```
 ___________________________________.
| ;;|                          |;;||
| []|--------------------------|[]||
| ;;|                          |;;||
| ;;|      --------------      |;;||
| ;;|     | COMP6447 |        |;;||
| ;;|      --------------      |;;||
| ;;|  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
| ;;|                          |;;||
| ;;|__________________________|;;||
| ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;||
| ;;;;;;;___________________ ;;;;;||
| ;;;;;;|   ___            |;;;;;;||
| ;;;;;;|  |;;;|           |;;;;;;||
| ;;;;;;|  |;;;|           |;;;;;;||
| ;;;;;;|  |;;;|           |;;;;;;||
| ;;;;;;|  |;;;|           |;;;;;;||
| ;;;;;;|  |___|           |;;;;;;||
\_______|__________________|______||
 ~~~~~^^^^^^^^^^^^^^^^^^^~~~~~~
```

First

Please gib feedback on lectures + tutorials + course content. Always learning :)

Hopefully everyone is setup re: wargame 1 and tooling

- Week 1 setup is required for all further wargames / exams
- Contact your tutor if you are still having trouble

How computers work and Reversing

and maybe some hacks (**exploitation**)

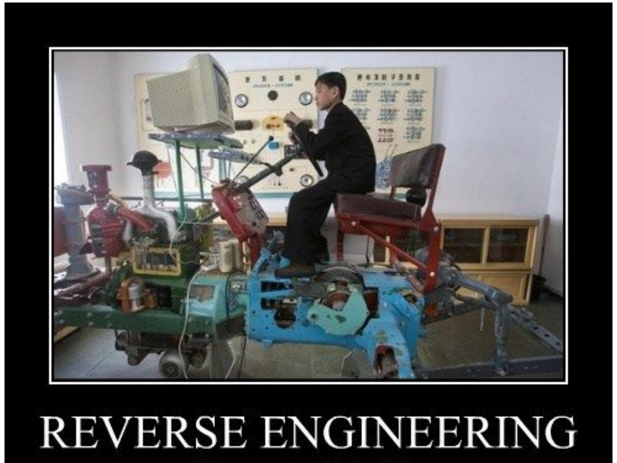
ethics

- Don’t break the law
- Don’t be a dick
- Do Bug Bounties, don’t test networks that you don’t have permission for

what is re?

- reverse engineering
- The opposite of engineering, which makes reverse engineering science
- You take something you don't know, and figure 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.



why do we want to do this?

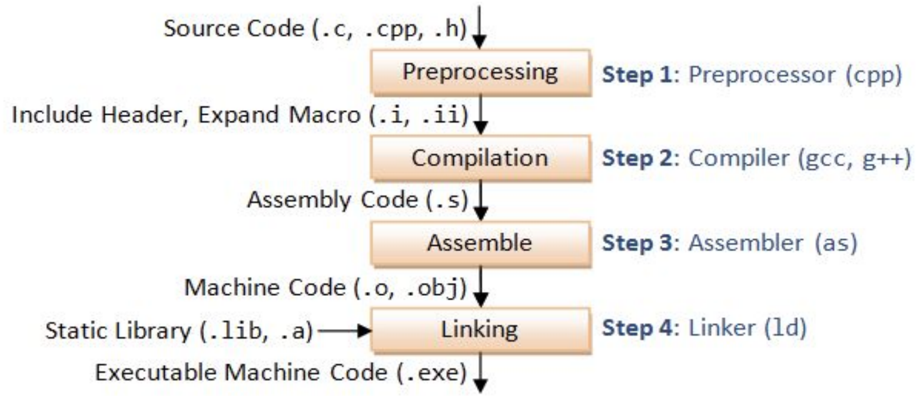
Cool stuff like

- Breaking copyright protection/DRM
- “security testing” – To find vulnerabilities :D
- Malware - Work out how it works to stop it, or even just determine the threat. (or clone it and sell it)
- Crypto – A lot of custom crypto, once you see the source, is ridiculously easy to break.
- understanding programs with no documentation – frequent with drivers
- exploitation is **trivial** if you understand what you’re exploiting
    - Hence we learn reversing first

keygens anyone...??

Compilers

- In layman's terms, compilers take source code and make it into assembly. (not always, but..)
- Assemblers take assembly and make it into machine code.
- Linkers take machine code and make it into an executable program.
- Loaders (normally part of the OS kernel) take a program and make it into a process.
- Source code -> Compiler -> ASM
- ASM -> Assembler -> Machine Code (with offsets)
- Machine Code (with offsets) -> Linker -> Program
- Program -> Loader -> Process


