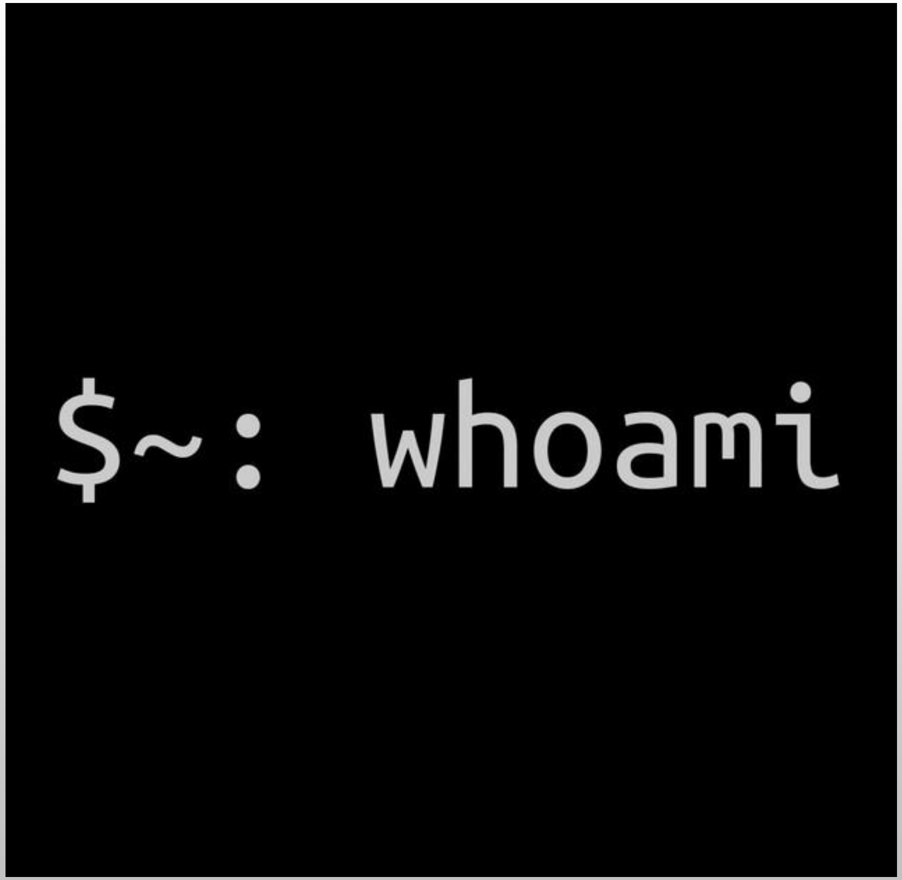
EXPLOIT DEVELOPMENT

ABOUT ME

- Manish Sharma (3+yrs experience in AppSec)
- Security Engineer at Victoria's Secret
- eCXD certified | CEH (expired)
- Github/LinkedIn/Twitter: sh377c0d3



DISCLAIMER

- This entire talk is more at personal level and doesn't contain or relate to any of my former or current professional associations.

- All content has been sourced from publicly available sources like the internet. The presenter does not infringe or claim rights over any content, images and any other work being presented here, and all rights belong to respective owners only.

AGENDA

- Introduction to Exploit Development
- Basic of Windows and Linux Concepts
- Fuzzing and Crash Analysis
- Finding Offset and Overwriting EIP
- Finding Bad Characters
- Stack-based Buffer Overflow

AGENDA (CONTD.)

- Introduction to Egg Hunting
- Return-Oriented Programming (ROP)
- Conclusion and Next Steps

BEFORE WE START

!!!

CHANGES IN LABS AND CHALLENGES

FUN EXAMPLE OF BINARY EXPLOITATION

“[TAS] Super Mario World "Arbitrary Code Execution" in 02:25.19 by Masterjun”

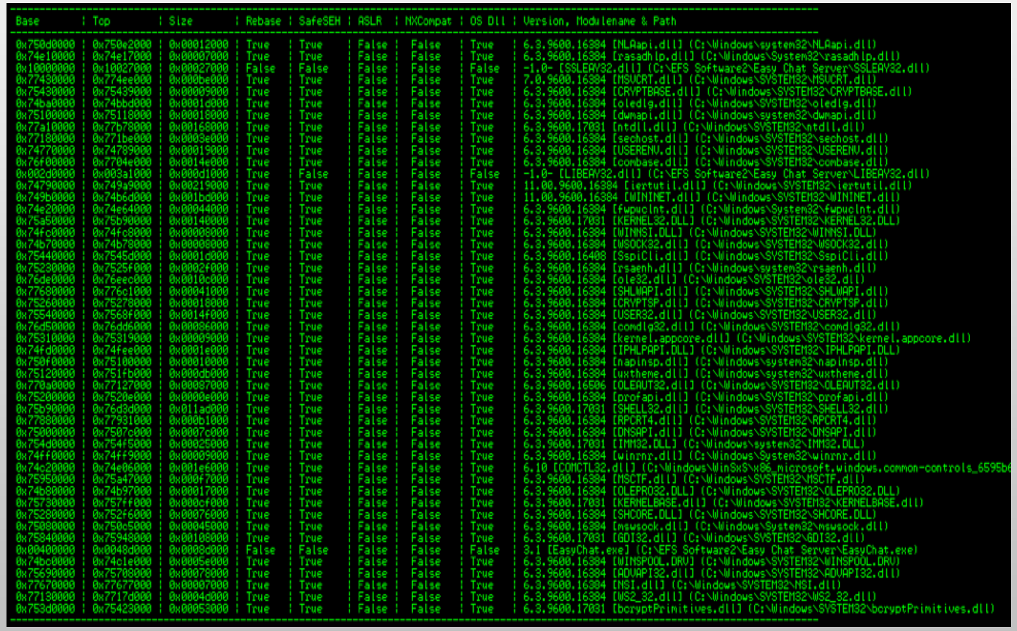
INTRODUCTION TO EXPLOIT DEVELOPMENT

- Exploit development is the process of finding, creating, and developing software or code that takes advantage of a vulnerability in a computer system, network, or application to cause unintended or unauthorized behavior.

