# 程序的机器级表示：高级话题

Machine-Level Programming : Advanced Topics

# 本章内容

Topic

- 内存布局

  Memory Layout

- 缓冲区溢出

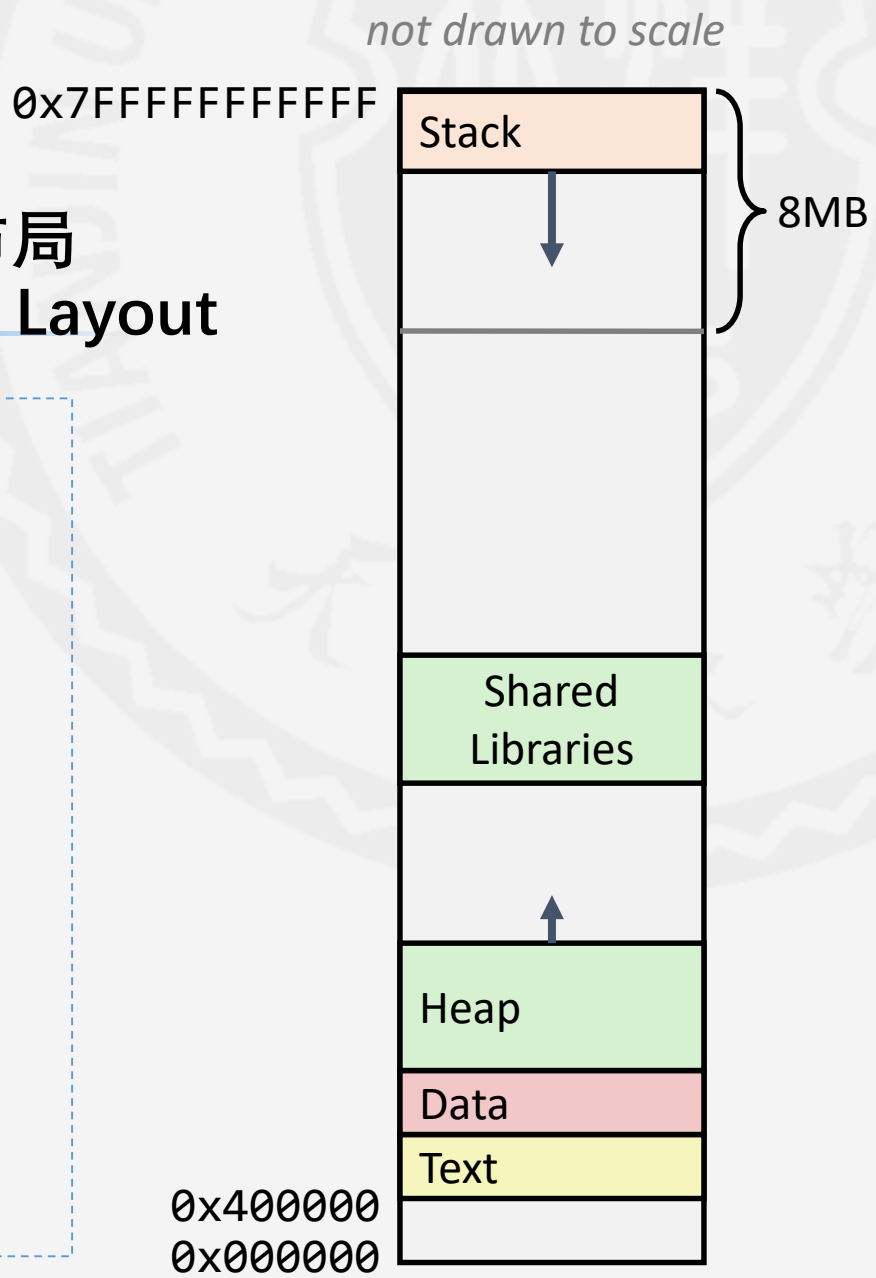
  Buffer Overflow

# 内存布局

Memory Layout

## X86-64/Linux 程序的内存布局
## x86-64/Linux Program Memory Layout

- 栈
  Stack
  - 运行时栈（默认：8MB的限制）
    Runtime stack (8MB limit)
  - 例如：局部变量
    E. g., local variables
- 堆
  Heap
  - 根据需要动态分配
    Dynamically allocated as needed
  - 当调用 malloc()、calloc()和new()时申请
    When call  malloc(), calloc(), new()
- 数据
  Data
  - 静态分配的数据