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

how do we uncompile??

- Most machine code is generated from a high level language i.e. c, c++
- We want to reverse the compile and turn the ASM into that ^
- C/C++ generates relatively clean code and will be relatively easy to represent
- Other languages like golang will result in ugly C code
- Some information will be lost e.g. comments, defines, etc
- Sometimes even function names can be stripped from the binary

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)

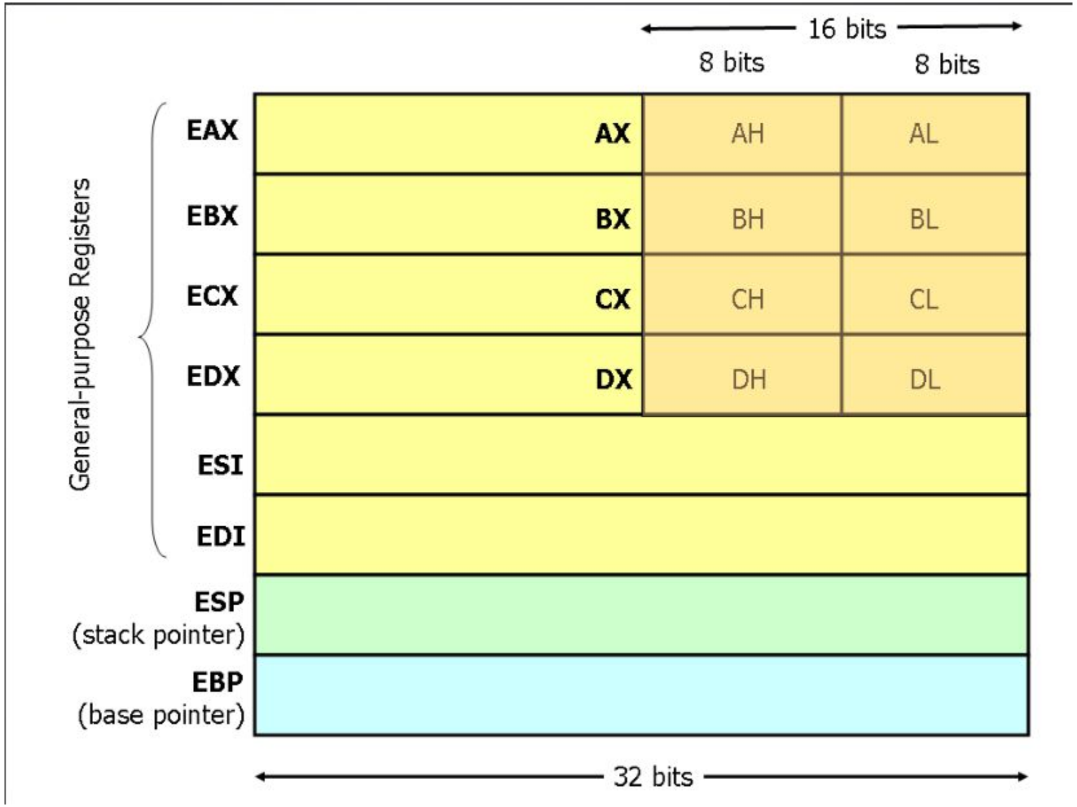
Some registers have special reserved purposes, some do not and are "general purpose", but are frequently used for specific purposes (we'll go into some of these later)

x86 is extremely backwards compatible

This course will deal with x86 only.

What registers does x86 have?

image stolen from University of Virginia Computer Science



tell me more adam

- The first four registers (EAX, EBX, ECX, EDX) can be accessed in 16 and 8 bit sections as well.
- For example:
  - 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.
- No direct way to access the top 16 bits of these registers only.
- x86 used to be sixteen bits! (the E stands for EXTENDED)
- 32 bit x86 is slightly less confusing than 64 bit

more about registers

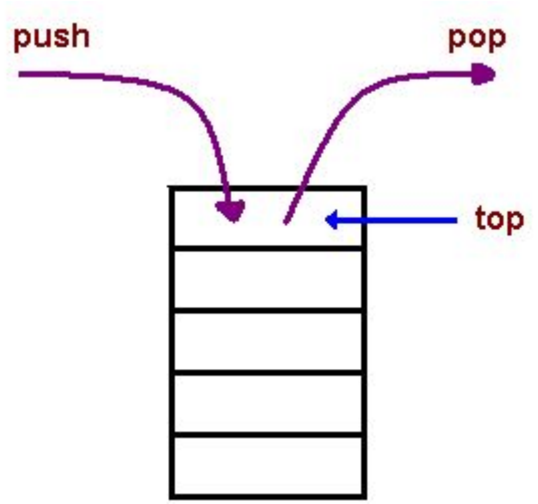
- EIP is the instruction pointer / program counter. It points to the next instruction to be executed. EIP cannot be modified directly.
- EAX, EBX, ECX, EDX, ESI and EDI are all general purpose registers.
- On some architectures EAX is frequently used as the first argument to a function call, and is frequently used as the return code from a function call.

some more on what a stack is

here is a very scientifical diagram

- EBP and ESP are mostly used in relation to the programs “stack”. (They can be manually used for other reasons, but this will break most programs).

LIFO DATA STRUCTURE: Last in, first out.



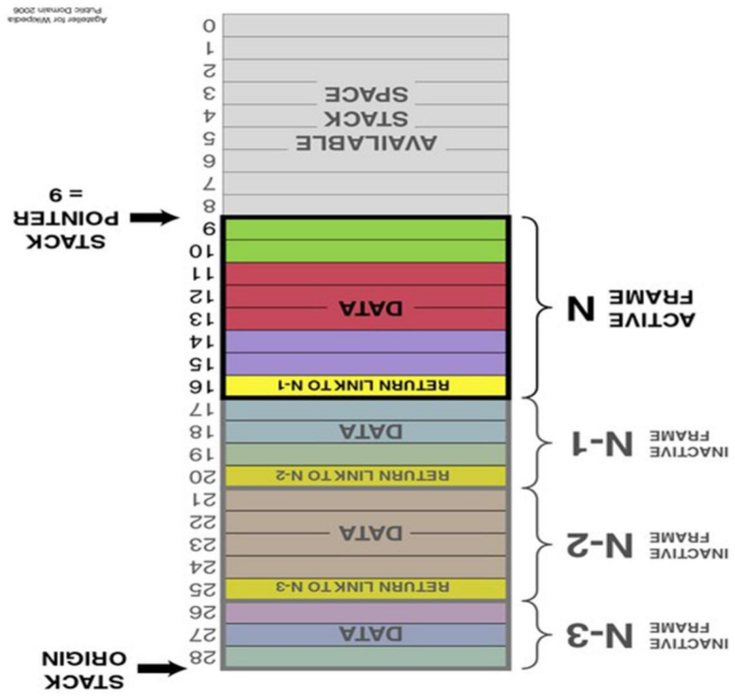
ESP & EBP

- ESP point to the top of the stack! That is, ESP contains the memory address of the top of the stack
- EBP is the "Extended Base Pointer". It is also commonly referred to as the "Frame pointer".

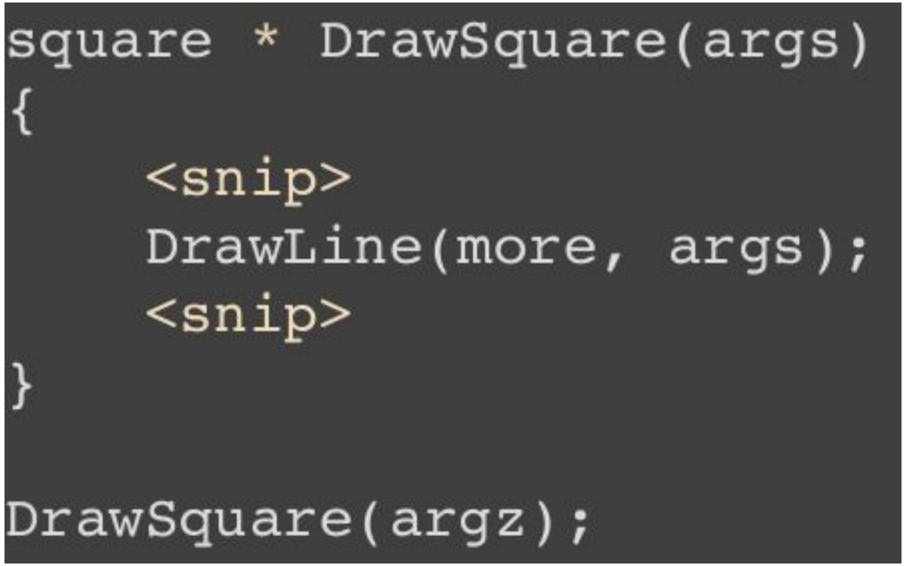
the stack “grows” from high addresses to low addresses

- Together these are used to define a “Stack frame”
- Stack frames are used so that each function has its 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.

this diagram may help

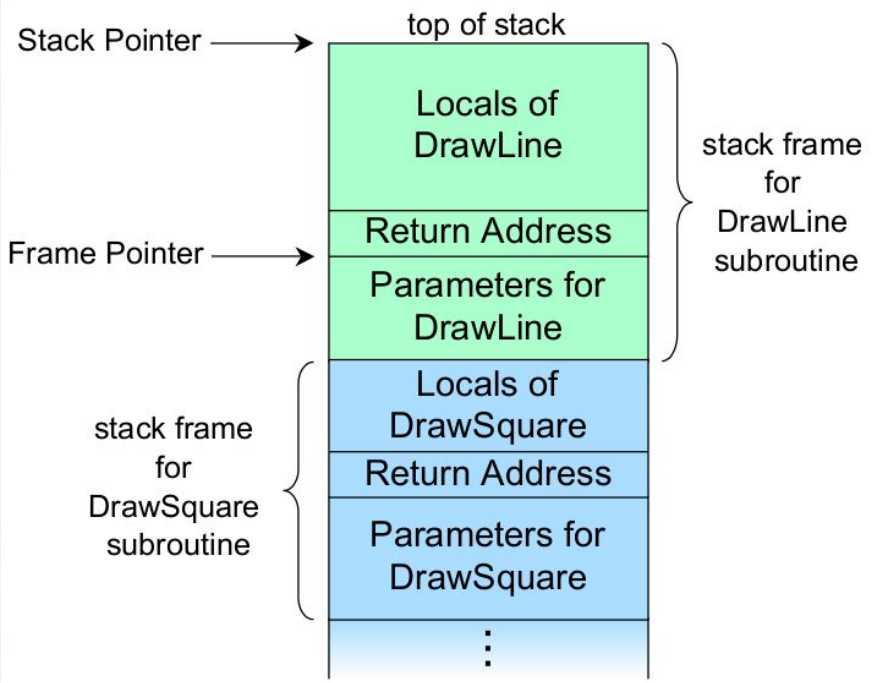


This might actually help



