Shellcoding

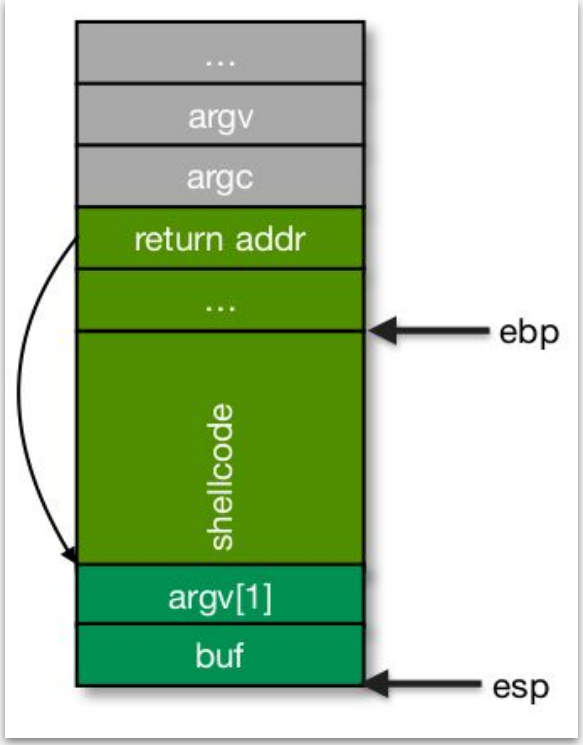
Mustakimur R. Khandaker

Shellcode

The attacker's goal is to **execute arbitrary code.**

- Hijacking control flow is the first step.
- Traditionally, the next step is to spawn a shell.
  - Write the shellcode to the buffer.
  - Hijack *EIP* to the shellcode.
  - Using *exec("/bin/sh")* syscall.

```
...
0x080483fa <+22>:  call   0x8048300 <strcpy@plt>
0x080483ff <+27>:  leave
0x08048400 <+28>:  ret
```



More on Shellcode

Executable content (often called shellcode or exploits).

- Use a system call *(execve)* to spawn a shell.
- Usually, a shell should be started for remote exploits - input/output redirection via socket.

Shellcode can do practically anything.

- Create a new user.
- Change user password.
- Bind a shell to a socket port (remote shell).
- Open a connection to the attacker machine (backdoor).

Executing a System Call

1. Store syscall number in *eax*.
2. Save arg 1 in *ebx*, arg 2 in *ecx*, arg 3 in *edx*.
3. Execute int 0x80 (or *sysenter*).
4. Syscall runs and returns the result in *eax*.

execve (“/bin/sh”, 0, 0)

eax: 0x0b
ebx: addr of “/bin/sh”
ecx: 0

Shellcode Example

```
xor  ecx, ecx ; clear ecx
mul  ecx      ; clear eax
push ecx      ; push “/bin/sh” to stack
push 0x68732f2f
push 0x6e69622f
mov  ebx, esp ; set ebx
mov  al, 0xb  ; set eax
int  0x80
```

Shellcode Assembly

Notice no NULL chars. Why?

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Executable Shellcode

Test the Shellcode

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

char code[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
              "\x73\x68\x68\x2f\x62\x69\x6e\x89"
              "\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
    printf ("Shellcode length : %d bytes\n", strlen (code));
    int(*f)()=(int(*)())code;
    f();
}
```

```
$ gcc -o shellcode -fno-stack-protector
    -z execstack shellcode.c
```

Shellcode Execution

Error: register edx also set to 0

```
xor  ecx, ecx
mul  ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov  ebx, esp
mov  al, 0xb
int  0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers

0x0	0x0
0x68	h
0x73	s
0x2f	/
0x2f	/
0x6e	n
0x69	i
0x62	b
0x2f	/

esp →

Stack

Shellcodes Database for Study Cases

http://shell-storm.org/shellcode/

Shellcodes are written in machine code (assembly language), so they are architecture dependent.

Database of shellcode is available on the web. Some of them even has search option integrated in debugger.

- E.g. GEF debugger has command shellcode to search shellcode from shell-storm database.

```
gef➤ shellcode search arm
[+] Showing matching shellcodes
901     Linux/ARM       Add map in /etc/hosts file - 79 bytes
853     Linux/ARM       chmod("/etc/passwd", 0777) - 39 bytes
854     Linux/ARM       creat("/root/pwned", 0777) - 39 bytes
855     Linux/ARM       execve("/bin/sh", [], [0 vars]) - 35 bytes
729     Linux/ARM       Bind Connect UDP Port 68
730     Linux/ARM       Bindshell port 0x1337
[...]
gef➤ shellcode get 698
[+] Downloading shellcode id=698
[+] Shellcode written as '/tmp/sc-EfcWtM.txt'
```

Intel x86-64

- Linux/x86-64 - Dynamic null-free reverse TCP shell - 65 bytes *by Philippe Dugre*
- Linux/x86-64 - execveat("/bin//sh") - 29 bytes *by ZadYree, vaelio and DaShrooms*
- Linux/x86-64 - Add map in /etc/hosts file - 110 bytes *by Osanda Malith Jayathissa*
- Linux/x86-64 - Connect Back Shellcode - 139 bytes *by MadMouse*
- Linux/x86-64 - access() Egghunter - 49 bytes *by Doreth.Z10*
- Linux/x86-64 - Shutdown - 64 bytes *by Keyman*
- Linux/x86-64 - Read password - 105 bytes *by Keyman*