- The goal of exploit development is often to gain control of a system, steal sensitive information, or cause damage.

INTRODUCTION TO EXPLOIT DEVELOPMENT

- Identify the Entry Point
- Fuzz the application/software for a crash
- Re-create the crash
- Control the Execution
- Hunt and eliminate bad characters
- Generate shellcode for exploitation
- Obtain a Shell



BASIC OF WINDOWS AND LINUX CONCEPTS

STACK

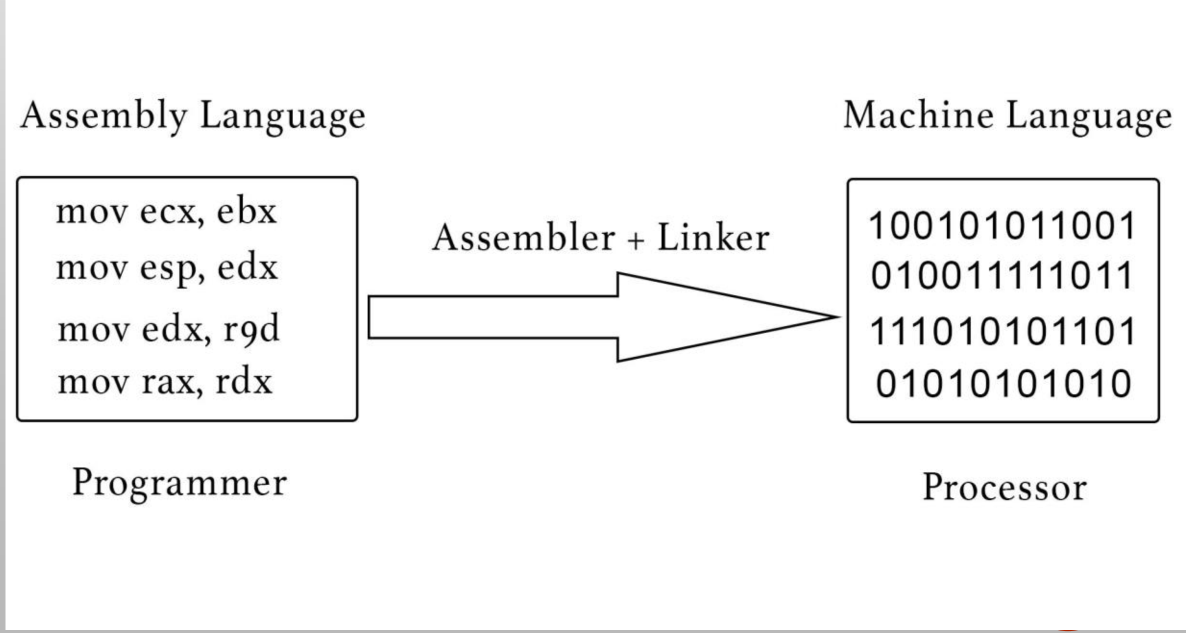
- Stack is an area of memory within a process that is used by the processor to save data.

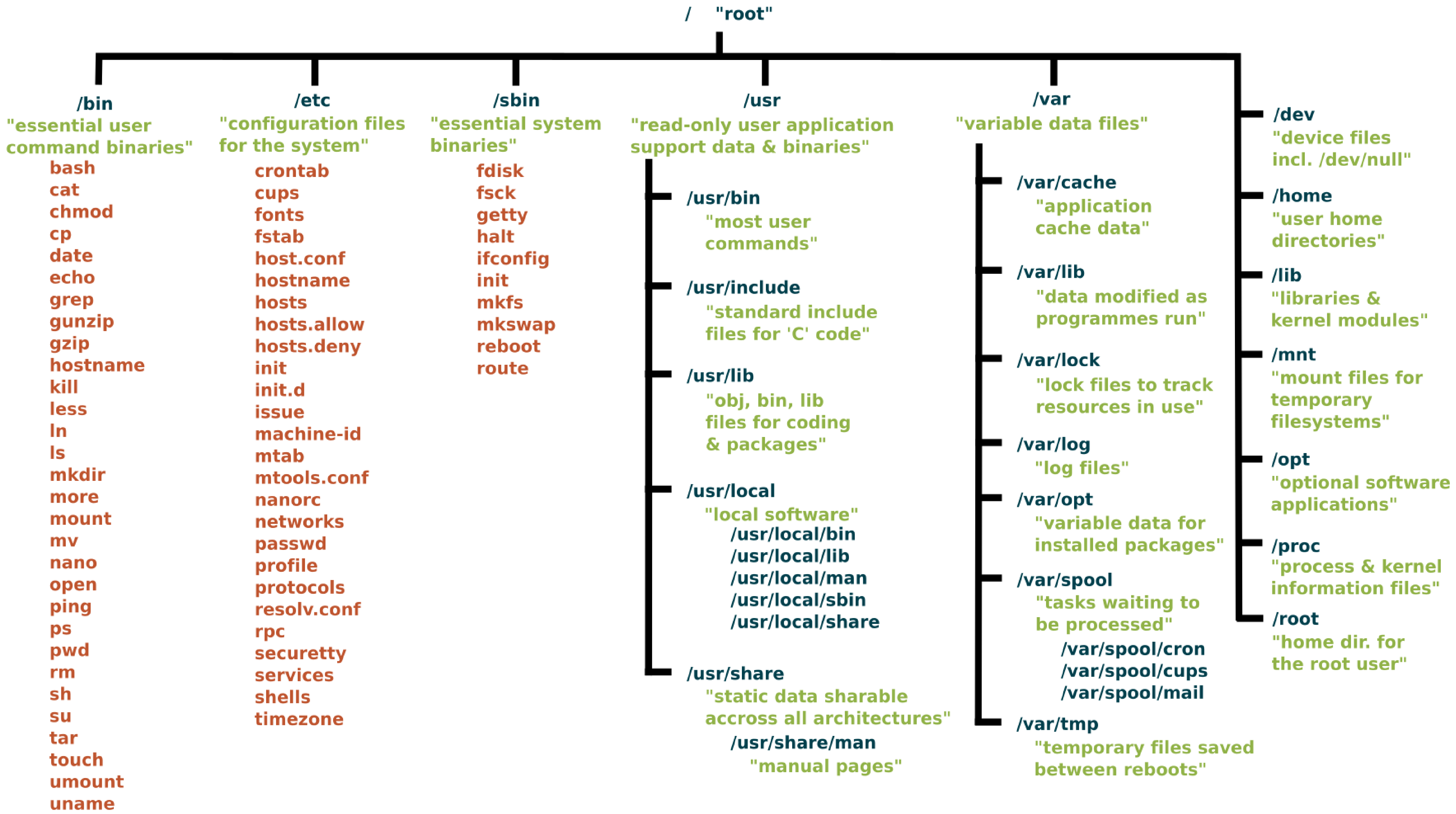
- Registers are small in size but the fastest among all the temporary data storage, the stack offers a large space.