```
square * DrawSquare(args)
{
    <snip>
    DrawLine(more, args);
    <snip>
}

DrawSquare(argz);
```



Patterns - function prologue/epilogue

How does the stack get “setup”?

- 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
```

It will then reserve space for local arguments:

```
ADD ESP, 8
```

<insert drawing on stack movement>

Heap

We will learn much more about the heap in future weaks :-)

These are used for dynamic allocation of memory, and are accessed by calls like malloc() and free().

The heap is not a stack $_{\text{duh}}$

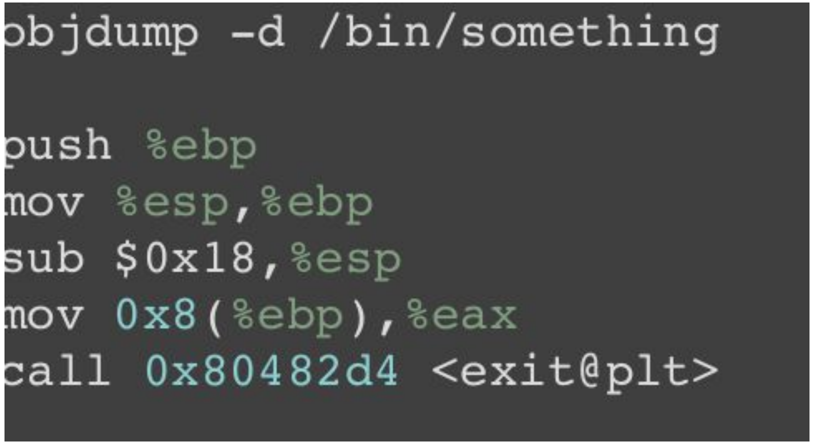
Adam pause for questions

- Next we will talk about some specific x86 instructions you should know for this weeks wargames

First lets talk syntax (AT&T syntax is bad)

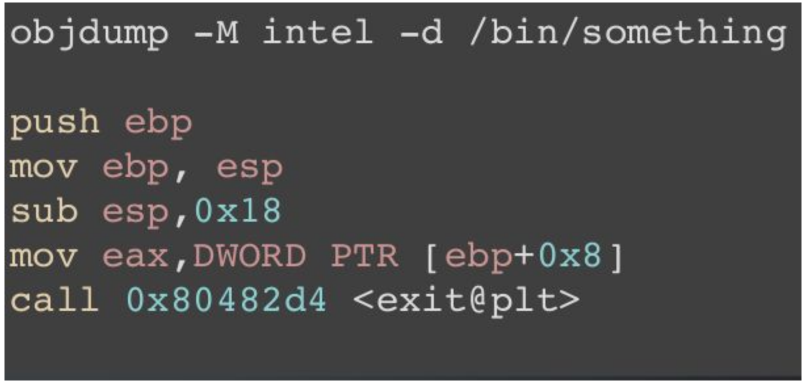
Intel is not the default in GNU, for whatever reason, but make it default in gdb

```
echo "set disassembly-flavor intel" >> ~/.gdbinit
```



```
objdump -d /bin/something

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



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

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

Some useful x86 instructions

- nop
- add/sub
- mov/lea
- jmp
- call
- ret
- int/sysenter/syscall
- and/xor
- test/cmp
- conditional jumps

Nop

- NOP
  - Does nothing
- PUSH/POP <register>
  - Stack manipulation
- ADD/SUB <dest> <source>
  - dest = dest + register
- JMP <offset>
  - Jumps forward/backwards offset amount of bytes
- CALL <address>
  - Push EIP onto stack as return address then jumps to address
- JMP <address>
  - jumps

```
PREFIX(optional) INSTRUCTION destination, source
```

mov

eax = ebx

```
MOV eax, ebx
```

Treat ebx as a value

eax = *ebx

```
MOV eax, [ebx]
```

Treat ebx as a pointer

eax = *(ebx + 4)

Arrays?

