Offensive Software Exploitation

SEC-300-01/CSI-301-02

Ali Hadi

@binaryz0ne

Exploit Mitigation

Preventing memory corruption techniques!!!

Slides are modified from Memory Corruption 101, NYU Poly, by Dino Dai Zovi

Exploit Mitigation – Part #2

Last session: SafeSEH, SEHOP, and Stack Guards (Canaries)…

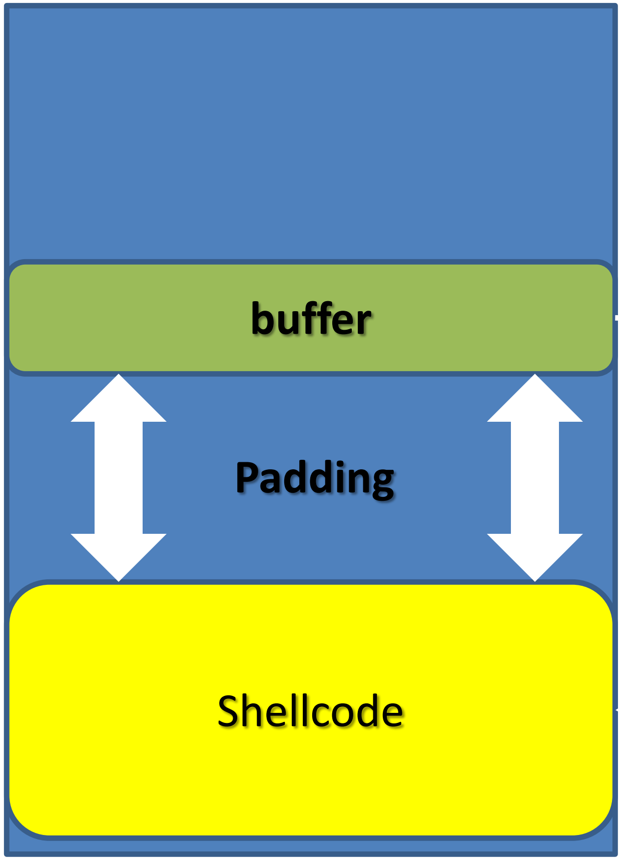
Data Execution Prevention (DEP) / No eXecute (NX)