ARM

- Linux/ARM - execve("/bin/sh", NULL, 0) - 34 bytes *by Jonathan 'dummys' Borgeaud*
- Linux/ARM - Add map in /etc/hosts file - 79 bytes *by Osanda Malith Jayathissa*
- Linux/ARM - chmod("/etc/passwd", 0777) - 39 bytes *by gunslinger_*
- Linux/ARM - creat("/root/pwned", 0777) - 39 bytes *by gunslinger_*
- Linux/ARM - execve("/bin/sh", [], [0 vars]) - 35 bytes *by gunslinger_*
- Linux/ARM - Bind Connect UDP Port 68 *by Daniel Godas-Lopez*
- Linux/ARM - Bindshell port 0x1337 *by Daniel Godas-Lopez*

MIPS

- Linux/mips - Reverse Shell Shellcode - 200 bytes *by Jacob Holcomb*
- Linux/mips - execve(/bin/sh) - 56 bytes *by core*
- Linux/mips - execve(/bin/sh, */bin/sh, 0) - 52 bytes *by entropy*
- Linux/mips - add user(UID 0) with password - 164 bytes *by rigan*
- Linux/mips - connect back shellcode (port 0x7a69) - 168 bytes *by rigan*
- Linux/mips - execve /bin/sh - 48 bytes *by rigan*

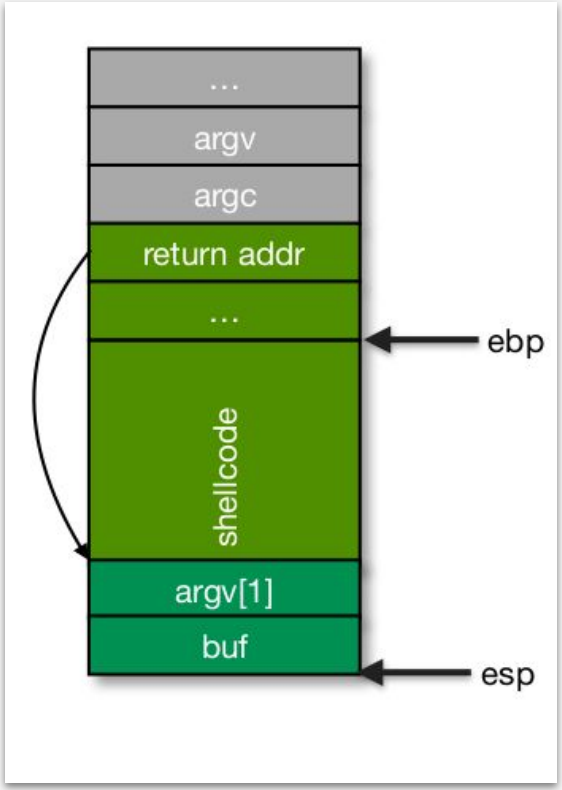
Shellcode Challenges

Need to know the address of memory-based parameters.

- E.g., “/bin/sh” of the execve syscall.
- Solutions:
  1. Push it to the stack and get addr from esp (as in the previous example).
  2. Use position-independent code.

Need to set return address to the shellcode.

- Use nop sled to allow for slackness.

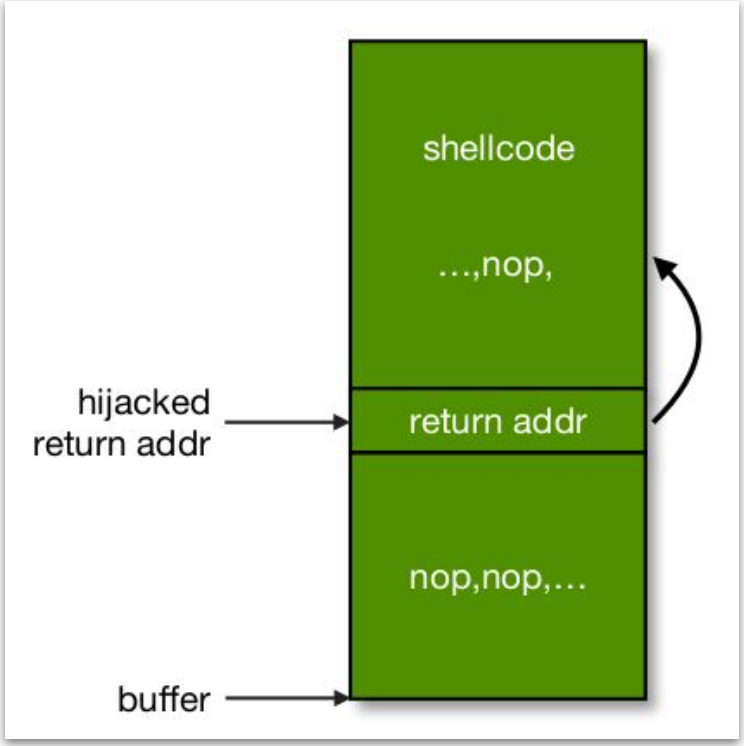


NOP Sled

Guess approximate stack state when target function is called.