- Stack is also used to track the execution of the program.

LITTLE BIT OF ASSEMBLY

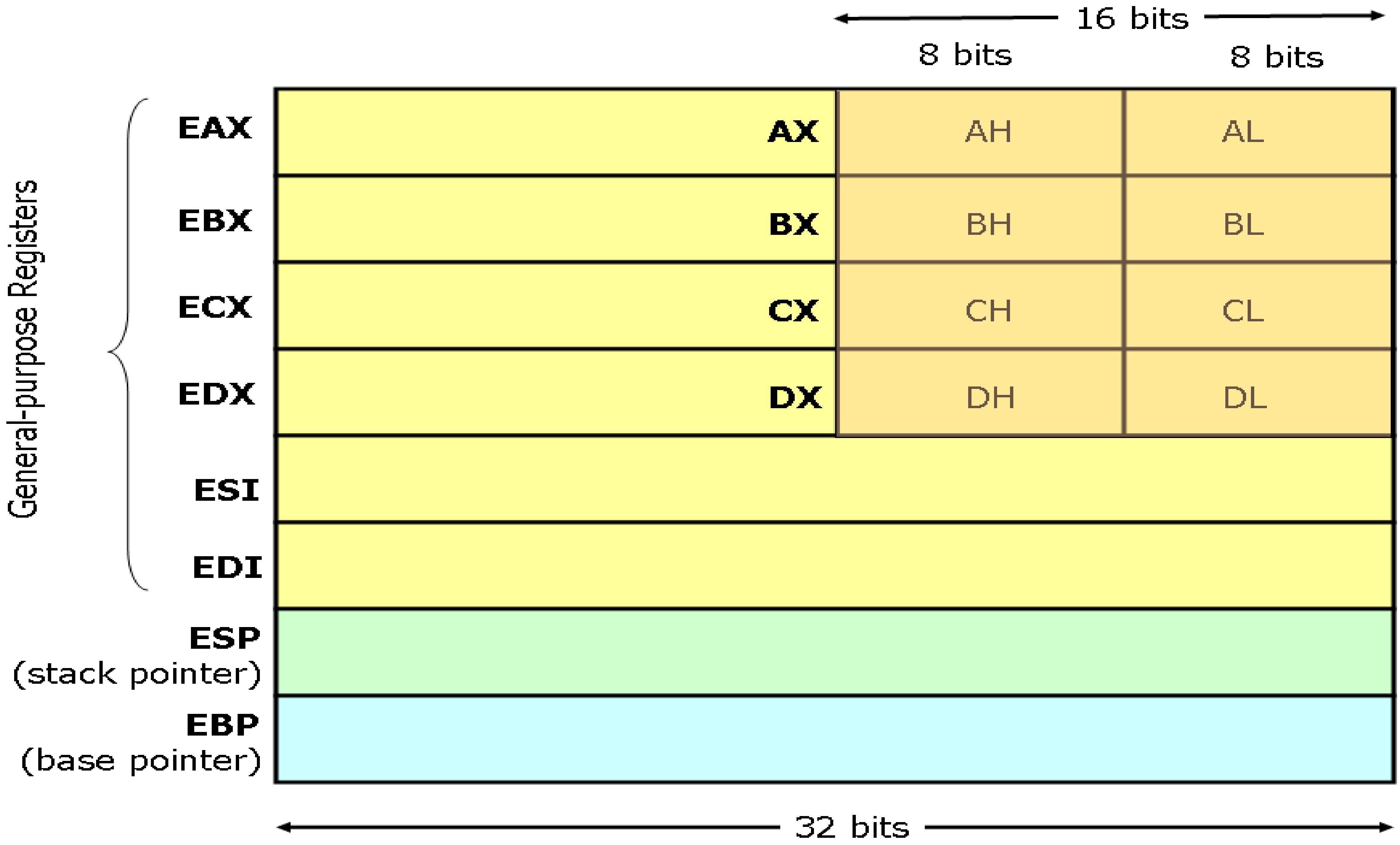
- All assembly languages are made up of instruction sets
- Instructions are generally simple arithmetic operations that take registers or constant values as arguments
- Also called Operands, OpCode, Op(s), mnemonics
- Intel syntax: operand destination, source
  - mov eax, 5
- AT&T syntax: operand source, destination
  - mov $5, eax
- We'll be relying on the Intel syntax