W ^ X

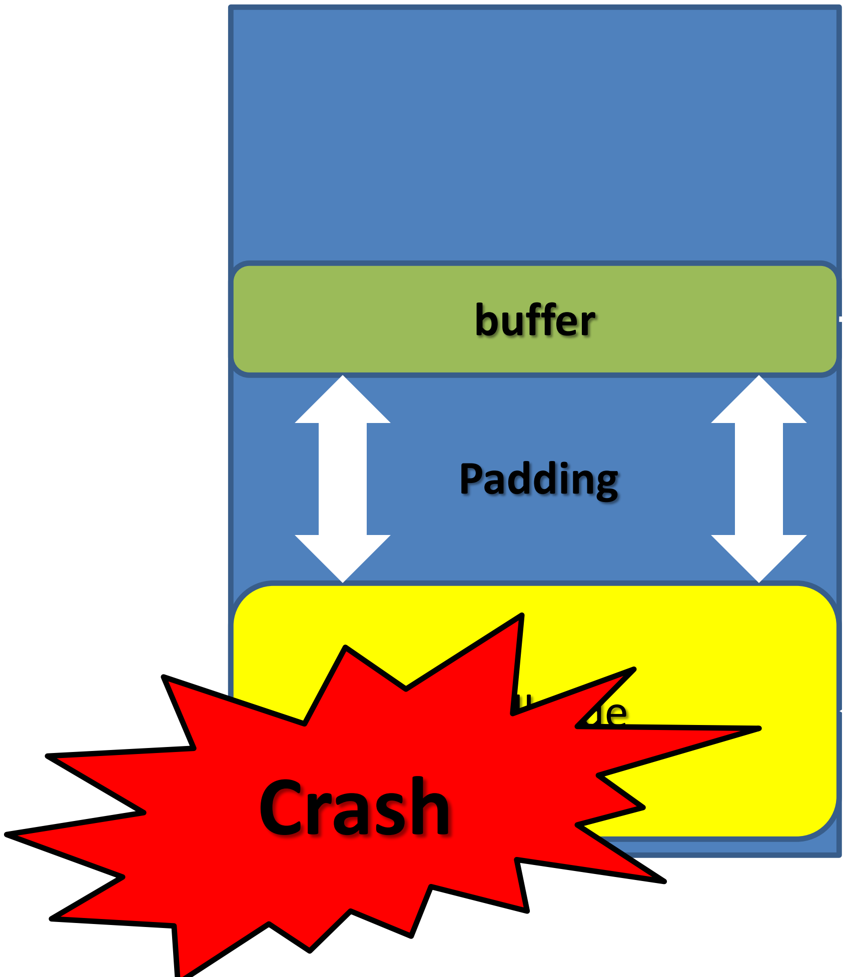
Defeating Exploits using DEP

- No-eXecute CPU technology
  - Intel → eXecute Disable (XD bit)
  - AMD → Enhanced Virus Protection
  - ARM → eXecute Never (XN)
- Has four modes: OptIn, OptOut, AlwaysOn, AlwaysOff
  - Permanent DEP uses SetProcessDEPPolicy for all programs compiled with /NXCOMPAT option
- Use *bcdedit.exe* to check your Windows DEP status

Defeating Exploits – Past.



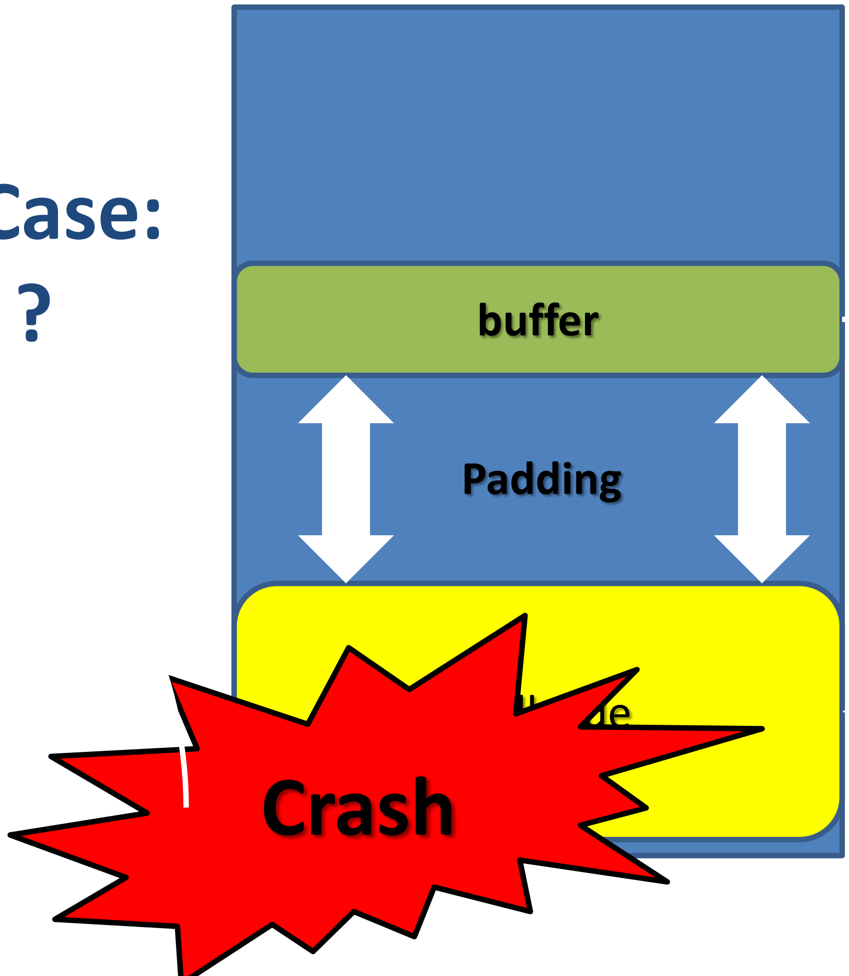
Data Execution Prevention (DEP)



exclusively either writable ***or*** executable

Data Execution Prevention – Cont.

Worst Case:
DoS ?



Data Execution Prevention – Cont.

Cited [1]

Software DEP

- Makes sure that SEH exception handlers point to non-writable memory (weak)

Hardware DEP

- Enforces that processor does not execute instructions from data memory pages (stack, heap)
- Make page permission bits meaningful
  - R != X
- Fallback to software if hardware DEP isn't supported
  - Not too good!