Insert many NOPs before the shellcode.

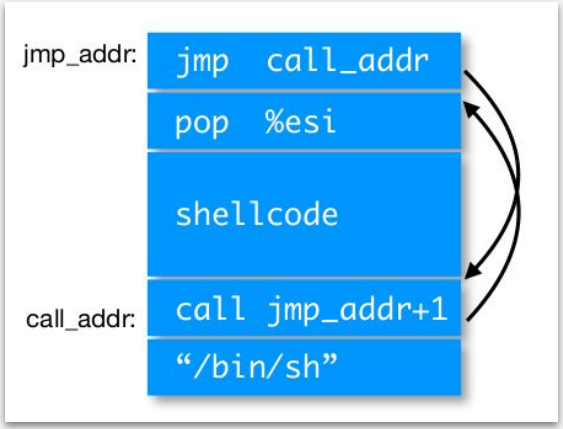
- Allow for some slackness of the return address.
- Many NOP instructions: *nop, xor eax, eax, inc ax*.



Position-independent Shellcode

Similar to position-independent code for shared libraries.

- Jmp at beginning of shell code to a call instruction.
- Call instruction sits right before the */bin/sh* string.
- Call goes back to *jmp+1* after pushing "return address" to the stack.
- Use relative jmp/call instructions.



PIS Example

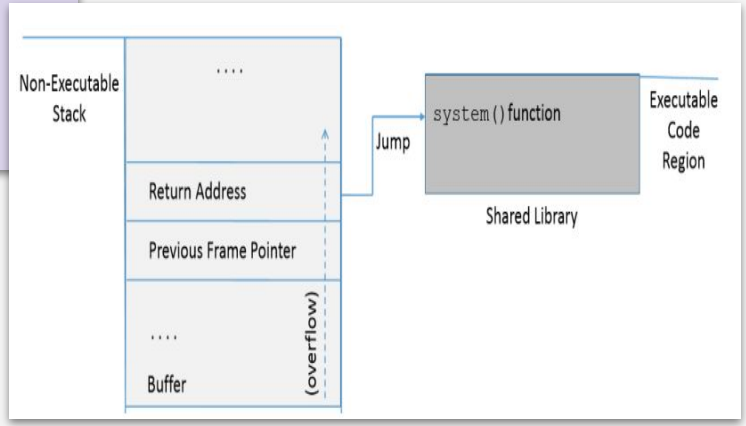
```
;setup
jmp     0x26                    ; 2 bytes
popl    %esi                    ; 1 byte
movl    %esi,0x8(%esi)          ; 3 bytes
movb    $0x0,0x7(%esi)          ; 4 bytes
movl    $0x0,0xc(%esi)          ; 7 bytes

;execve syscall
movl    $0xb,%eax               ; 5 bytes
movl    %esi,%ebx               ; 2 bytes
leal    0x8(%esi),%ecx          ; 3 bytes
leal    0xc(%esi),%edx          ; 3 bytes
int     $0x80                   ; 2 bytes

;exit syscall
movl    $0x1, %eax              ; 5 bytes
movl    $0x0, %ebx              ; 5 bytes
int     $0x80                   ; 2 bytes

;setup
call    -0x2b                   ; 5 bytes
.string \"/bin/sh\"             ; 8 bytes
```

Code Reuse Attacks

Code Reuse Attacks

They leverage existing code. No injected code is needed.
Code reuse attacks can bypass W^X and code signing.
Two types of code reuse attacks:

- Return-to-libc attacks reuse library (libc) functions.
- Return-oriented programming reuse gadgets of the victim.
  - Gadgets typically end in return instructions, chained together, they allow an attacker to perform arbitrary operations.



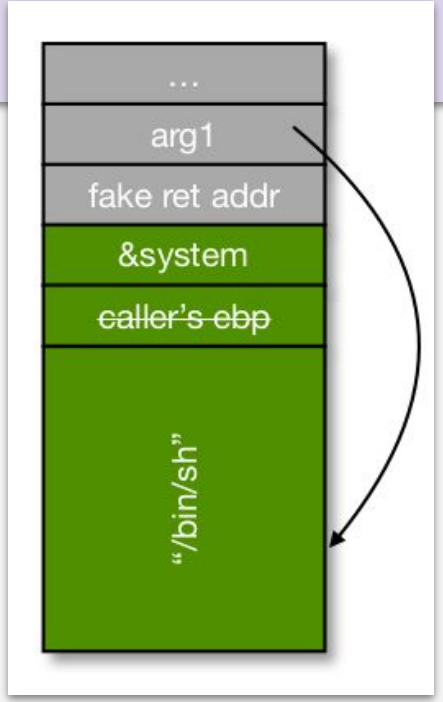
Return-to-libc Attacks

Overwrite control data (e.g., return address) by address of a library (e.g., libc) function.