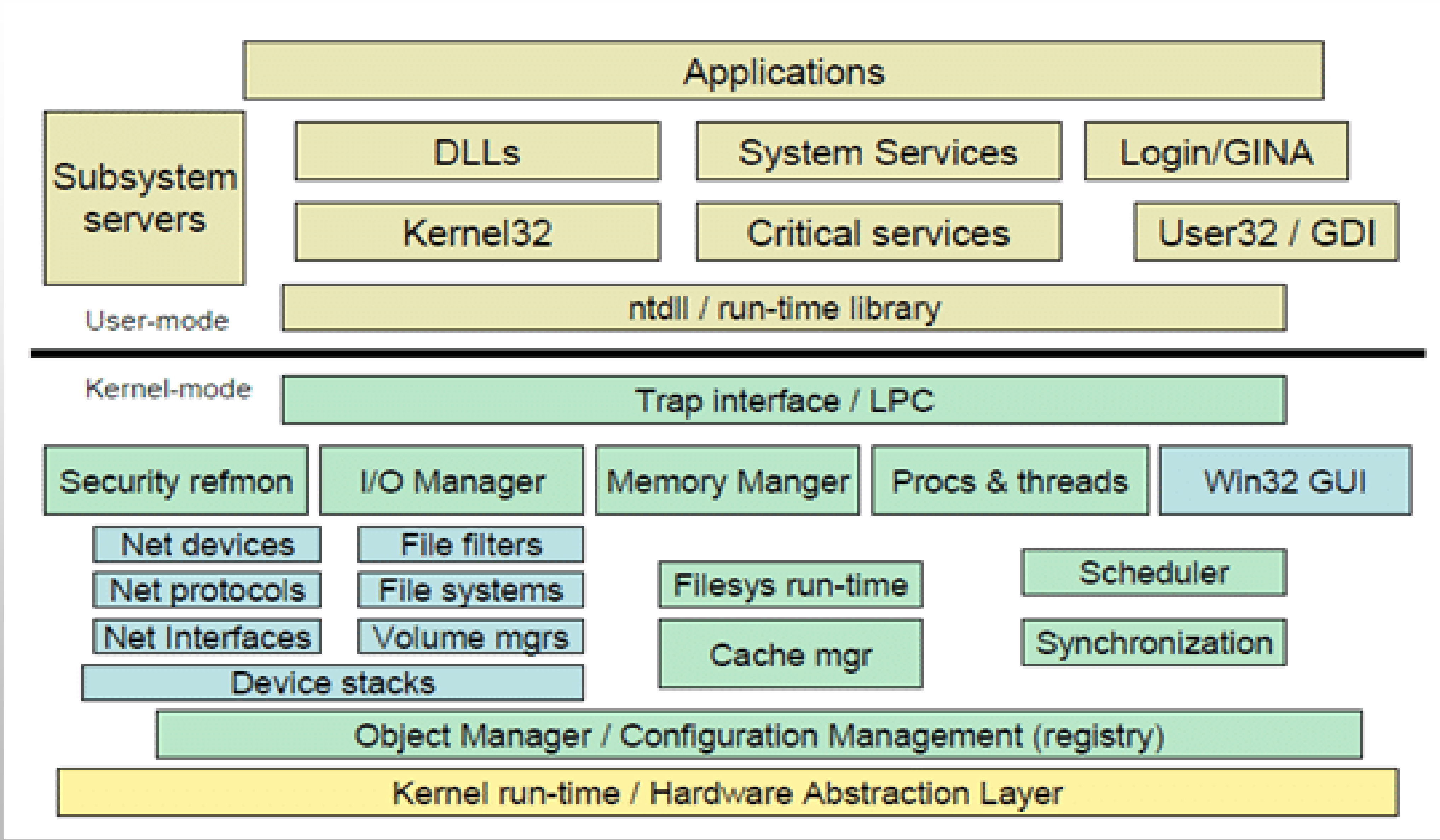

/ "root"
/bin
"essential user command binaries"
bash
cat
chmod
cp
date
echo
grep
gunzip
gzip
hostname
kill
less
ln
ls
mkdir
more
mount
mv
nano
open
ping
ps
pwd
rm
sh
su
tar
touch
umount
uname
/etc
"configuration files for the system"
crontab
cups
fonts
fstab
host.conf
hostname
hosts
hosts.allow
hosts.deny
init
init.d
issue
machine-id
mtab
mtools.conf
nanorc
networks
passwd
profile
protocols
resolv.conf
rpc
securetty
services
shells
timezone
/sbin
"essential system binaries"
fdisk
fsck
getty
halt
ifconfig
init
mkfs
mkswap
reboot
route
/usr
"read-only user application support data & binaries"
/usr/bin
"most user commands"
/usr/include
"standard include files for 'C' code"
/usr/lib
"obj, bin, lib files for coding & packages"
/usr/local
"local software"
/usr/local/bin
/usr/local/lib
/usr/local/man
/usr/local/sbin
/usr/local/share
/usr/share
"static data sharable accross all architectures"
/usr/share/man
"manual pages"
/var
"variable data files"
/var/cache
"application cache data"
/var/lib
"data modified as programmes run"
/var/lock
"lock files to track resources in use"
/var/log
"log files"
/var/opt
"variable data for installed packages"
/var/spool
"tasks waiting to be processed"
/var/spool/cron
/var/spool/cups
/var/spool/mail
/var/tmp
"temporary files saved between reboots"
/dev
"device files incl. /dev/null"
/home
"user home directories"
/lib
"libraries & kernel modules"
/mnt
"mount files for temporary filesystems"
/opt
"optional software applications"
/proc
"process & kernel information files"
/root
"home dir. for the root user"

OH! REGISTERS

- EAX EBX ECX EDX - General purpose registers
- ESP - Stack pointer, "top" of the current stack frame (lower memory)
- EBP - Base pointer, "bottom" of the current stack frame (higher memory)
- EIP - Instruction pointer, pointer to the next instruction to be executed by the CPU
- EFLAGS - stores flag bits
- ZF - zero flag, set when result of an operation equals zero
- CF - carry flag, set when the result of an operation is too large/small
- SF - sign flag, set when the result of an operation is negative



16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits


Applications
Subsystem servers
DLLs
System Services
Login/GINA
Kernel32
Critical services
User32 / GDI
ntdll / run-time library
User-mode
Kernel-mode
Trap interface / LPC
Security refmon
I/O Manager
Memory Manger
Procs & threads
Win32 GUI
Net devices
Net protocols
Net Interfaces
File filters
File systems
Volume mgrs
Device stacks
Filesys run-time
Cache mgr
Scheduler
Synchronization
Object Manager / Configuration Management (registry)
Kernel run-time / Hardware Abstraction Layer

FUZZING AND CRASH ANALYSIS

Discovering faults in applications by providing unexpected input and monitoring for exceptions.