Bypassing DEP

Cited [1]

- Return-to-libc / code reuse
  - Return into the beginning of a library function
  - Function arguments come from attacker-controlled stack
  - Can be chained to call multiple functions in a row

- On XP SP2 and Windows 2003, attacker could return to a particular place in NTDLL and disable DEP for the entire process

Return-to-libc (ret2libc)

Cited [1]

- An attack against non-executable memory segments (DEP, W^X, etc)
- Instead of overwriting return address to return into shellcode, return into a loaded library to simulate a function call
- Data from attacker's controlled buffer on stack are used as the function's arguments
  - i.e. call system (bash or cmd)

Getting around non-executable stack (and fix)", Solar Designer (BUGTRAQ, August 1997)

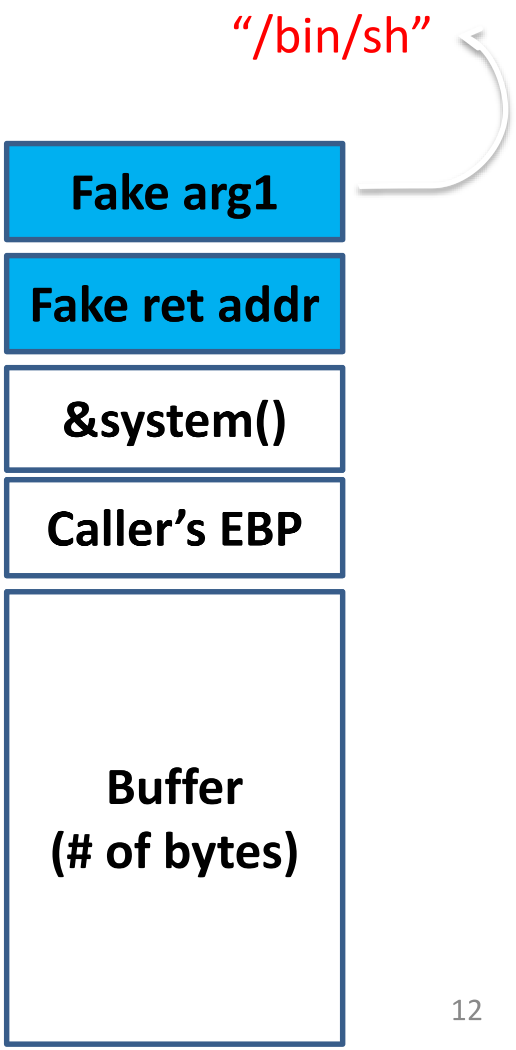
Return-to-libc (ret2libc) – Cont.

Cited [1]

Overwrite return address by address of a libc function

- setup fake return address and argument(s)
- ret will “call” libc function

No injected code!



Return Chaining

Cited [1]

- Stack unwinds upward
- Can be used to call multiple functions in succession
- First function must return into code to advance stack pointer over function arguments
  - i.e. pop-pop-ret
  - Assuming cdecl and 2 arguments

Argument 2
Argument 1
&(pop-pop-ret)
Function 2
Argument 2
Argument 1
&(pop-pop-ret)
Function 1

