized in constructor function

Constructor

```
; int __stdcall ArrayObj__ArrayObj(LPLONG lpAddend, int)
??0ArrayObj@@IAE@PAVCSession@@PAVVAR@@@Z proc near
                                        ; CODE XREF: ArrayProtoObj::ArrayProtoObj(CSession *,VAR *)+F↑p
                                        ; ArrayObj::Create(ArrayObj * *,CSession *,int,VAR *,VAR *,int)+19↓p

lpAddend        = dword ptr  8
arg_4           = dword ptr  0Ch

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                push    offset ?cbstrArray@@3UConstBstr@@A ; int
                push    [ebp+arg_4]     ; int
                mov     esi, ecx        ; NameTbl is a base class of ArrayObj - call base constructor
                push    [ebp+lpAddend]  ; lpAddend
                call    ??0NameTbl@@IAE@PAVCSession@@PAVVAR@@1@Z ; NameTbl::NameTbl(CSession *,VAR *,VAR *)
                and     dword ptr [esi+58h], 0 ; initialize [esi+58] to 0
                mov     dword ptr [esi], offset ??_7ArrayObj@@6B@ ; const ArrayObj::`vftable'
                mov     eax, esi        ; [esi+0] is a VTable - seen below
                pop     esi
                pop     ebp
                retn    8
??0ArrayObj@@IAE@PAVCSession@@PAVVAR@@@Z endp

; ---------------------------------------------------------------------------
                align 10h
; const ArrayObj::`vftable'
??_7ArrayObj@@6B@ dd offset ?QueryInterface@ArrayObj@@UAGJABU_GUID@@PAPAX@Z
                                        ; DATA XREF: ArrayObj::ArrayObj(CSession *,VAR *)+1C↑o
                                        ; ArrayObj::QueryInterface(_GUID const &,void * *)
                dd offset ?AddRef@NameTbl@@UAGKXZ ; NameTbl::AddRef(void)
                dd offset ?Release@NameTbl@@UAGKXZ ; NameTbl::Release(void)
                dd offset ?GetTypeInfoCount@CTypeLibWrapper@@UAGJPAI@Z ; CTypeLibWrapper::GetTypeInfoCount(uint
                dd offset ?GetTypeInfo@NameTbl@@UAGJIKPAPAUITypeInfo@@@Z ; NameTbl::GetTypeInfo(uint,ulong,ITyp
                dd offset ?GetIDsOfNames@NameTbl@@UAGJABU_GUID@@PAPAGIKPAJ@Z ; NameTbl::GetIDsOfNames(_GUID con
```

Pattern Recognition

- Introductory reversing patterns…
  - Reversing applications takes a lot of practice
  - Certain code constructs occur over and over
  - These become easy to identify quickly

- Reversing is the art of discerning a bigger picture from the most minute details
  - Pattern recognition is the first step in this direction
  - Chain patterns together to figure out what a function is doing
  - Will require understanding of structures that maintain state (more on that shortly)
  - Patterns are necessarily compiler dependent (and optimization level)
  - We will focus on recent MS compilations (what is most commonly used for MS binaries)

Operator NEW[]