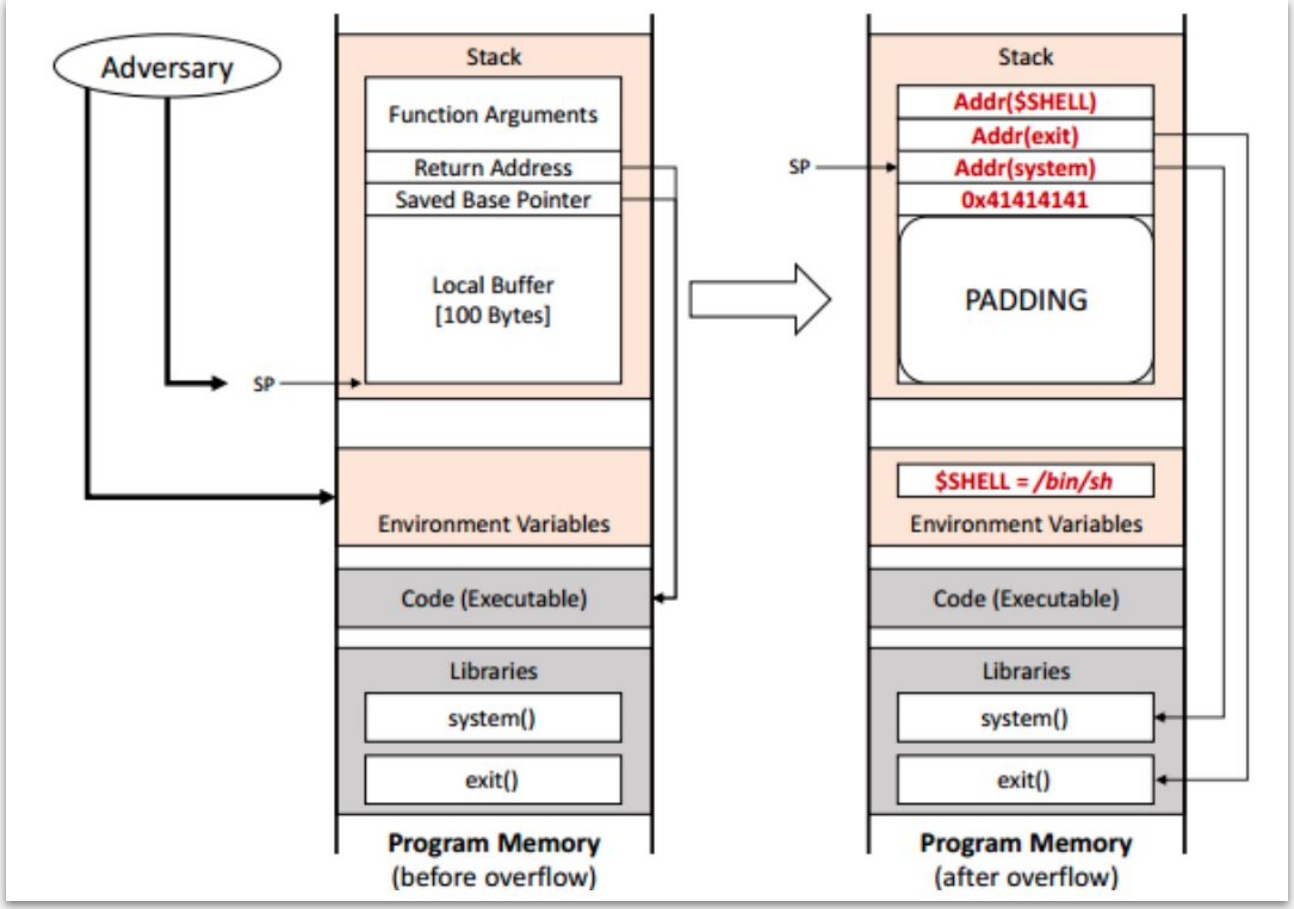
- ❏ Setup fake return address and arguments.
- ❏ The return instruction “calls” the libc function.

Common target functions

- *system*, *execve*, *mprotect*, *malloc*...



Lab Overview



Continue …

Task A :
Find address of *system()*.
- To overwrite return address with *system()*'s address.

Task B :
Find address of the *exit()*.
- To set return of *system()* call.

Task C :
Construct arguments for *system()*.
- Set an environment variable with shell string *"/bin/sh"*.
- Find out the address of the environment variable.
- Set the arguments of *system()* with the address.

Importance of exit()

Attacker would never want to left any trace of the attack.

- So, it is best decision to terminate the program normally without any segmentation fault.

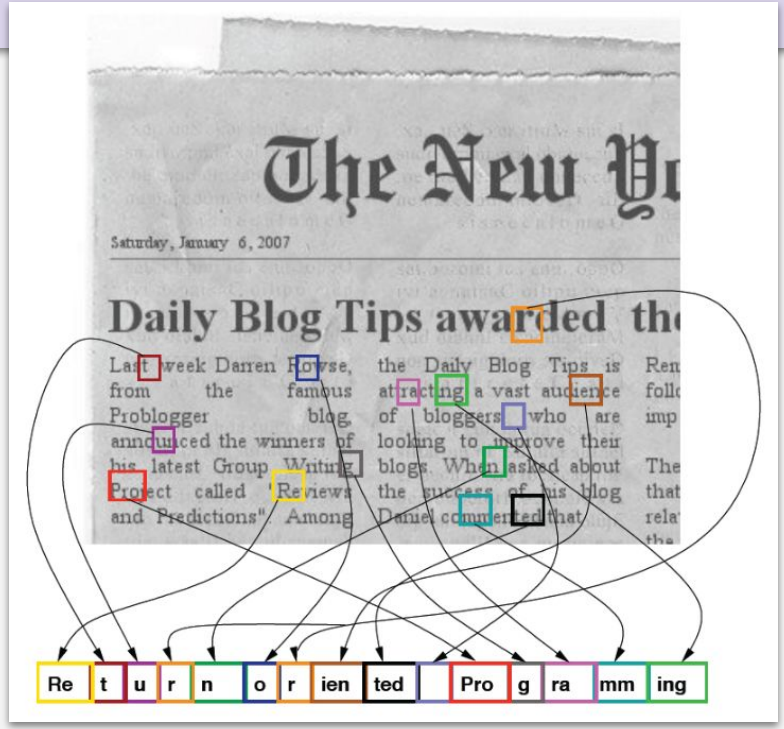
But, it is not necessary that the return of the *system()* call should be *exit()*.