    Statically allocated data
  - 例如：全局变量、静态变量、字符串常量
    E.g., global vars, static vars, string constants
- 代码/共享库
  Text / Shared Libraries
  - 可执行的机器指令
    Executable machine instructions
  - 只读
    Read-only



# 内存布局

Memory Layout

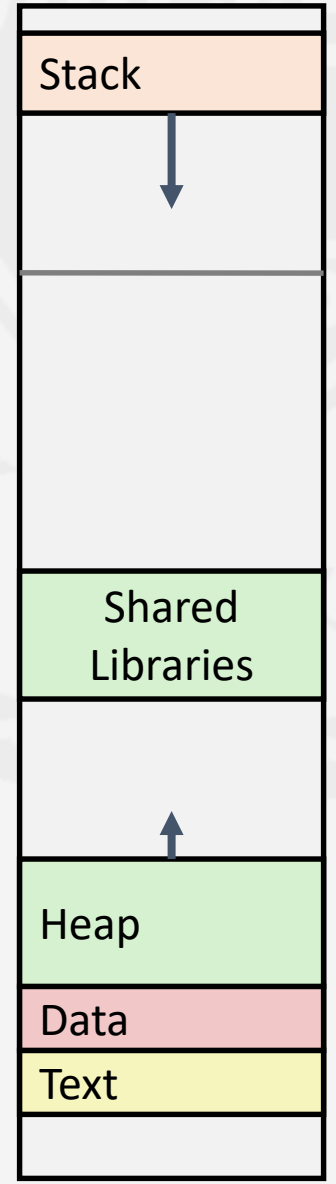
## 举例：内存分配
## Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
}
```

*都在哪些位置申请了内存?*

*Where does everything go?*

*not drawn to scale*



# 内存布局

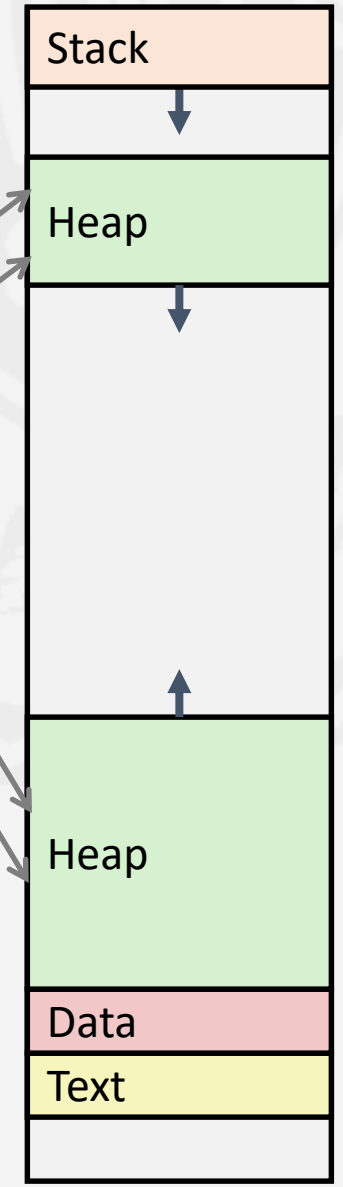
Memory Layout

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
}
```