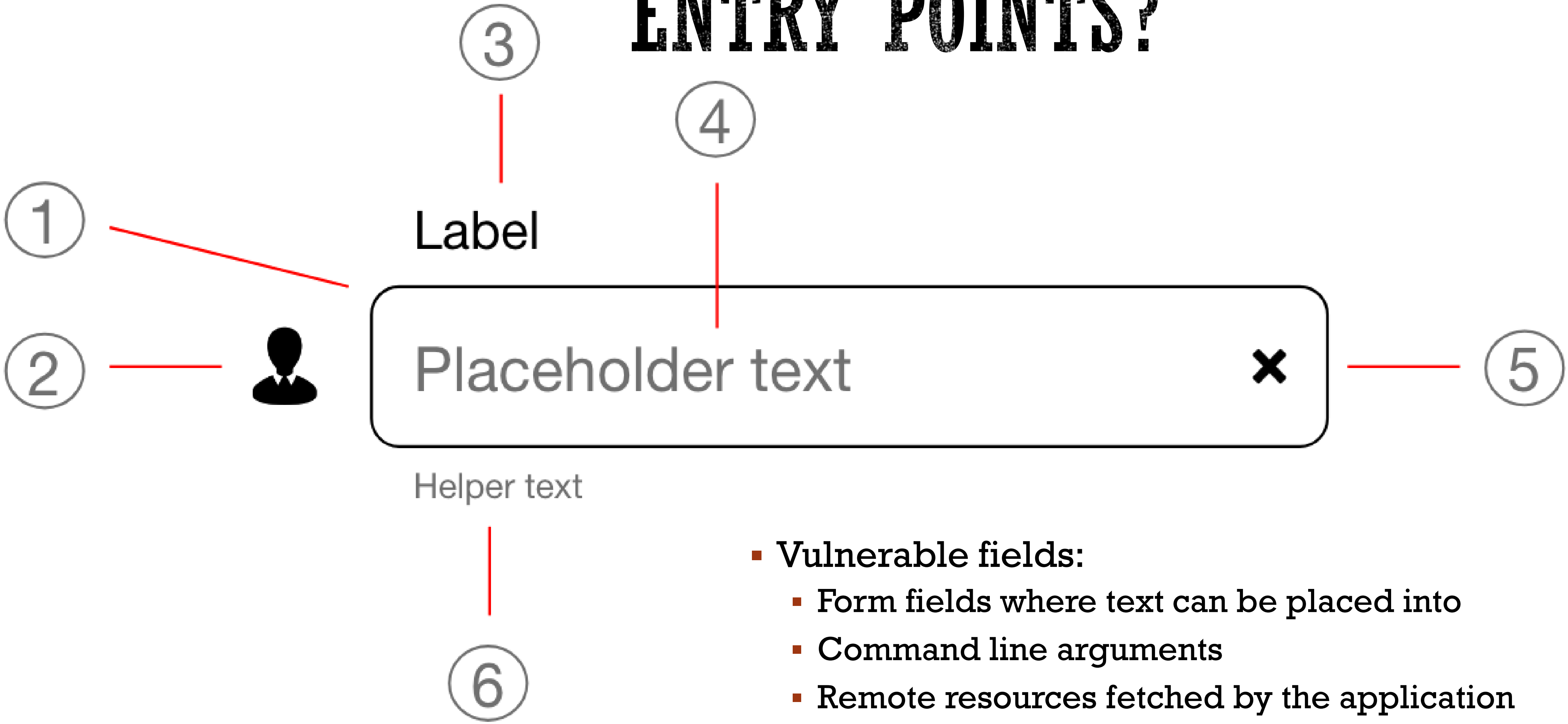
Types of fuzzers:

- Mutation-based
- Generation-based

Fuzzing Targets:

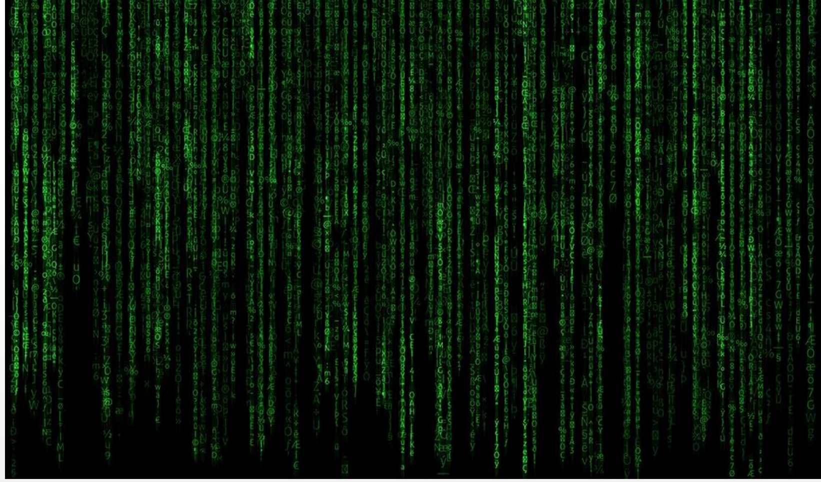
- Environment variables and Arguments
- Web application and server
- File Format Network Protocol
- Web browsers
- In-memory

ENTRY POINTS?