- It is okay to chain multiple libc function calls but should be ended up with *exit()*.

Return Oriented Programming

An attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called "gadgets".

- Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code.
- Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine.



Why?

64-bit processors brought a change to the subroutine calling convention.

- Required the first argument to a function to be passed in a register instead of on the stack.
  - an attacker could no longer set up a library function call with desired arguments just by manipulating the call stack.

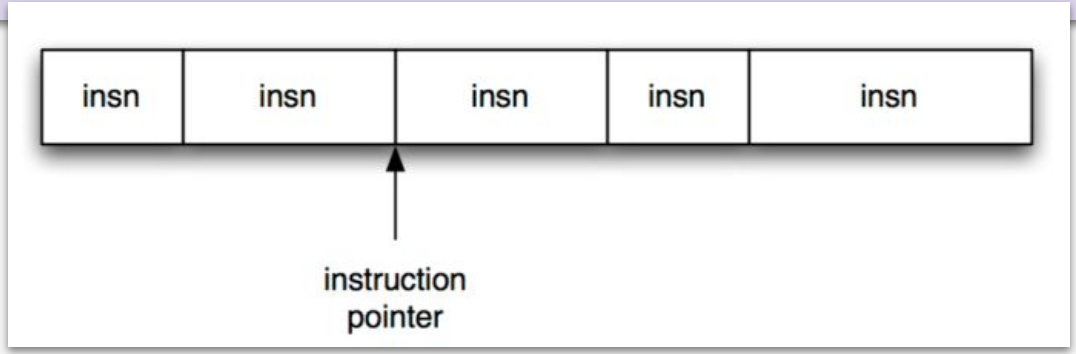
Shared library developers also began to restrict library functions that performed sensitive actions.

- such as system call wrappers.
  - As a result, return-into-library attacks became much more difficult to mount successfully.

Normal Code Execution

Instruction pointer (EIP) determines which instruction to fetch & execute.

- CPU automatically advances EIP to next instruction after executing the current one.
- Control flow instructions change the value of EIP.
  - (conditional) jmps, call, return.

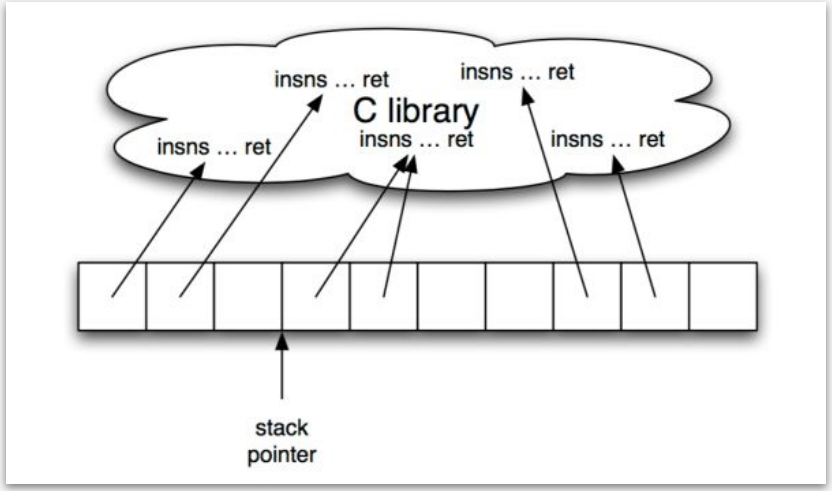


ROP Execution

Stack pointer (ESP) determines which instruction sequence to fetch & execute.

- An instruction sequence is also called a gadget.
- A gadget usually ends with a ret instruction.