*not drawn to scale*

*address range ~$2^{48}$*

| | |
|---|---|
| local | 0x00007ffe4d3be87c |
| p1 | 0x00007f7262a1e010 |
| p3 | 0x00007f7162a1d010 |
| p4 | 0x000000008359d120 |
| p2 | 0x000000008359d010 |
| big_array | 0x0000000080601060 |
| huge_array | 0x0000000000601060 |
| main() | 0x000000000040060c |
| useless() | 0x0000000000400590 |



# 本章内容

Topic

- 内存布局

  Memory Layout

- 缓冲区溢出

  Buffer Overflow

  - 漏洞

    Vulnerability

  - 对抗

    Thwarting

# 缓冲区溢出

Buffer Overflow

## 计算机的漏洞
## Vulnerability of computer

- 1988年11月
  November, 1988
  - 互联网蠕虫攻击了成千上万的互联网主机
    Internet Worm attacks thousands of Internet hosts
  - 这是如何发生的?
    How did it happen?

- 2014年4月
  April, 2017
  - 心脏滴血漏洞
    Heartbleed

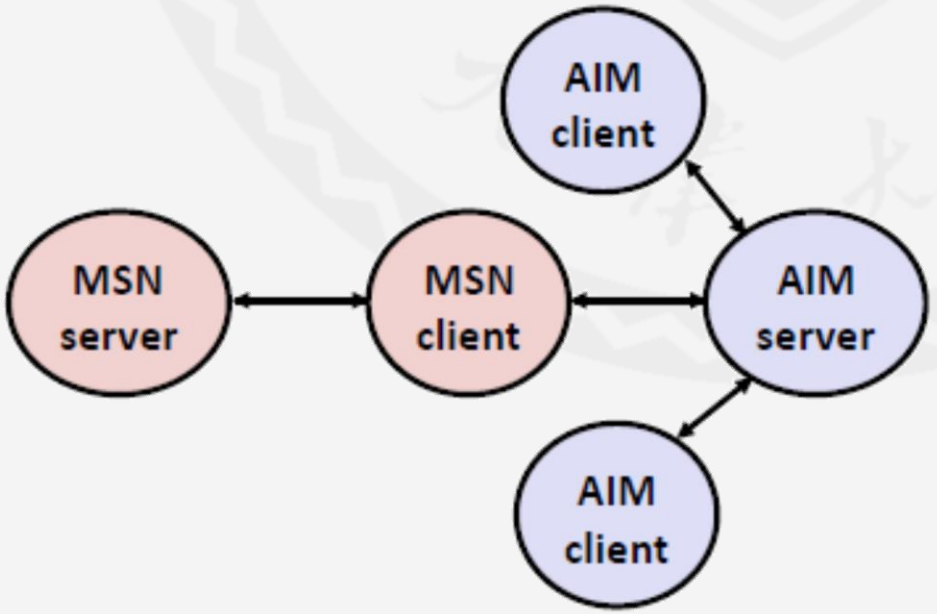
- 2017年4月
  April, 2017
  - 勒索蠕虫
    WannaCry Worm

# 缓冲区溢出

Buffer Overflow

## 即时通讯软件战争
## IM War

- 1999年7月
  July, 1999
  - 微软公司推出了MSN Messenger（即时通讯软件）
    Microsoft launches MSN Messenger (instant messaging system)
  - MSN客户端可以直接访问当时流行的AOL公司的即时通讯软件（AIM）的服务器
    Messenger clients can access popular AOL Instant Messaging Service (AIM) servers