```
MOV eax, [ebx+4]
```

Treat ebx as a pointer

```
PREFIX(optional) INSTRUCTION destination, source
```

lea

Load Effective Address

Loads the address of the source operand into the destination

Equivalent to & in C

```
LEA ebx, [ecx+4]
```

==

```
MOV ebx, ecx
ADD ebx, 4
```

Often used for fast multiplication + addition

```
LEA edx, [edi+edx*4]
```

call / ret

call <addr>:

- Push eip
- Jmp <addr>

ret: Opposite of **CALL** instruction

- Restores saved **EIP** from stack
- Equivalent to **POP EIP**

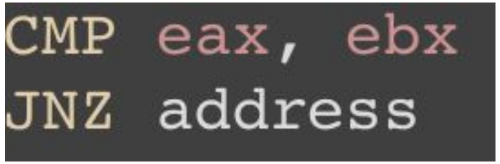
CMP / Test

- Two main instructions used for comparisons
  - TEST and CMP
- TEST <a> <b>
  - AND’s together a and b
  - Sets the flags (Zero flag, Negative/Positive Flags) in EFLAGS
- CMP <a> <b>
  - SUB’s b from a
  - Sets the flags (Zero flag, Negative/Positive Flags) in EFLAGS

All these instructions do is set flags in EFLAGS.

jumping

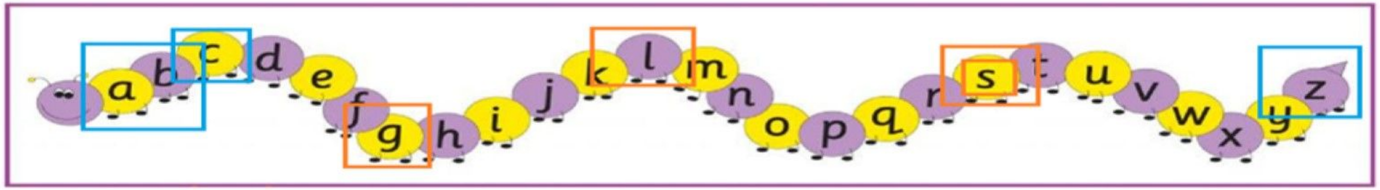
- Now that we have compared two values, we can jump conditionally
- There are dozens of instructions such as
    - JZ jump if zero flag is set
    - JNZ jump if zero flag is not set
    - JGE jump if greater or equal (positive flag is set || zero flag)
    - JLE jump if less or equal   (negative flag is set || zero flag)
    - Some are signed, some are not
- Example ->
    - Jump to address if EAX != EBX
    - JNE and JNZ are essentially the same
- These all just jump based on EFLAGS



```
CMP eax, ebx
JNZ address
```

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



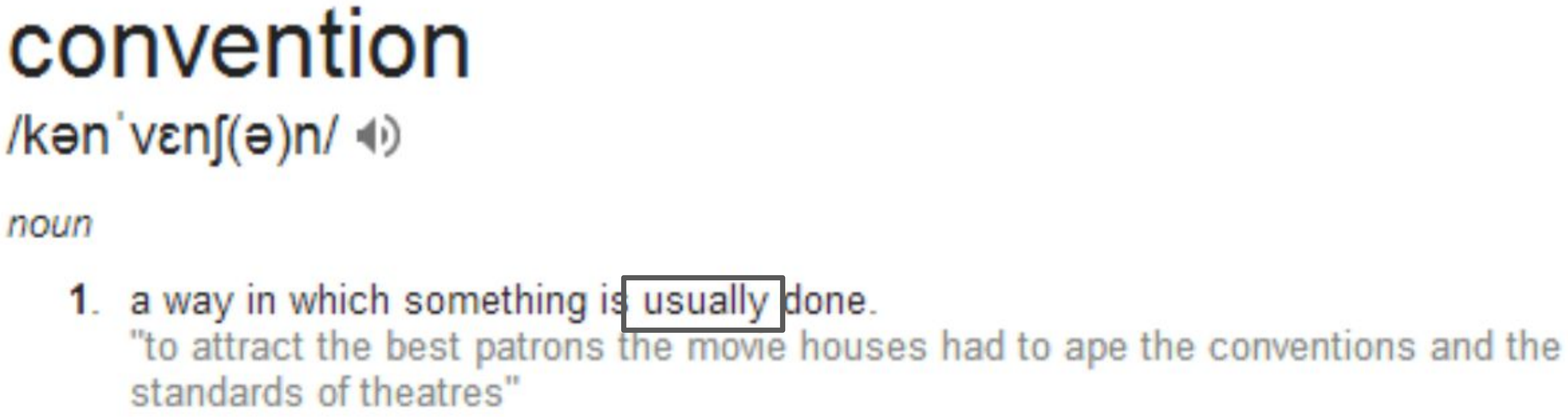
signed, the newcomer:
jg - jump greater / jge - jump greater equal
jl - jump less / jle - jump less equal
movsx - move, sign extend

js - jump if sign flag is set

Calling conventions

- Calling conventions describe how one function should call another in a certain system
- The way **arguments + return address** are passed to a function **differs** by calling convention
- There are a lot of different calling conventions
  - I’ll cover the most common one in linux x86 (CDECL)

importantly



convention

/kənˈvɛnʃ(ə)n/

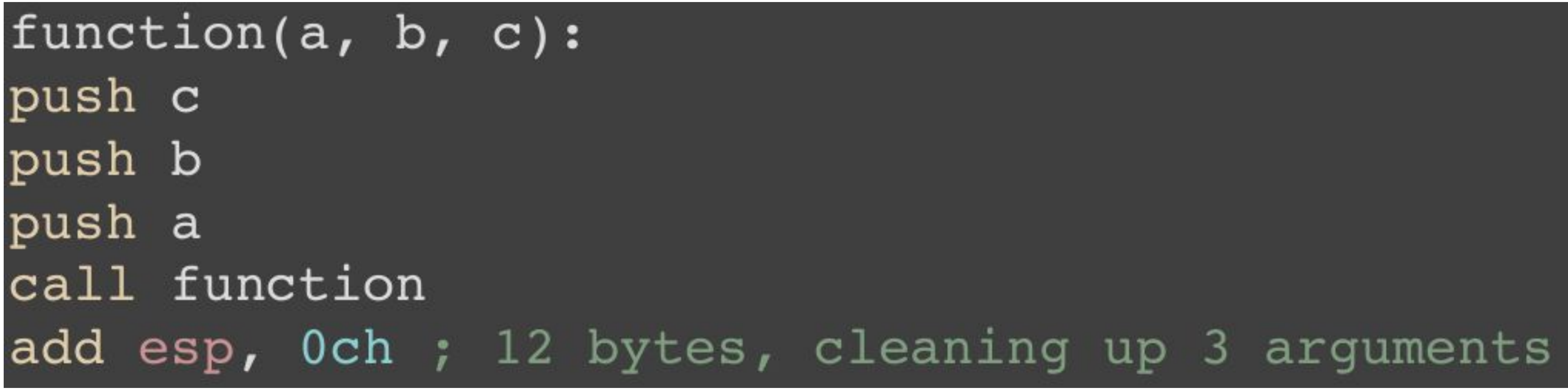
noun

1. a way in which something is $\boxed{\text{usually}}$ done.
   "to attract the best patrons the movie houses had to ape the conventions and the standards of theatres"

cdecl

Arguments are pushed onto the stack in **reverse order**

Callee clean up, return value in **eax**