A: Address

S: Space

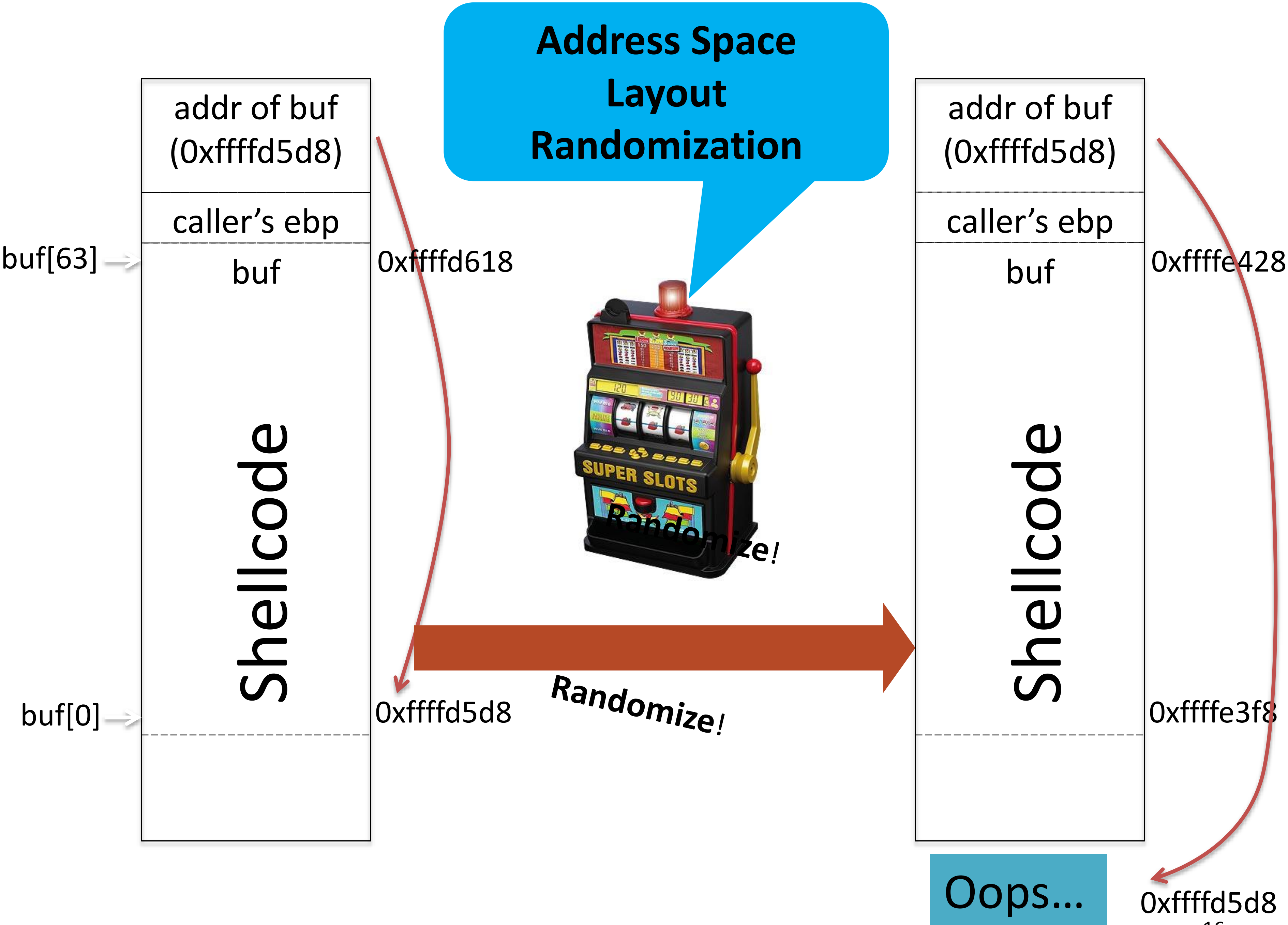
L: Layout

R: Randomization

ASLR

Cited [1]

- Almost all exploits require hard-coding memory addresses
- If those addresses are impossible to predict, those exploits would not be possible
- ASLR moves around code (executable and libraries), data (stacks, heaps, and other memory regions)
- Windows Vista randomizes DLLs at boot-time, everything else at run-time


Address Space Layout Randomization
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
buf[63]
buf[0]
0xffffd618
0xffffd5d8
Randomize!
SUPER SLOTS
Randomize!
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
0xffffe428
0xffffe3f8
Oops...
0xffffd5d8

ASLR

Traditional exploits need precise addresses

- *stack-based overflows:* location of shell code
- *return2libc:* library addresses

- Problem: program's memory layout is fixed
  - stack, heap, libraries etc.

- Solution: randomize addresses of each region!

Memory

Base address a | Base address b | Base address c

Program

- Code
- Uninitialized data
- Initialized data

Mapped

- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack

- Main stack

ASLR Randomization

a + 16 bit rand r_1

b + 16 bit rand r_2

c + 24 bit rand r_3

Program

- Code
- Uninitialized data
- Initialized data

Mapped

- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack

- Main stack

Bypassing ASLR

Cited [1]

- Poor entropy
  - Sometimes the randomization isn't random enough or the attacker may try as many times as needed
- Memory address disclosure
  - Some vulnerabilities or other tricks can be used to reveal memory addresses in the target process
- Using non-ASLR enabled module

- One address may be enough to build your exploit !!!

return-oriented programming

Return-Oriented Programming

Cited [1]

- Instead of returning to functions, return to instruction sequences followed by a return instruction
- Can return into middle of existing instructions to simulate different instructions
- All we need are useable byte sequences anywhere in executable memory
  - Forge shell code out of existing application logic **gadgets**