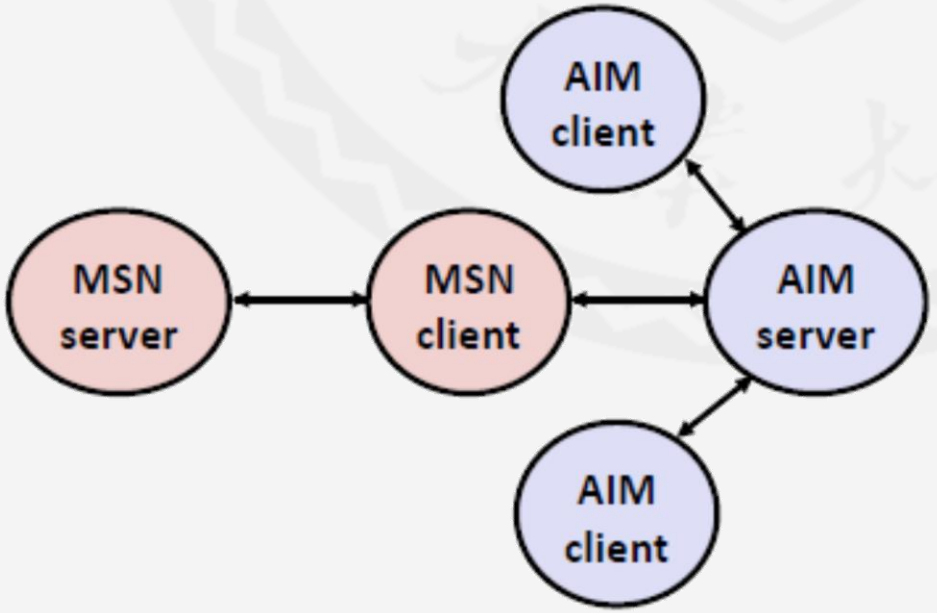


# 缓冲区溢出

Buffer Overflow

## 即时通讯软件战争
## IM War

- 1999年8月
  August, 1999
  - 奇怪的是，Messenger客户端无法再访问
    Mysteriously, Messenger clients can no longer access AIM servers
  - 微软和AOL开始了即时通讯软件战争:
    Microsoft and AOL begin the IM war:
    - AOL修改了服务器程序以拒绝MSN客户端的接入
      AOL changes server to disallow Messenger clients
    - 微软对MSN做出修改以应对AOL
      Microsoft makes changes to clients to defeat AOL changes
    - 如此这般，交手了至少13个回合
      At least 13 such skirmishes
  - 这是如何发生的?
    How did it happen?



# 小知识：蠕虫和病毒
# Worms and Viruses

- 蠕虫：一个程序
  Worm: A program that
  - 可以独立运行
    Can run by itself
  - 可以将其完整的工作版本传播到其他计算机
    Can propagate a fully working version of itself to other computers
- 病毒：一个程序片段
  Virus: Code that
  - 注入到其他的程序中
    Add itself to other programs
  - 不能够独立运行
    Cannot run independently

# 缓冲区溢出

Buffer Overflow

## 这样的问题很严重
## Such problems are a BIG deal

- 通常是由一种叫做“缓冲区溢出”的问题造成的
  Generally called a “buffer overflow”
  - 当对一个数组的访问超出的其内存分配的区域时
    when exceeding the memory size allocated for an array
- 为什么很严重?
  Why a big deal?
  - 它是安全漏洞产生的头号技术原因
    It’s the #1 technical cause of security vulnerabilities
  - 这个头号原因主要是由于社会上的工程师/用户的无知造成的
    #1 overall cause is social engineering / user ignorance

- 最常见的缓冲区溢出的形式
  Most common form
  - 没有对输入字符串的长度进行检查
    Unchecked lengths on string inputs
  - 特别是栈上面的字符数组
    Particularly for bounded character arrays on the stack
    - 有时被称为栈破坏
      sometimes referred to as stack smashing

# 字符串库代码
# String Library Code

Unix 中 gets() 函数的实现
Implementation of Unix function gets()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