```
function(a, b, c):
push c
push b
push a
call function
add esp, 0ch ; 12 bytes, cleaning up 3 arguments
```

arguments are often relative to ebp

In cdecl, [ebp + 8] is usually the first argument.

[ebp + 12] Is the second.. And so on.

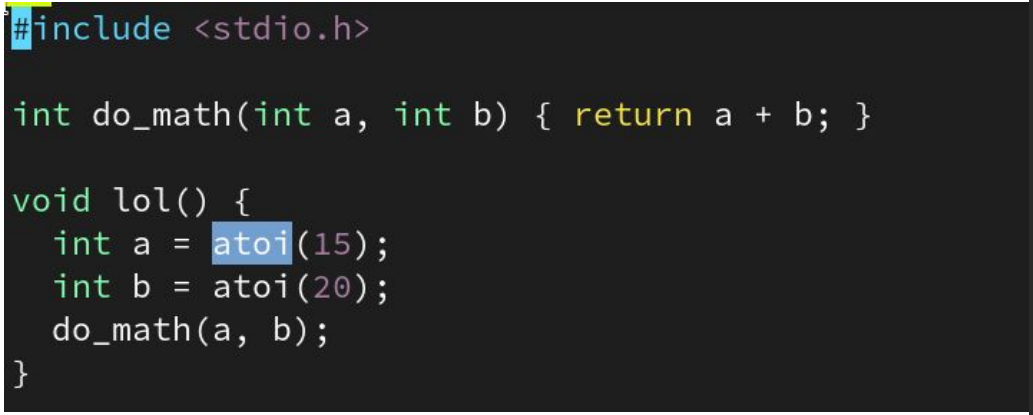
Remember + moves us closer to the top of the stack, as the stack grows to lower addresses.

Local variables go immediately after these (higher up the stack, lower down in memory)

ebp is often static in a stack frame

```
High Address  +-------------+
              | data        |
              |             |
              +-------------+
              | arg1        |  ebp+8
              |             |
              +-------------+
              | return addr |
              |             |
              +-------------+
              | (saved ebp) |  ebp
              |             |
              +-------------+
              | local var   |  ebp-4
              |             |
              +-------------+
              | other stuff |  esp
              |             |
Low Address   +-------------+
```

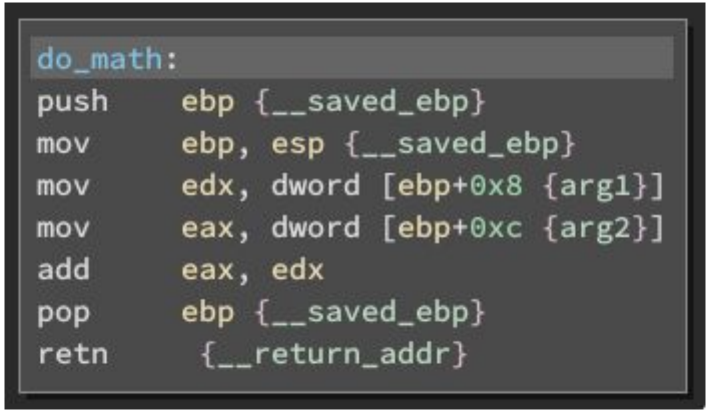
how does cdecl decl?



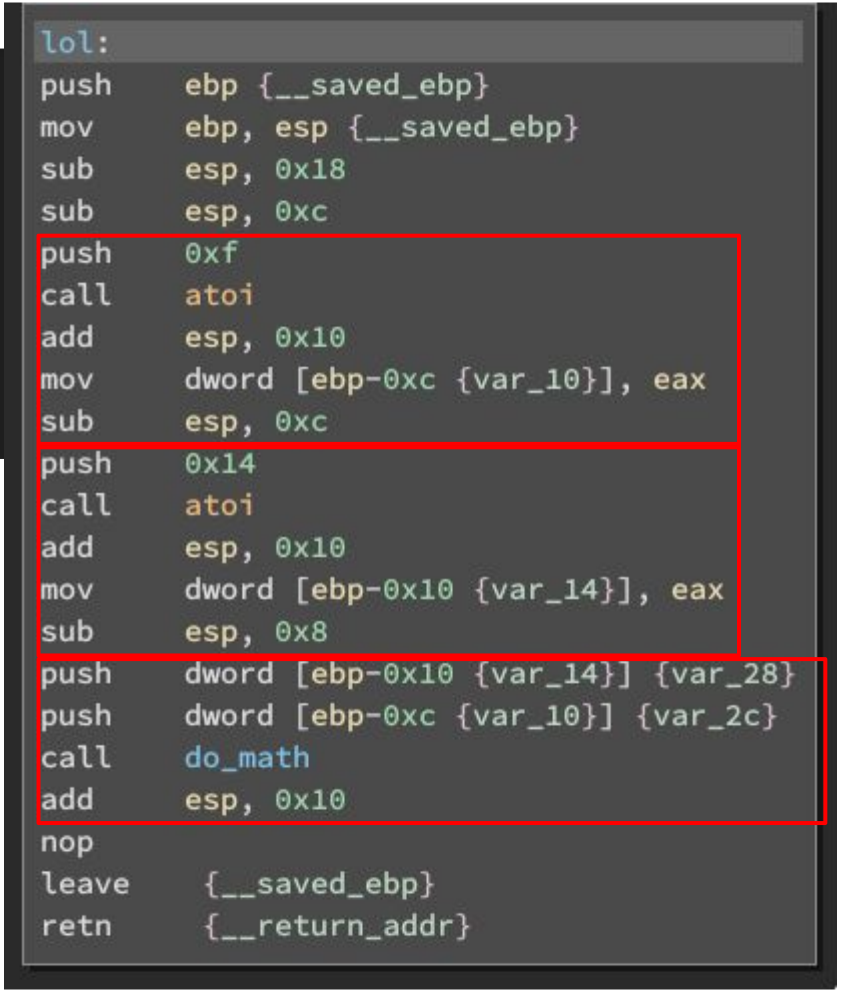
```c
#include <stdio.h>

int do_math(int a, int b) { return a + b; }

void lol() {
  int a = atoi(15);
  int b = atoi(20);
  do_math(a, b);
}
```



```
do_math:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
mov     edx, dword [ebp+0x8 {arg1}]
mov     eax, dword [ebp+0xc {arg2}]
add     eax, edx
pop     ebp {__saved_ebp}
retn     {__return_addr}
```