C++ new operator – note implicit integer overflow check using ‘seto’, “set byte if overflow”

```
.text:73E9699C    mov     eax, [ebp+var_1C]
.text:73E9699F    push    4
.text:73E969A1    pop     edx
.text:73E969A2    mul     edx
.text:73E969A4    seto    cl
.text:73E969A7    neg     ecx
.text:73E969A9    or      ecx, eax
.text:73E969AB    push    ecx             ; size_t
.text:73E969AC    call    ??2@YAPAXI@Z    ; operator new(uint)
.text:73E969B1    pop     ecx
.text:73E969B2    mov     [ebp+var_20], eax
.text:73E969B5    test    eax, eax
.text:73E969B7    jz      short loc_73E96A27
```

```
int *var_20 = new int[var_1C];
```

```
push    4Ch             ; unsigned int
call    ??2@YAPAXI@Z    ; operator new(uint)
xor     edx, edx        ; allocate structure (total size = 0x4C bytes)
cmp     eax, edx
pop     ecx
jnz     short loc_77163EEE ; check if operator new() returned successfully
mov     eax, 8007000Eh
jmp     short loc_77163F6B
------------------------------------------------------------

                        ; CODE XREF: CDispTypeInfo::GetTypeAttr(tagTYPEATTR * *)+11↑j
mov     ecx, [ebp+arg_0]
push    esi
mov     esi, [ecx+0Ch]
mov     [eax+28h], esi  ; [eax+28] = DWORD (unknown type - (signed) int/pointer?)
mov     esi, [ecx+8]
mov     [eax+10h], esi  ; same with [eax+10]
mov     [eax+38h], dx   ; [eax+38], [eax+3A], [eax+2E] are all shorts (unknown sign)
mov     [eax+3Ah], dx
mov     [eax+2Eh], dx
mov     esi, [ecx+0Ch]  ; [eax+28] appears to be an integer based on how it's used
sub     esi, 3
push    edi
jz      short loc_77163F23
dec     esi
dec     esi
jnz     short loc_77163F32
mov     [eax+2Ch], dx   ; [eax+2C] and [eax+30] also appear to be shorts
mov     word ptr [eax+30h], 1
jmp     short loc_77163F32
```

memcpy() – Windows Vista forward

The memcpy() function is optimized as rep movsd (changed in Vista!)

```
.text:77B23EB1                mov     esi, [edi+20h]
.text:77B23EB4                mov     edi, eax
.text:77B23EB6                mov     ecx, ebx
.text:77B23EB8                mov     eax, ecx
.text:77B23EBA                shr     ecx, 2
.text:77B23EBD                rep movsd
.text:77B23EBF                mov     ecx, eax
.text:77B23EC1                mov     eax, [ebp+arg_0]
.text:77B23EC4                and     ecx, 3
.text:77B23EC7                rep movsb
.text:77B23EC9                add     [eax+20h], ebx
```

```
memcpy(dst, some_structure->offset_20, size);
```

- Or it's just the function:

```
push    20h             ; n
push    eax             ; src
push    0               ; dest
call    _memcpy
```

strlen()

```
argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    edi
mov     eax, [esp+4+argv]
mov     edi, [eax+4]
mov     eax, 0
mov     ecx, 0FFFFFFFFh
repne scasb
mov     eax, ecx
not     eax
sub     eax, 1
pop     edi
retn
main endp
```

- **scasb**: Will compare the byte at AL with the byte value in ES:EDI and sets the flags accordingly.
- **repne**, 'repeat not equal' executes **ECX** times while scasb hasn't found *(edi) == al

wstrlen()

- W is for wide char, which means each 'letter' is 2 bytes wide. You'll see this on Windows more.
- Below **eax** is a pointer to a wchar string

```
.text:00401010 loc_401010:                                          ; CODE XREF:
.text:00401010                 mov     cx, [eax]
.text:00401013                 add     eax, 2
.text:00401016                 test    cx, cx
.text:00401019                 jnz     short loc_401010
.text:0040101B                 sub     eax, edx
.text:0040101D                 sar     eax, 1
```

switches

Switches are often compiled into a jump table and possibly a byte array...

```
.text:2918B30A                                        ; ATL::CCom
.text:2918B30A                 movzx   eax, cx
.text:2918B30D                 cmp     eax, 48h
.text:2918B310                 mov     [esi], cx
.text:2918B313                 jg      short loc_2918B380
.text:2918B315                 jz      short loc_2918B37C
.text:2918B317                 dec     eax
.text:2918B318                 dec     eax
.text:2918B319                 cmp     eax, 15h
.text:2918B31C                 ja      loc_2918B40D
.text:2918B322                 movzx   eax, ds:byte_2918B47B[eax]
.text:2918B329                 jmp     ds:off_2918B463[eax*4]
.text:2918B330
```

```
if(value > 0x48)
    action_2918B380();
else if (value == 0x48)
    action_2918b37C();

value -= 2;

switch(value)
{
    ...
}
```

```
.text:2918B463 off_2918B463    dd offset loc_2918B36F   ; DATA XREF: ATL::CComVariant::ReadFrc
.text:2918B467                 dd offset loc_2918B374
.text:2918B46B                 dd offset loc_2918B378
.text:2918B46F                 dd offset loc_2918B330
.text:2918B473                 dd offset loc_2918B355
.text:2918B477                 dd offset loc_2918B40D
.text:2918B47B byte_2918B47B   db 0                     ; DATA XREF: ATL::CComVariant::ReadFrc
.text:2918B47C                 dd 2020101h, 1030502h, 5030500h, 40405h, 1020201h, 0CCCCCC01h
.text:2918B494                 db 2 dup(0CCh)
.text:2918B496
```