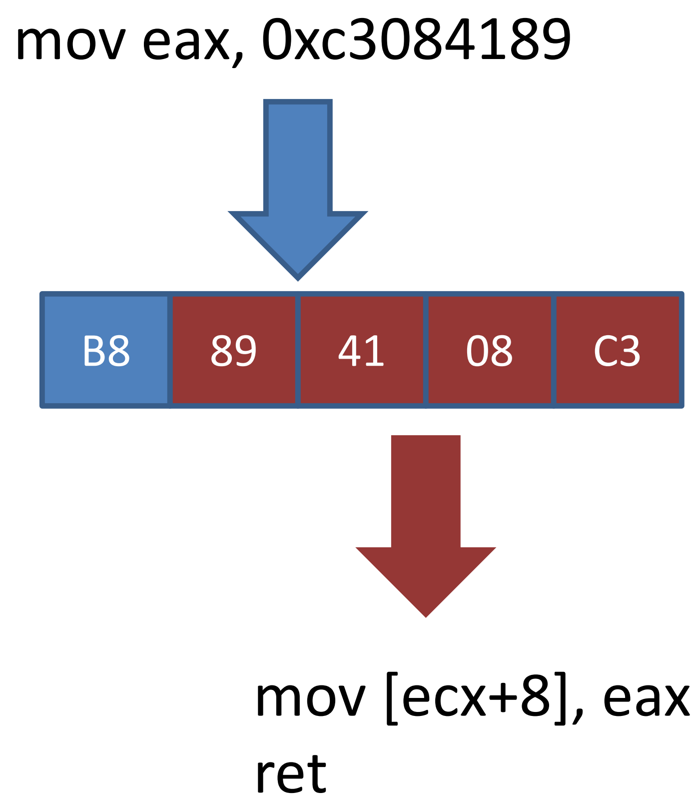


Image by Dino Dai Zovi

Return-Oriented Programming

Cited [1]

- Return into useful instruction sequences followed by return instructions
- Chain useful sequences together to form useful operations ("**gadgets**")

Requirements:

- vulnerability + gadgets + some un-randomized code (addresses of gadgets must be known)

ROP Programming

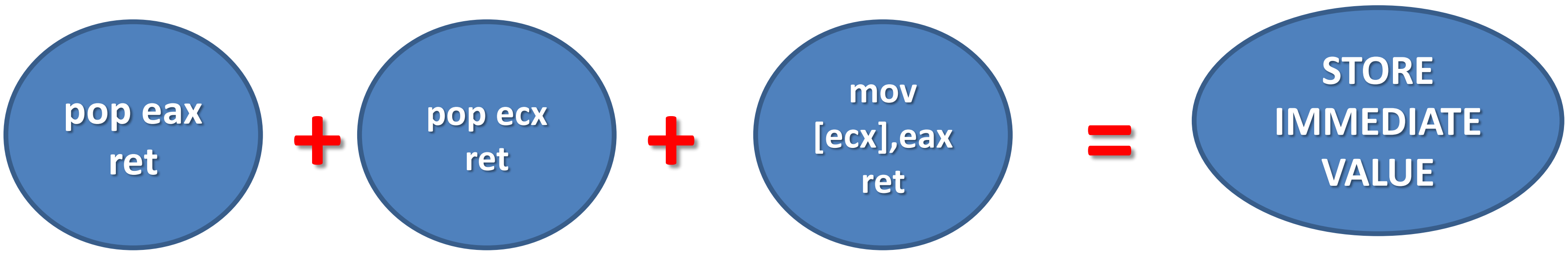
Cited [1]

1. Disassemble code
2. Identify ***useful*** code sequences as gadgets
3. Assemble gadgets into desired shellcode

Return-Oriented Gadgets

Cited [1]

- Various instruction sequences can be combined to form gadgets



After all that…

- Bypassing DEP & ASLR makes you Mohammad Ali of Software Exploitation ☺

Summary

- Explained exploit mitigation techniques (Compiler/System)
- Explained different mitigation techniques such as DEP and ASLR
- What is Ret2libc
- What is Return-Oriented Programming and how to benefit from it for software exploitation

References

- Memory Corruption 101, NYU Poly, Dino Dai Zovi
- DEP Evasion Techniques, http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html
- SEHOP, http://www.sysdream.com/articles/sehop_en.pdf
- Shellcode Storm, http://shell-storm.org/shellcode/
- Stack /GS, https://msdn.microsoft.com/en-us/library/8dbf701c%28VS.80%29.aspx?f=255&MSPPError=-2147217396