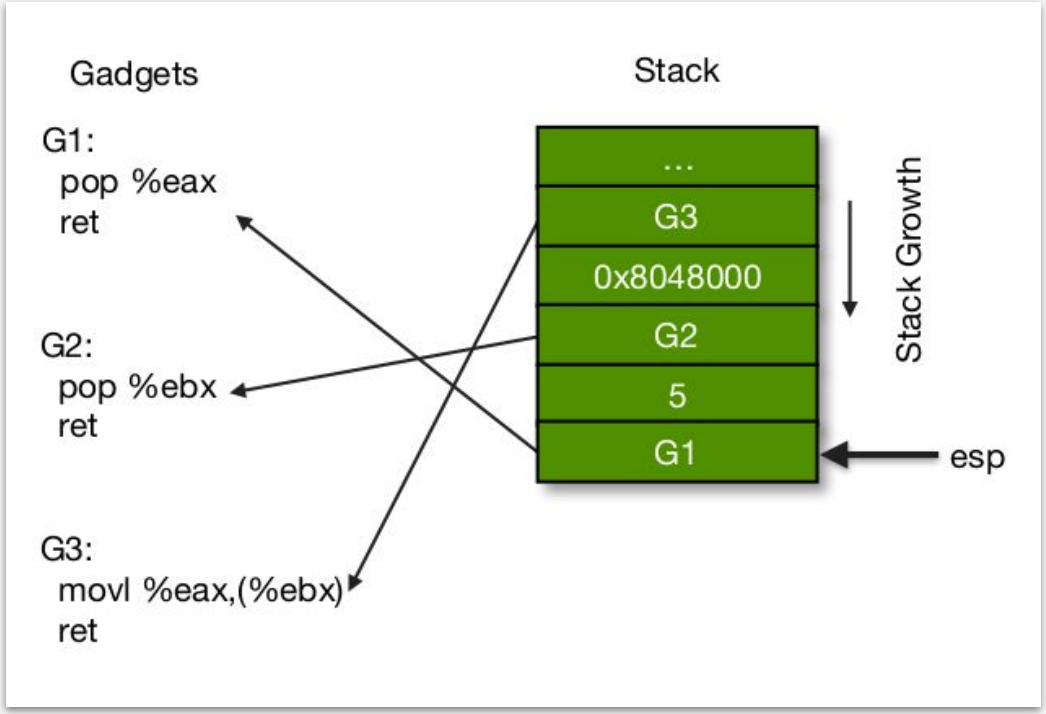
CPU does NOT automatically advance ESP; but the ret at end of each gadget does.



ROP Example

Use ESP as program counter to store 5@0x8048000.

- Without introducing new code.



ROP Thesis

Any sufficiently large program codebase
&
Attacker control of the stack

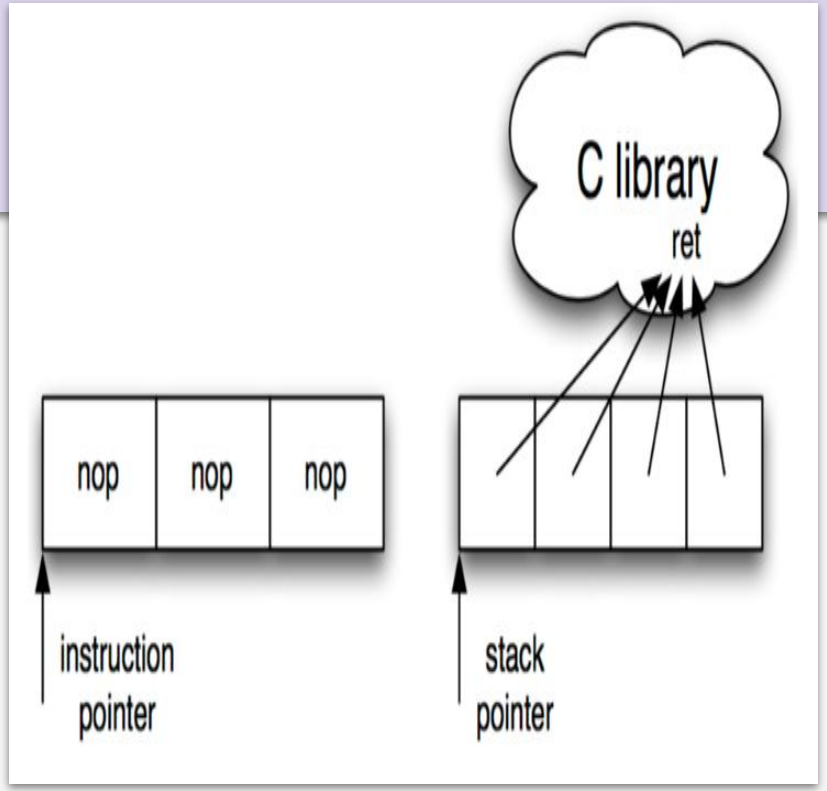
Arbitrary attacker computation and behavior,
without code injection

(in the absence of control-flow integrity)

ROP Building Blocks – NOPs

NOP instruction does nothing but advance EIP.
- Useful in NOP sled.

ROP equivalent:
- Ret-only gadgets to advance ESP.

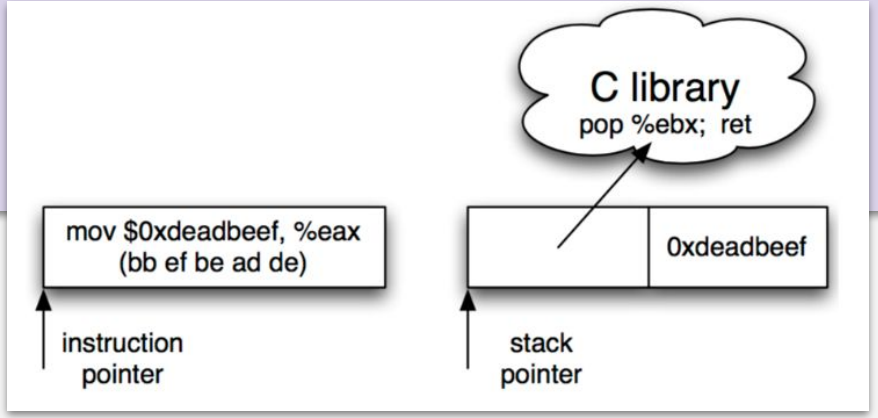


ROP Building Blocks - Constants

Instructions can directly encode constants.