```
lol:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
sub     esp, 0x18
sub     esp, 0xc
push    0xf
call    atoi
add     esp, 0x10
mov     dword [ebp-0xc {var_10}], eax
sub     esp, 0xc
push    0x14
call    atoi
add     esp, 0x10
mov     dword [ebp-0x10 {var_14}], eax
sub     esp, 0x8
push    dword [ebp-0x10 {var_14}] {var_28}
push    dword [ebp-0xc {var_10}] {var_2c}
call    do_math
add     esp, 0x10
nop
leave    {__saved_ebp}
retn     {__return_addr}
```

draw it out on paper the first time

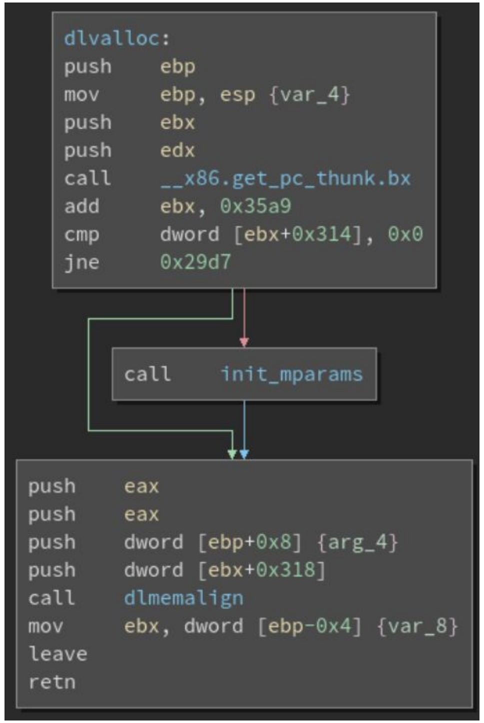
draw the stack yourself, on a paper

draw the value of each register

Disassemblers

Our primary tool, takes a program and makes it into a listing of ASM

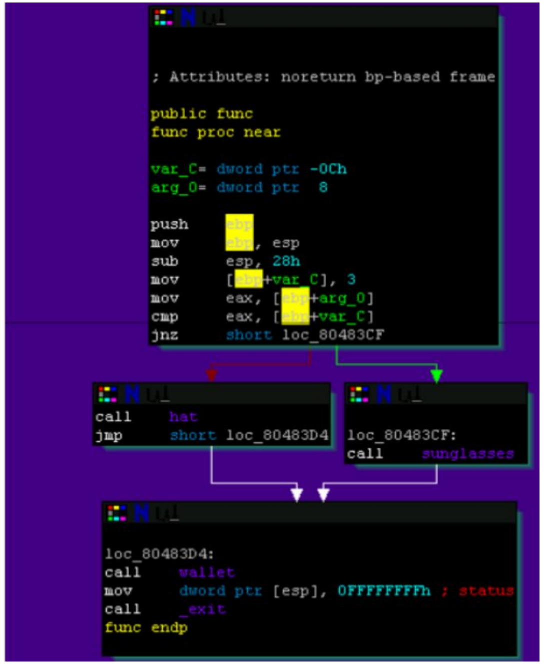
Good disassemblers (IDA Pro, binaryninja, hopper) make our job much easier



```
dlvalloc:
push    ebp
mov     ebp, esp {var_4}
push    ebx
push    edx
call    __x86.get_pc_thunk.bx
add     ebx, 0x35a9
cmp     dword [ebx+0x314], 0x0
jne     0x29d7
```

```
call    init_mparams
```

```
push    eax
push    eax
push    dword [ebp+0x8] {arg_4}
push    dword [ebx+0x318]
call    dlmemalign
mov     ebx, dword [ebp-0x4] {var_8}
leave
retn
```



```
; Attributes: noreturn bp-based frame

public func
func proc near

var_C= dword ptr -0Ch
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_C], 3
mov     eax, [ebp+arg_0]
cmp     eax, [ebp+var_C]
jnz     short loc_80483CF
```

```
call    hat
jmp     short loc_80483D4
```

```
loc_80483CF:
call    sunglasses
```

```
loc_80483D4:
call    wallet
mov     dword ptr [esp], 0FFFFFFFFh ; status
call    _exit
func endp
```

catch us next time for more...

Next week we’ll cover:

- floating point arithmatic
- reversing more complex programming patterns (switches, loops)
- reversing data structures (ints, floats, structs, classes, etc)
- dynamic analysis
    - ltrace/strace
    - gdb
    - Valgrind
- And we will talk a bit about disassemblers + decompilers
- reversing big programs (thing chrome/firefox scale)
- Some more calling conventions

exploitation   (buffer overflows)

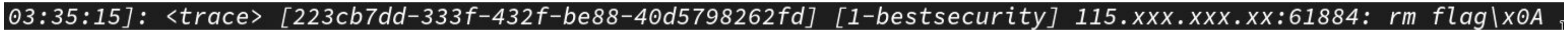
We will be covering:

- What is the length field?

ETHICS

- Don’t break the law
  - Do not assume “good intentions” will help you.
- Bug bounties exist for you to have fun with
- Don’t attack other people’s infrastructure without explicit permission
  - Lawyers are scary trust me 👀 👀
- **Don’t ruin things for others (ie: taking down course infra or removing flags)**

IN MY OPINION Never disclose always sell :)



```
03:35:15]: <trace> [223cb7dd-333f-432f-be88-40d5798262fd] [1-bestsecurity] 115.xxx.xxx.xx:61884: rm flag\x0A
```

Tl;dr security

Security, in my mind, is about **people**:

1. **Assuming** people and computers aren’t **evil**
2. **Assuming** the **length field** is correct

tl;dr exploitation

- Exploitation is the process of attacking a vulnerability:
- Poppin’ a shell, rewtin a box, haqin the gibson.
- We’ll learn how to write basic exploits for low level vulnerabilities
- For practice, you are going to want to turn off ASLR on linux (as root)