对读入的字符数量没有做具体的限制
No way to specify limit on number of characters to read

# 缓冲区溢出

Buffer Overflow

## C标准库中已经不推荐使用的函数
## Deprecated Functions in C Library

- 在其他的库函数中也有类似的问题
  Similar problems with other library functions
  - `strcpy, strcat`：会进行任意长度的字符串赋值
    `strcpy, strcat` : copy strings of arbitrary length
  - `scanf, fscanf, sscanf`：会使用 `%s` 转换模式
    `scanf, fscanf, sscanf` : when given %s conversion specification

| Deprecated Functions |
| --- |
| `char *strcpy(char* dest, const char *src);` |
| `char *strcat(char *dest, const char *src);` |
| `int scanf(const char *format, [argument...]);` |
| `int fscanf(FILE *stream, const char *format, [argument...]);` |
| `int sscanf(const char *buffer, const char *format, [argument...]);` |

# 缓冲区溢出

Buffer Overflow

## 缓冲区相关的代码漏洞
## Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

多大才算足够大呢?

btw, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901
0123456789012345678901
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

# 反汇编缓冲区相关的代码
# Buffer Overflow Disassembly

echo:

```
0000000004006cf <echo>:
 4006cf:   48 83 ec 18              sub    $0x18,%rsp
 4006d3:   48 89 e7                 mov    %rsp,%rdi
 4006d6:   e8 a5 ff ff ff           callq  400680 <gets>
 4006db:   48 89 e7                 mov    %rsp,%rdi
 4006de:   e8 3d fe ff ff           callq  400520 <puts@plt>
 4006e3:   48 83 c4 18              add    $0x18,%rsp
 4006e7:   c3                       retq
```

call_echo:

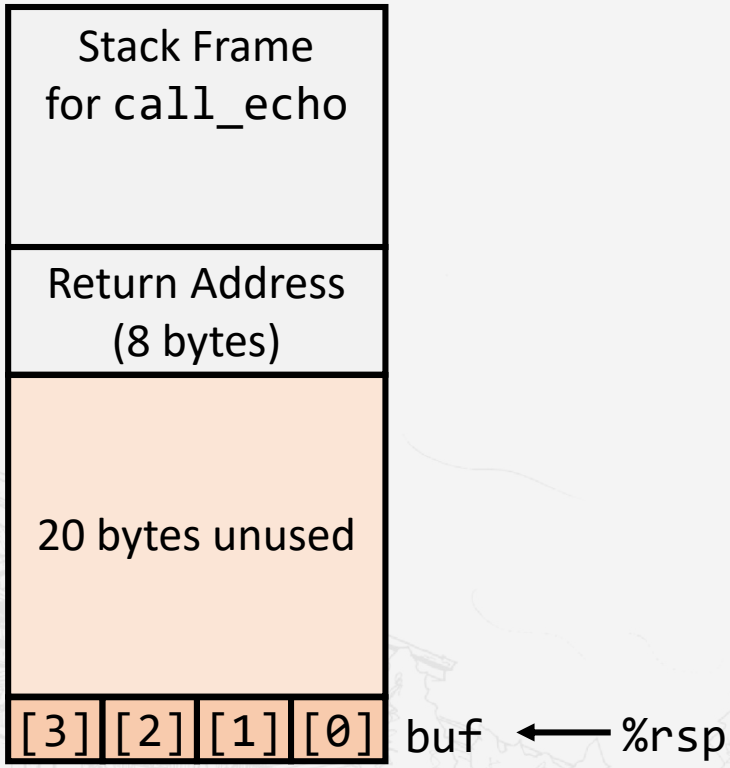
```
 4006e8:   48 83 ec 08              sub    $0x8,%rsp
 4006ec:   b8 00 00 00 00           mov    $0x0,%eax
 4006f1:   e8 d9 ff ff ff           callq  4006cf <echo>
 4006f6:   48 83 c4 08              add    $0x8,%rsp
 4006fa:   c3                       retq
```