ROP equivalent:

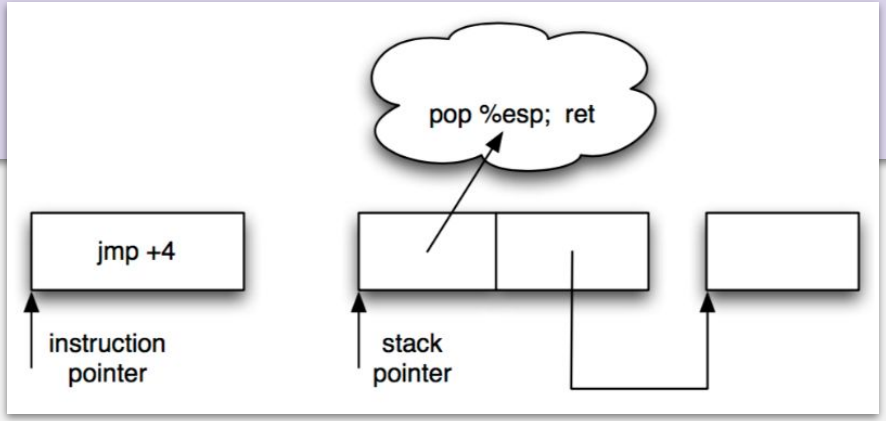
- Store constants on the stack.
- Pop into the register to use.



ROP Building Blocks - Control Flow

Branch instructions can (conditionally) set EIP to new values.

ROP equivalent:
- (Conditionally) set ESP to new value.



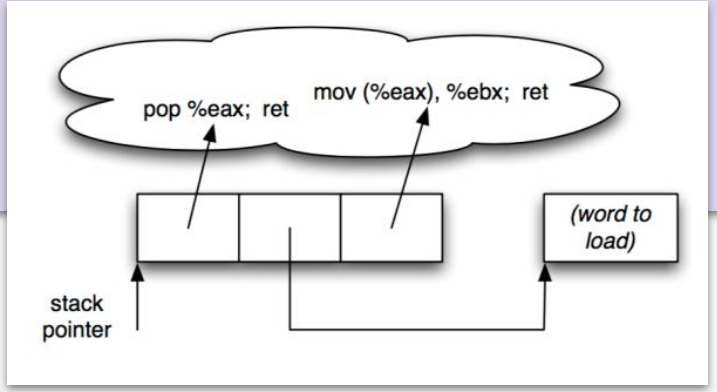
ROP Building Blocks - Multiple Gadgets

Sometimes multiple gadgets needed to encode logic unit.

- e.g., memory loading, arithmetics, conditional branches...

Example: to load memory to register:

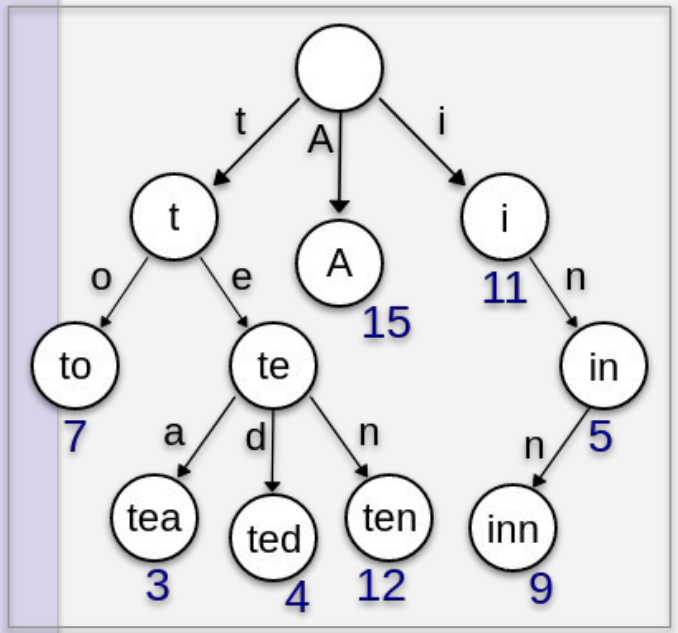
- Load address of source word into %eax.
- Load memory at (%eax) into %ebx.