```
$ sudo sysctl kernel.randomize_va_space=0
```

(warning makes your box more hackable..)

- Learn more by reading phrack and uninformed:
- Smashing the stack for fun and profit (1996): http://phrack.org/issues/49/14.html

what is a buffer?

char name[16]

What is a buffer overflow

char name[**16**];

gets(name);   (man gets?)

or

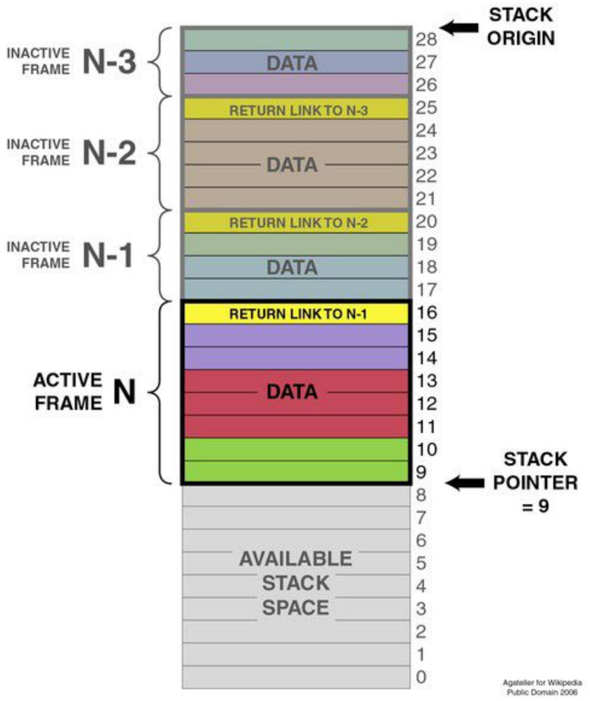
fgets(name, **32**, stdin)

char *fgets(char *s, int size, FILE *stream);

What is a length field?

a bit more on stacks

- Sure we can overflow a buffer.. so what?
- What would we want to **overwrite** on the stack?
- **Other variables**? Maybe **flags** that give the user more permissions?
- other **stuff** (instruction pointers?)



more on smashing that stack

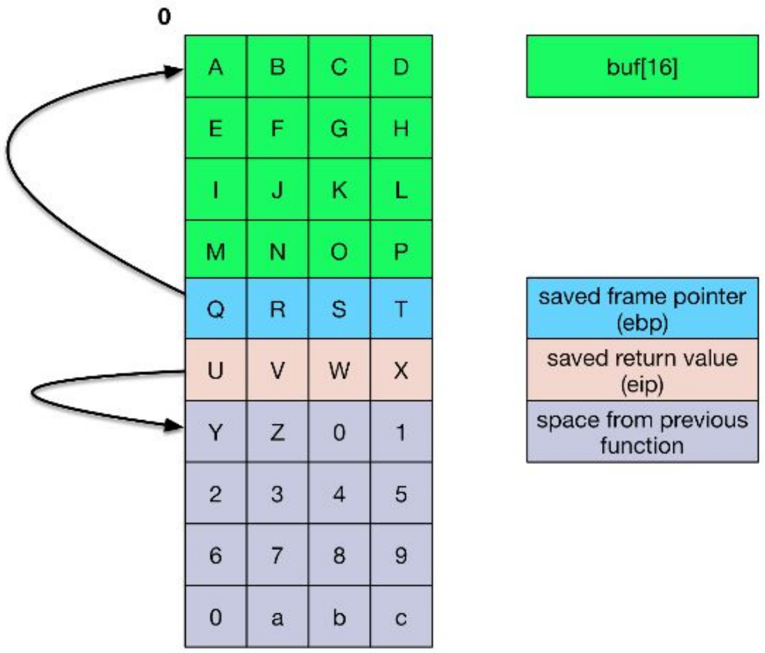
```
void stupid(char *s) {

        char buf[16];

        strcpy(buf, s);

}

stupid("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abc")
```



demo

lets hope i prepared a demo

how does this give us code execution?

Remember the saved instruction pointer on the stack. (hint: we now control that stack lol)

- EIP just contains a pointer to the code to run…
- A pointer is just a 4 byte number.. In little endian. (sounds like intro chal from last week :p)

If we know the address we want to jump to, writing it is easy

exercise is left as a challenge to the reader (tutorials will go more in depth on how to do this) TODO(adam) add a better example here

How can i find a place to jump to?

```
$ objdump -t blind
```

Lists all sections (& functions) in a binary…

Some challenges have a function called win :O

what happens if you jump to win

Wargame 2

- Already released
- Start early

3 challenges to introduce you to buffer overflows

this is bad??

80% of software engineers are bad in my eyes

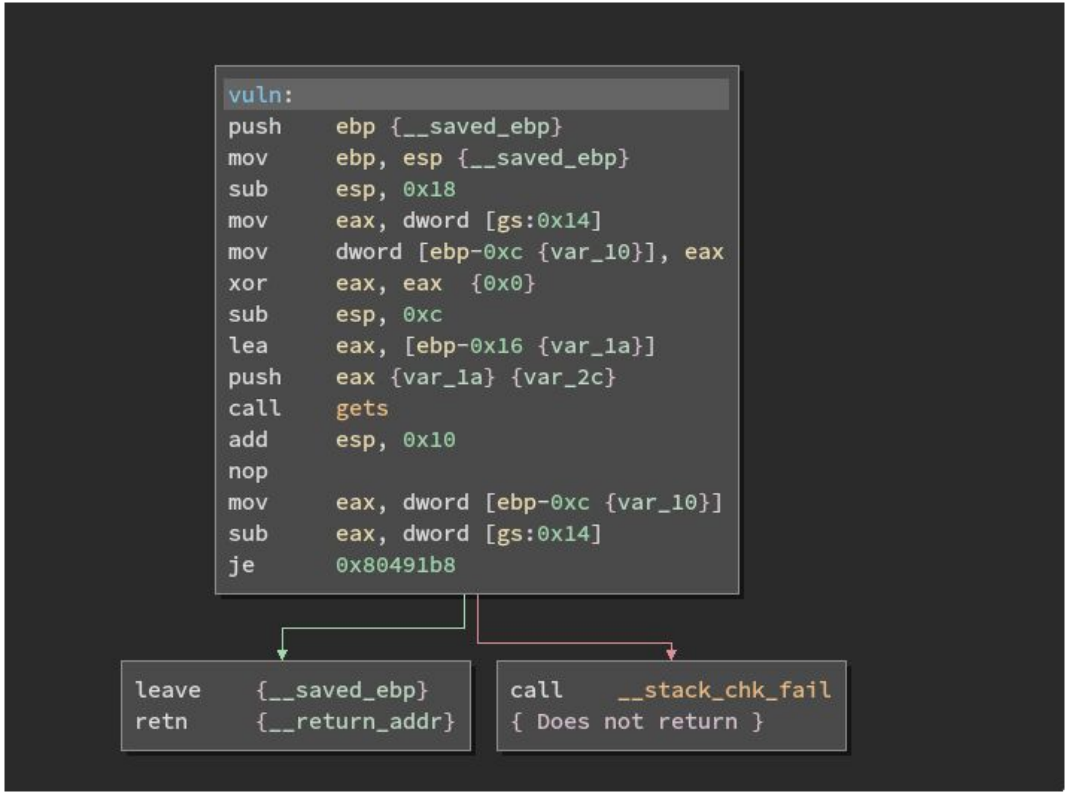
how could this not be a problem in every modern application??

use protection

- Stack cookies/canary
  - Place a random 4 byte value in between the userdata (buffers etc) and the saved return address.
  - Value is checked before function RET to see if has been overwritten
- Variable reordering
  - Sometimes you don’t want to smash the return value, maybe just overwrite a local variable
  - Making all the non-arrays be above the arrays will stop them being overwritten
  - Cannot be 100% effective, can still possible control previous strings, etc.
- Stack overflows (alone) aren’t very exploitable these days but do serve as an excellent introduction to writing exploits

More on canaries

- Canary is placed after EBP
- Overwriting EBP/EIP becomes hard