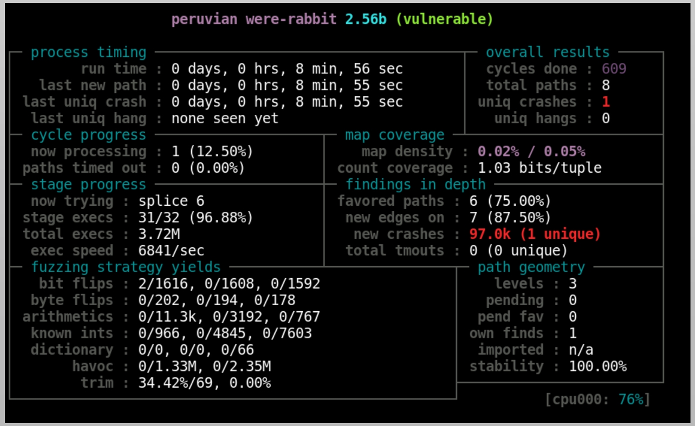


- Vulnerable fields:
  - Form fields where text can be placed into
  - Command line arguments
  - Remote resources fetched by the application
  - Files parsed by an application

FUZZING AND CRASH ANALYSIS



- Diverse Input Generation
- Coverage Analysis
- Mutation and Generation
- Boundary Testing & Ethical Considerations
- Continuous Process
- Customization >>> False Positives
- Tool Selection



FINDING OFFSET AND OVERWRITING EIP

- Supply an input of a certain length to the Binary.
- Make the EIP register to point to a certain address.
- EIP control – reuse the dead code
- In Windows, application get access violation in the debugger.
- For Linux, you got "gdb".
- Creating pattern and finding offset from that pattern is most useful.
- What are the bad characters?   Well….
- We got all the things, what now? Now, it's shellcode time!

BEFORE WE MOVE TO FORWARD

Linux

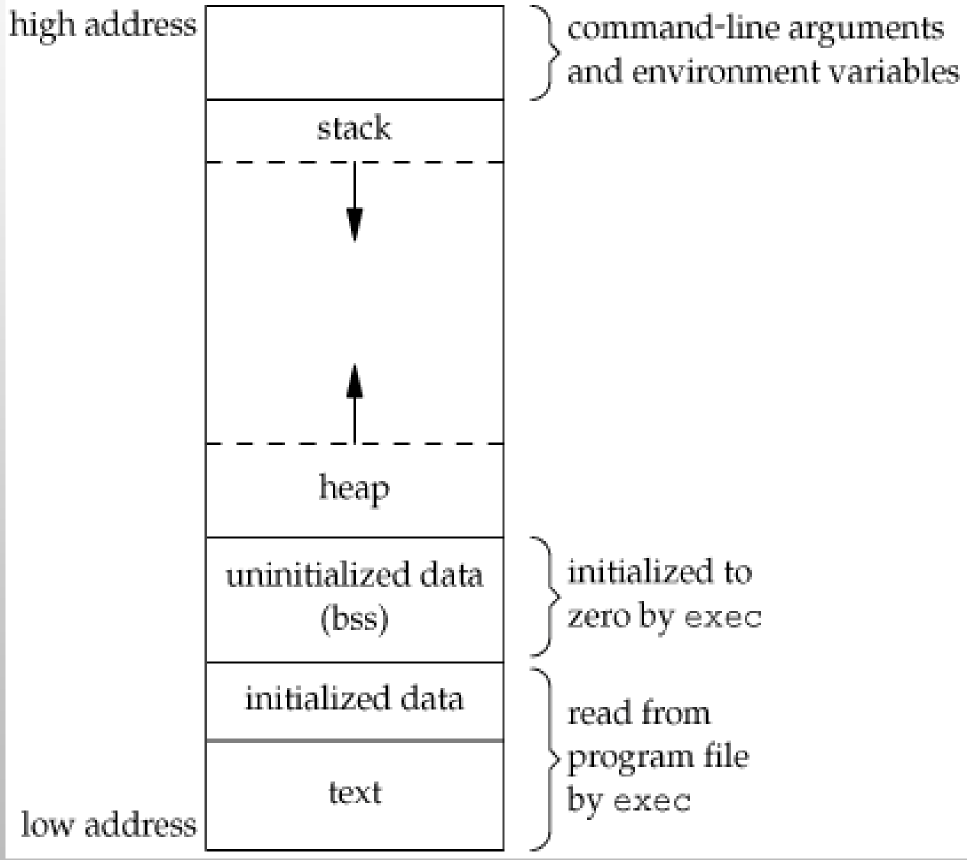
Compile: gcc -fno-stack-protector -z execstack program.c -o program

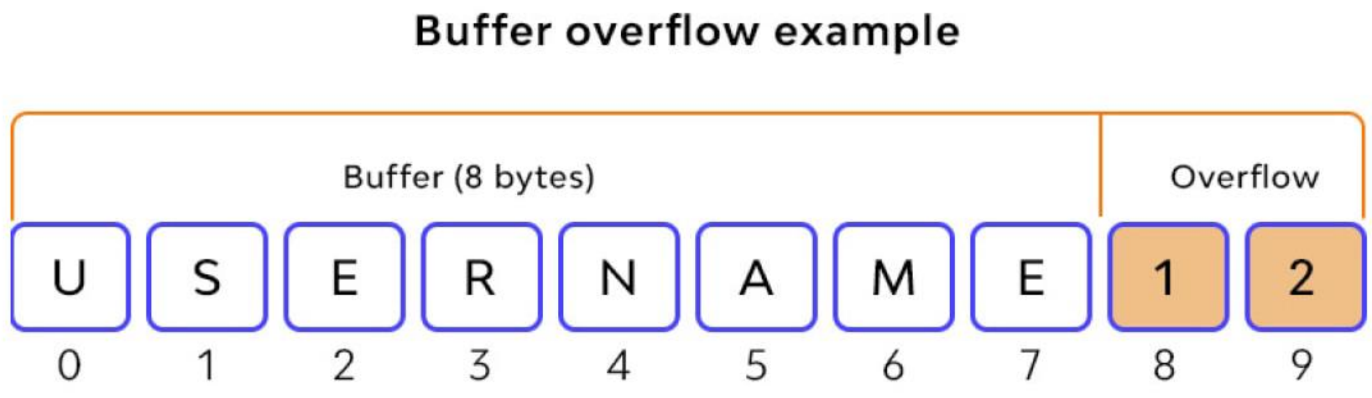
Disable ASLR: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

STACK-BASED BUFFER OVERFLOW

- Stack overflow, also called buffer overflow or stack-based buffer overflow
- It occurs due to a programmatic error.
- This may happen when the program is Insecurely handling user-supplied data.
- The core of buffer overflow exploitation on Windows is the same as it is on Linux.