Finding Gadgets

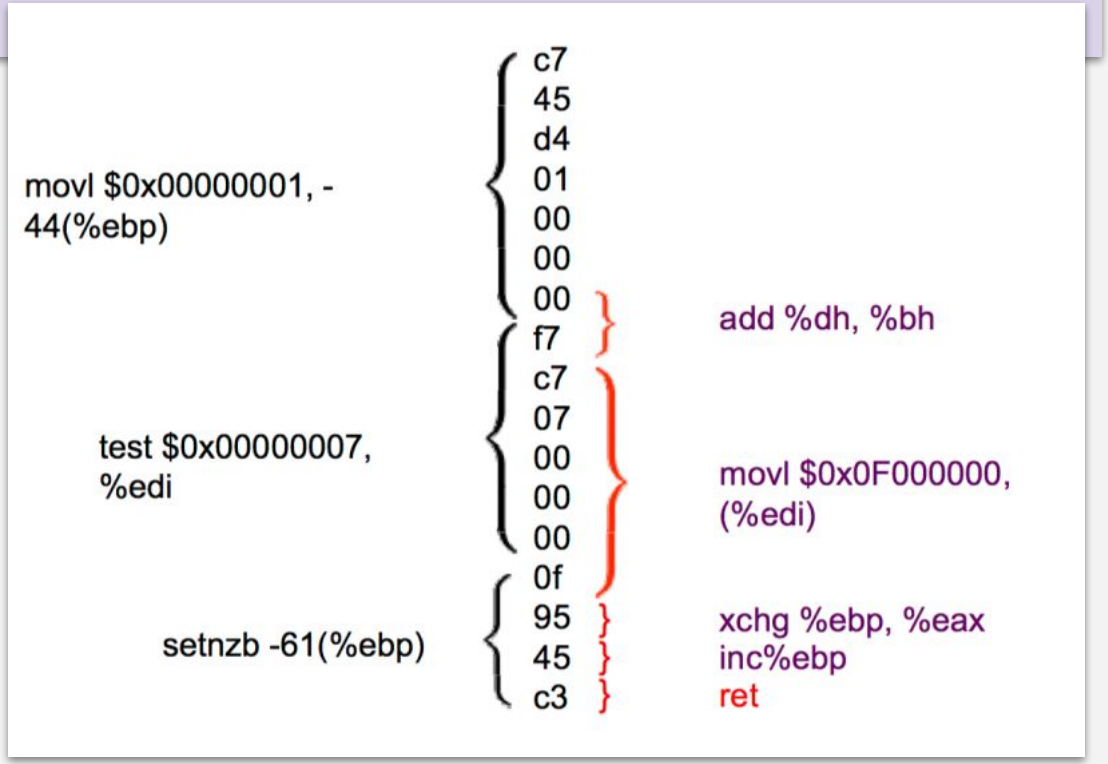
Finding instruction sequences:

- Any instruction sequence ending with *"ret"* is useful — could be part of a gadget.
- **Algorithmic problem:** recover all sequence of valid instructions from libc that end in a *"ret"* instruction.
- **Idea:** at each *ret (C3 in hex)*, look back.
  - Are preceding **i** bytes a valid length-**i** instruction?
  - Backward recursive traverse from found instructions.
  - Collect instruction sequences in a *trie*.



Unintended Instructions (ecb_crypt, x86)

X86 has variable-length instructions, unintended instructions can be “created” by jumping to the middle of original instructions.



Recent Advances in ROP

ROP is Turing complete for both x86 and other RISC CPUs.
- Turing complete ROP compiler: https://github.com/pakt/ropc.
- Though, attacker does not care about Turing completeness.

Jump-oriented programming was proposed.

Just-in-time ROP (JIT ROP).
- Defeat fine-grained code randomization.
- Recursively exploiting a memory disclosure to map the code of the victim process on-the-fly.
- Discover gadgets and JIT-compile a ROP program.

Blind ROP (BROP).
- Remote brute-force ROP without knowing the target program.

Block Oriented Programming Compiler (BOPC).
- automatically synthesizing arbitrary, Turing-complete, Data-Only payloads.

ROP: Summary

- Code injection is not necessary for arbitrary exploitation.
- Defenses that distinguish “good code” from “bad code” are useless.
- Return-oriented programming likely possible on every architecture, not just x86.
- Compilers make sophisticated return-oriented exploits easy to write.

Ret2eax

- Overflow ***buf*** in ***msglog*** to place shellcode in ***buf.***
- ***Strcpy*** saves the ***buf*** address in ***eax.***
- Hijacking control flow to a subsequent ***call *eax*** runs shellcode.

```
void msglog(char *input)
{
    char buf[64];
    strcpy(buf, input);
}

int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("exploitme <msg>\n");
        return -1;
    }

    msglog(argv[1]);
    return 0;
}
```

strcpy returns a pointer to **buf** in register **eax**

Overwrite the return addr with addr of "**call *eax**"

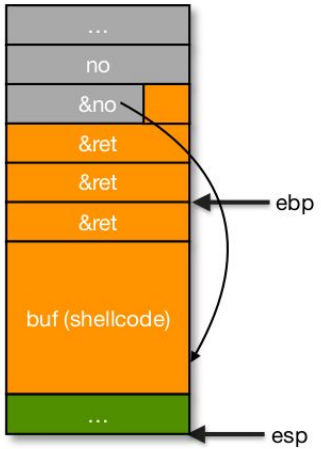
Ret2ret (Stack Juggling)

Stack already has a pointer to the stack (a local variable).

- Overwrite one byte of this pointer (with 0) to make it point to the shellcode.
- Vulnerable buffer sits lower on the stack than the pointer.

Return (with several ***ret*** instructions) to this pointer to run the shellcode.

```
void f(char *str)
{
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    int no = 1;
    int *ptr = &no;
    f(argv[1]);
}
```



Homework 2: Shellcoding