# 缓冲区溢出

Buffer Overflow

## 缓冲区相关代码的栈结构
## Stack Structure of Buffer Overflow Code

*调用 gets 函数前*
*Before call to gets*




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

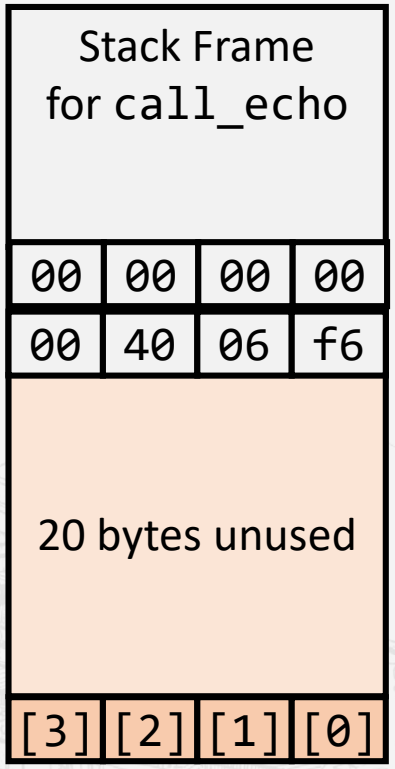
```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

# 缓冲区溢出

Buffer Overflow

## 缓冲区相关代码的栈结构
## Stack Structure of Buffer Overflow Code

*调用 gets 函数前*
*Before call to gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

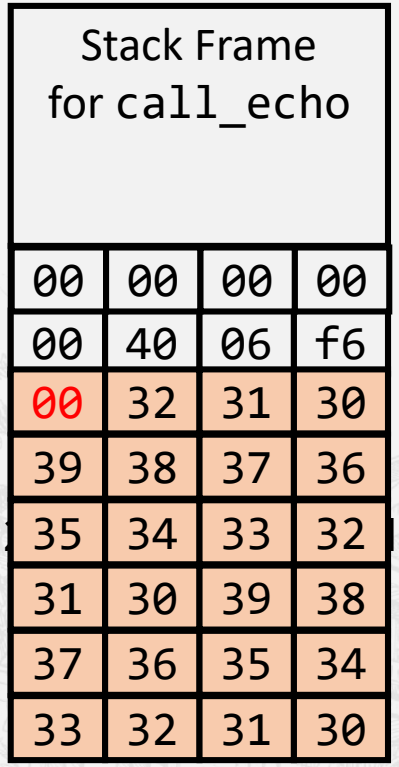
```
  . . .
4006f1:   callq  4006cf <echo>
4006f6:   add    $0x8,%rsp
  . . .
```

# 缓冲区溢出

Buffer Overflow

## 缓冲区相关代码的栈结构 （情况1）
## Stack Structure of Buffer Overflow Code （Case 1）

*gets函 数调用后*
*After call to gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

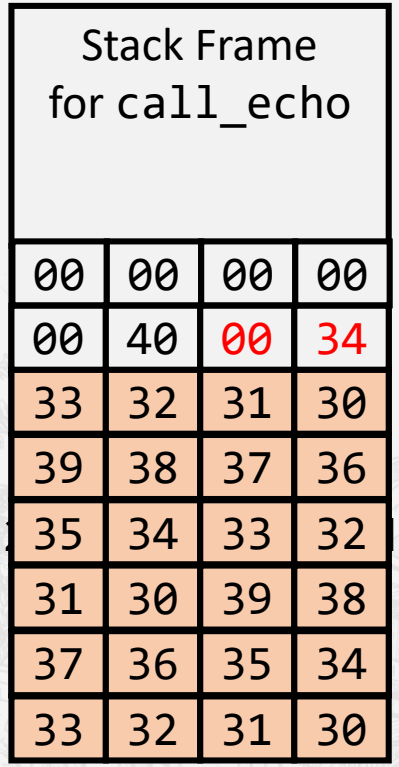
```
. . .
4006f1:   callq  4006cf <echo>
4006f6:   add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901
0123456789012345678901
```

缓冲区溢出，但是没有破坏栈的结构
Overflowed buffer, but did not corrupt stack structure

# 缓冲区相关代码的栈结构 （情况2）
# Stack Structure of Buffer Overflow Code （Case 2）

*gets函 数调用后*
*After call to gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
4006f1:   callq  4006cf <echo>
4006f6:   add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

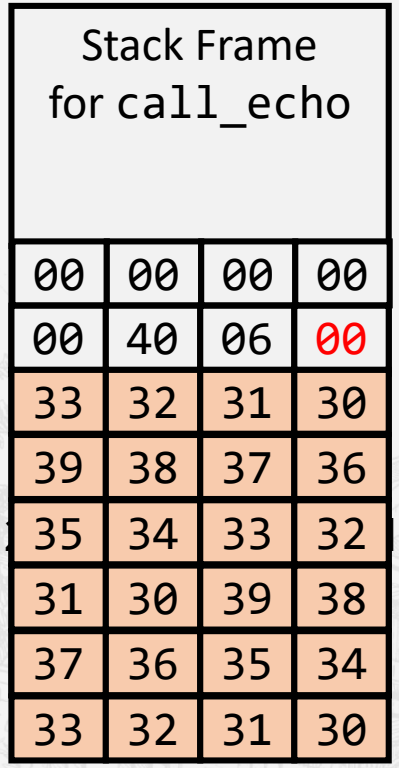
缓冲区溢出，且破坏了函数返回地址
Overflowed buffer and corrupted return pointer

# 缓冲区溢出

Buffer Overflow

## 缓冲区相关代码的栈结构 （情况3）
## Stack Structure of Buffer Overflow Code （Case 3）

*gets函 数调用后*
*After call to gets*




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
. . .
4006f1:   callq  4006cf <echo>
4006f6:   add    $0x8,%rsp
. . .
```