STACK-BASED BUFFER OVERFLOW



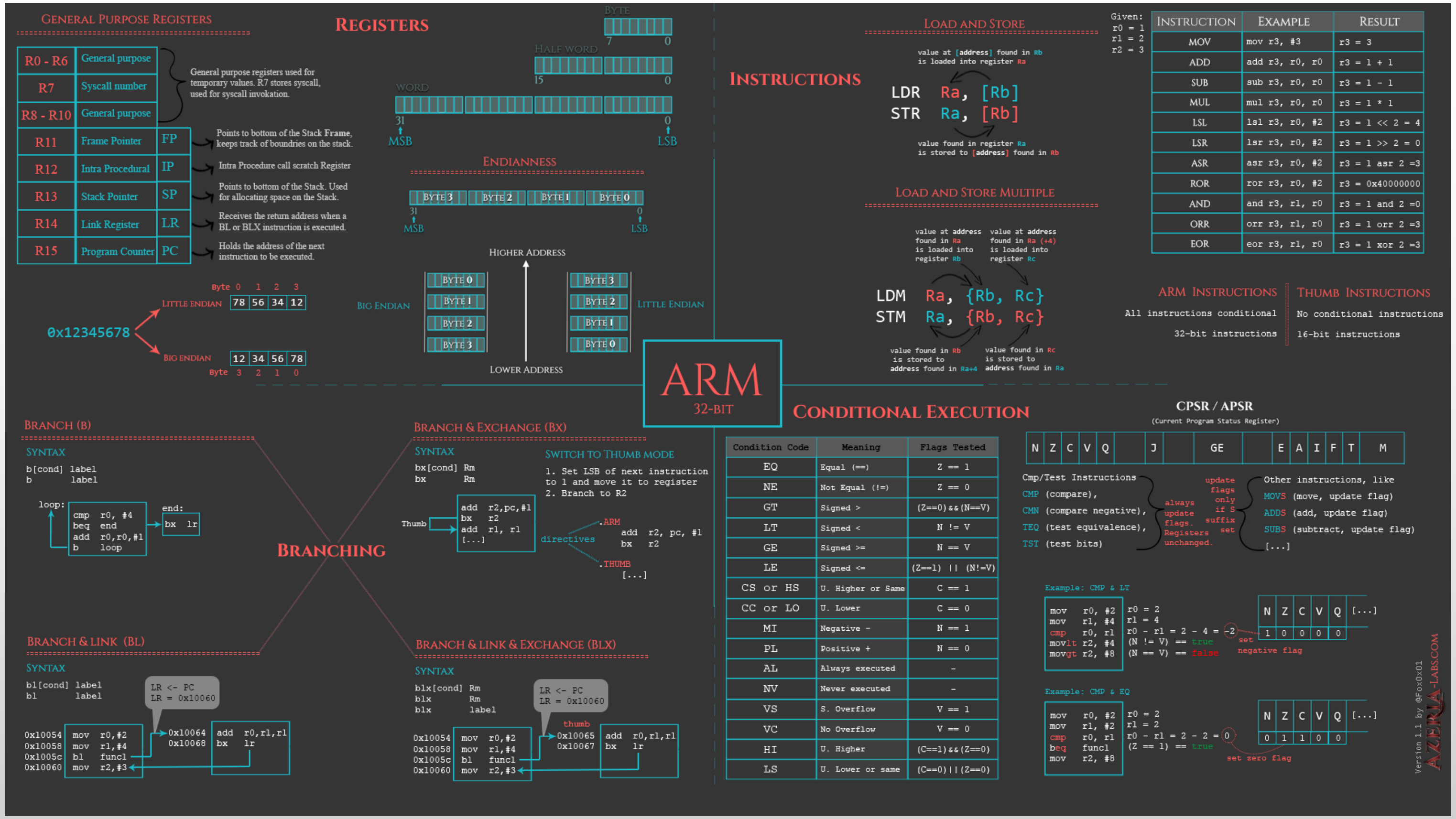


AGAIN !!! BEFORE WE MOVE TO HANDS-ON

Linux

Compile: gcc -fno-stack-protector -z execstack program.c -o program

Disable ASLR: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space



```
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

AN ADVENTURE TOO BIG FOR THE REAL WORLD
A STEVEN SPIELBERG FILM
READY PLAYER ONE
SPACE INVADERS
MUSIC BY ALAN
EXECUTIVE PRODUCERS DANIEL LUPI CHRIS DeFARIA AND BRUCE BERMAN
BASED ON THE NOVEL BY ERNEST CLINE
SCREENPLAY BY ZAK PENN
DONALD De LINE, p.g.a. MACOSKO KRIEGER, p.g.a. STEVEN SPIELBERG, p.g.a. DAN FARAH, p.g.a.
DIRECTED BY STEVEN SPIELBERG

INTRODUCTION TO EGG HUNTING

- Egg Hunter shellcode simply means small sized shellcode
- Writing shellcode to Exploit within a Limited space
- Shellcode won't fit in the available space
- Storing User input in the memory for long run than expected.
- Relays on system calls that have ability to traverse process memory

INTRODUCTION TO EGG HUNTING

- Character transformation may occur
- These are not only limited to characters
- Sometime whole memory chunk
- Reason?? It’s Unknown
- Sometimes the buffer is truncated -- hard to fit shellcode
- Then how to archive exploitation?

INTRODUCTION TO EGG HUNTING

- Egg Hunter can be generated in Immunity Debugger with the help of Mona.py
  - !mona egg –t r00t3r
  - Simple Format of an Egg Hunter shell is:
  - EGGEGG + shellcode
- Here EGGEGG is nothing, but tag or word repeated twice

So, Step goes like this:

1. Write a shellcode in the limited buffer to find EGGEGG
2. Once the shellcode is executed, then it'll look for both occurrence of EGG
3. Once EGGEGG is found it'll execute our desired exploit which is present after EGGEGG!