Snitches get switches, yo

Can be also a bunch of **DEC/jXX** statements

```
.text:291A486A
.text:291A486A loc_291A486A:                           ; CODE X
.text:291A486A                 sub     eax, 8
.text:291A486D                 jz      short loc_291A48A4
.text:291A486F                 dec     eax
.text:291A4870                 jz      short loc_291A489C
.text:291A4872                 dec     eax
.text:291A4873                 jz      short loc_291A4894
.text:291A4875                 dec     eax
.text:291A4876                 jz      short loc_291A488C
.text:291A4878                 sub     eax, 8
.text:291A487B                 jz      short loc_291A4884
.text:291A487D
.text:291A487D loc_291A487D:                           ; CODE X
.text:291A487D                 mov     eax, 80070057h
.text:291A4882                 jmp     short loc_291A48BD
.text:291A4884 ; ---------------------------------------------------------------
.text:291A4884
.text:291A4884 loc_291A4884:                           ; CODE X
.text:291A4884                 mov     esi, [ebp+arg_0]
.text:291A4887                 add     esi, 6Ch
.text:291A488A                 jmp     short loc_291A48AA
.text:291A488C ; ---------------------------------------------------------------
.text:291A488C
.text:291A488C loc_291A488C:                           ; CODE X
.text:291A488C                 mov     esi, [ebp+arg_0]
.text:291A488F                 add     esi, 30h
.text:291A4892                 jmp     short loc_291A48AA
.text:291A4894 ; ---------------------------------------------------------------
```

```
call    dword ptr [ecx+18h]
```

Indirection – A PITA

- Indirection is a bit of a pain
  - Indirect calls are made when code utilizes function pointers, classes with virtual methods, or COM objects
  - Register or memory based call instructions used
  - Example: **call [ecx+8]** as opposed to **call sub_12345678;**
- Standard C++ Objects
  - Key is to locate the constructor
  - Constructor tells us: offsets in objects to vtables, location of vtables, and possibly type information for a large number of member variables
  - Constructor called right after object is allocated (or early in a function for stack objects)
  - Pretty distinctive – setup several vtables, and also setting a large number of member variables to 0, or allocating members
- Easiest way to deal with this is to it dynamically – break on it and fill in the indirection yourself

Approaches to RE

- Similar to approaches to source code auditing
- Starting at the top:
  - Find main(), off u go son
  - Good for small programs, malware
  - Bad for large programs, can be inefficient
- Starting at user controllable input
  - Good for vulnerability finding / finding parts of the program you can affect
  - Often easy to find (e.g. find socket accept(), files read, etc)
- Finding particular strings or recognisable constructs
  - Good for examining a particular part of the program that you might be interested in, e.g. finding where the string “Please enter serial key” is used
  - Encryption often has easily identifiable patterns of instruction usage and constants
- strace on linux, procmon on windows

strace / ltrace

```
$ cat 2.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char ** argv)
{
    int f = open(argv[1], O_RDONLY);
    return 0;
}

$ gcc 2.c -o 2 -m32
$ strace -i ./2 aaa
<snip all the bs>
[f7fd9c10] open("aaa", O_RDONLY)        = -1 ENOENT (No such file
or directory)
<snip>
```

Write up and examples

http://how2haq.com/re1.htm

http://how2haq.com/re2.htm

Bonus Assignment (Individual) – 2 bonus marks
Remove the copy protection from 010 editor and send a brief write-up of how you did it (what you changed) to comp9447@gmail.com before next week's class.