```
vuln:
push    ebp {__saved_ebp}
mov     ebp, esp {__saved_ebp}
sub     esp, 0x18
mov     eax, dword [gs:0x14]
mov     dword [ebp-0xc {var_10}], eax
xor     eax, eax  {0x0}
sub     esp, 0xc
lea     eax, [ebp-0x16 {var_1a}]
push    eax {var_1a} {var_2c}
call    gets
add     esp, 0x10
nop
mov     eax, dword [ebp-0xc {var_10}]
sub     eax, dword [gs:0x14]
je      0x80491b8
```

```
leave     {__saved_ebp}
retn      {__return_addr}
```

```
call    __stack_chk_fail
{ Does not return }
```

but what if i still want to hack

So how do we defeat a stack canary??

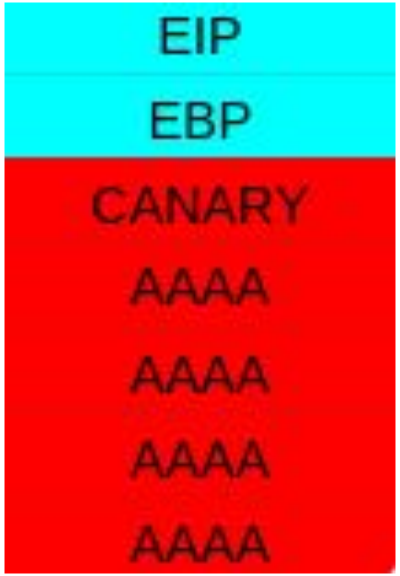
We control the stack with our overflow right?

..ez pz.. We just need to leak the value of the cookie.

Since it's just another value on the stack, if we know what it is… we can put it back, and it will be fineeee

Means we need TWO exploits. One to **leak** the canary and one to **overflow** the stack.

We made this **easier** for you in wargame 2



ASLR

- We can control EIP - now what? It depends on the program and the OS
- Back in the day we could just return to our own code on the stack
  - Overwrite EIP with the static location of our shellcode, win!
  - We can do this on XP but let's be slightly more intelligent
- Address Space Layout Randomisation
  - Randomising the stack/heap/executable/dlls/other important bits location
  - Means on these systems we can't just hardcode an address and win
  - But ASLR support has to be compiled for executables, most don't!

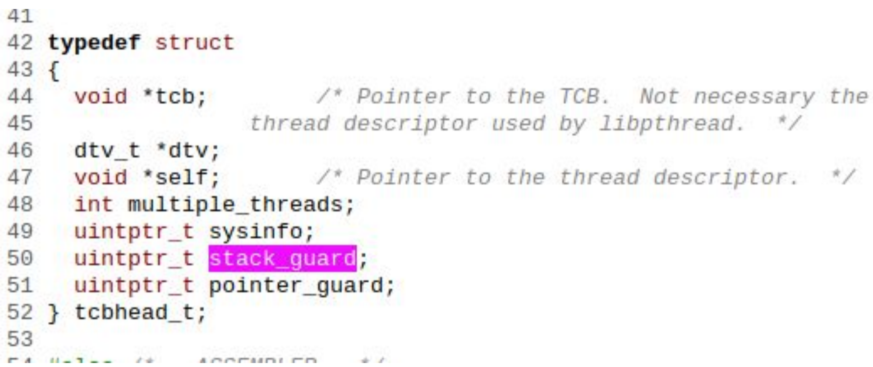
Been in Windows since Vista, Linux since 2.6.12 (2005), PaX since 2001

ASLR / NX / DEP etc. will be covered in detail later in the course!

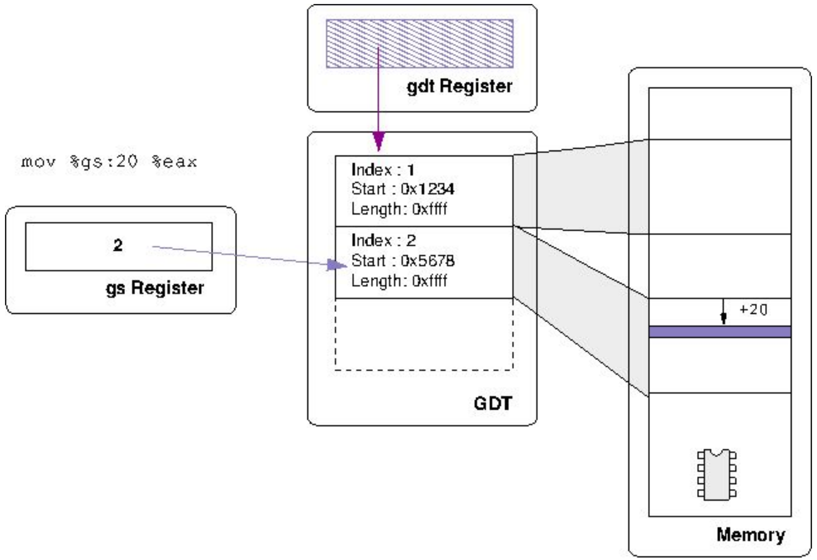
Disabled in Wargame 1

How does a canary actually work?

- The 4 byte stack canary is generated on start
  - Always ends in a null byte (why?)
  - Is always the same, for every function call during a programs/threads life
- GS is a x86 segment register
  - Points to this struct in glbc (thread locale cache)
  - gs:0x14 = struct + 0x14bytes



```
typedef struct
{
  void *tcb;		/* Pointer to the TCB.  Not necessary the
			   thread descriptor used by libpthread.  */
  dtv_t *dtv;
  void *self;		/* Pointer to the thread descriptor.  */
  int multiple_threads;
  uintptr_t sysinfo;
  uintptr_t stack_guard;
  uintptr_t pointer_guard;
} tcbhead_t;
```


mov %gs:20 %eax
2
gs Register
gdt Register
Index : 1
Start : 0x1234
Length: 0xffff
Index : 2
Start : 0x5678
Length: 0xffff
GDT
+20
Memory