```
         Immunity Debugger 1.85.0.0 : R'lyeh
         Need support? visit http://forum.immunityinc.com/
         "C:\Documents and Settings\Administrator\Desktop\VulnServer\vulnserver.exe"

         Console file 'C:\Documents and Settings\Administrator\Desktop\VulnServer\vulnserver.exe'
         [22:10:22] New process with ID 0000069C created
00401130 Main thread with ID 00000698 created
00400000 Modules C:\Documents and Settings\Administrator\Desktop\VulnServer\vulnserver.exe
62500000 Modules C:\Documents and Settings\Administrator\Desktop\VulnServer\essfunc.dll
71AA0000 Modules C:\WINDOWS\system32\WS2HELP.dll
71AB0000 Modules C:\WINDOWS\system32\WS2_32.DLL
77C10000 Modules C:\WINDOWS\system32\msvcrt.dll
77DD0000 Modules C:\WINDOWS\system32\ADVAPI32.dll
77E70000 Modules C:\WINDOWS\system32\RPCRT4.dll
77FE0000 Modules C:\WINDOWS\system32\Secur32.dll
7C800000 Modules C:\WINDOWS\system32\kernel32.dll
7C900000 Modules C:\WINDOWS\system32\ntdll.dll
00401130 [22:10:22] Program entry point
71A50000 Modules C:\WINDOWS\system32\mswsock.dll
662B0000 Modules C:\WINDOWS\system32\hnetcfg.dll
77F10000 Modules C:\WINDOWS\system32\GDI32.dll
7E410000 Modules C:\WINDOWS\system32\USER32.dll
0BADF00D [+] Command used:
0BADF00D !mona egg -t sh377c0d3
0BADF00D [+] Egg set to w00t
0BADF00D [+] Generating traditional 32bit egghunter code
0BADF00D [+] Preparing output file 'egghunter.txt'
0BADF00D     - (Re)setting logfile egghunter.txt
0BADF00D [+] Egghunter  (32 bytes):
         "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
         "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.016000
```

!mona egg -t sh377c0d3

start | Immunity Debugger - ... | C:\Documents and Se...

BEFORE WE GET INTO ROP

DEFENCE :

ASLR, NX, DEP, STACK CANARY ... and more

https://github.com/sashs/Ropper

https://github.com/JonathanSalwan/ROPgadget

https://github.com/corelan/mona

RETURN-ORIENTED PROGRAMMING (ROP)

- There is only ASLR, you could brute force the shellcode address.

- Only NX, you could return to libc , as it is always at the same address.

- What if … there is ASLR + NX?
  - Can't brute force now !
  - Can't return to the system, as it will always be at a different address.
  - Now How to Achieve Exploitation ?!

RETURN-ORIENTED PROGRAMMING (ROP)

- Return-Oriented Programming is successor of return-to-libc attack technique.

- In return-oriented programming, you can chain multiple functions to form a ROP chain.

- Gadgets? These are nothing but sequence of code residing in executable memory followed by return instruction.

RETURN-ORIENTED PROGRAMMING (ROP)

- Abuse code that is:
  - Already within the process address space
  - Not randomized (remember that ASLR randomizes certain sections, not everything)
- There could be another function instead of gadget().
- The only thing that should be done is that the stack should be prepared for another function.

RETURN-ORIENTED PROGRAMMING (ROP)

Gadgets and Returns

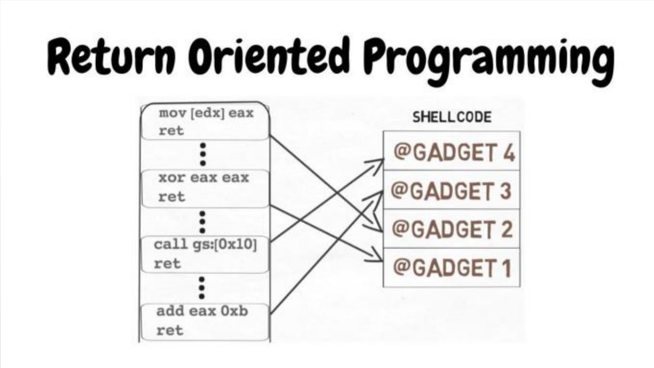
- 1122aa33 holds the real, intended instruction
- Let's offset it 1 byte and now it points to 1122aa34
- Just 1 byte off and completely different instructions followed by a return!
- This is how gadgets are built !!!

- **mov EIP, [ESP]**
- **add esp, 4** //those two are the standard RET implementation

- **add esp, 4** //this is the 4 (of RET 4) – align the stack by 4 bytes. If it's ret 8, then the following will be added: **esp, 8**

RETURN-ORIENTED PROGRAMMING (ROP)

- Explore and achieve Overflow vulnerability
- Overwrite return address program with a ROP gadget
- ROP gadget pop a value from stack and store it in register
- Now find out another ROP gadget for a specific function
- Chain gadgets to pop value into a REGISTER
- Execute second gadget to perform another specific operation
- BOOM !! ROP chain is executed calling vulnerable function!



[NX bypass with mprotect()]()

THANK YOU