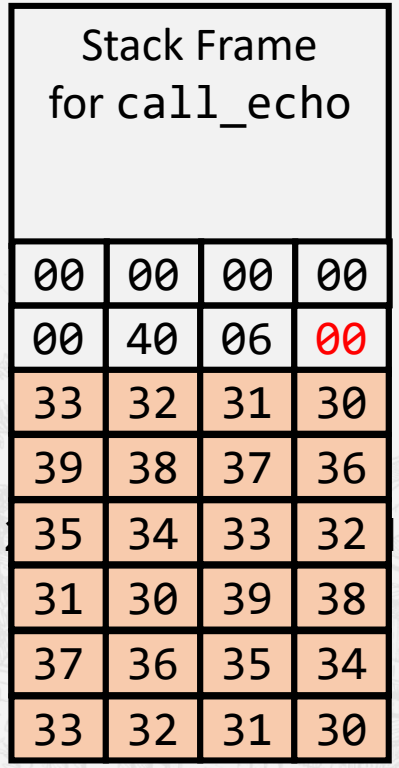
```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123456789012345678901234567890123
```

缓冲区溢出，且破坏了函数返回地址，但是程序好像还可以工作

Overflowed buffer, corrupted return pointer, but program seems to work!

# 缓冲区相关代码的栈结构 （情况3）
# Stack Structure of Buffer Overflow Code （Case 3）

*gets函 数调用后*
*After call to gets*




```
register_tm_clones:
  . . .
  400600:   mov    %rsp,%rbp
  400603:   mov    %rax,%rdx
  400606:   shr    $0x3f,%rdx
  40060a:   add    %rdx,%rax
  40060d:   sar    %rax
  400610:   jne    400614
  400612:   pop    %rbp
  400613:   retq
```

返回到一段无关的代码区域
“Returns” to unrelated code
发生了很多事，但是没有改变关键状态
Lots of things happen, without modifying critical state
最后执行retq返回到main函数
Eventually executes `retq` back to `main`

# 缓冲区溢出

Buffer Overflow

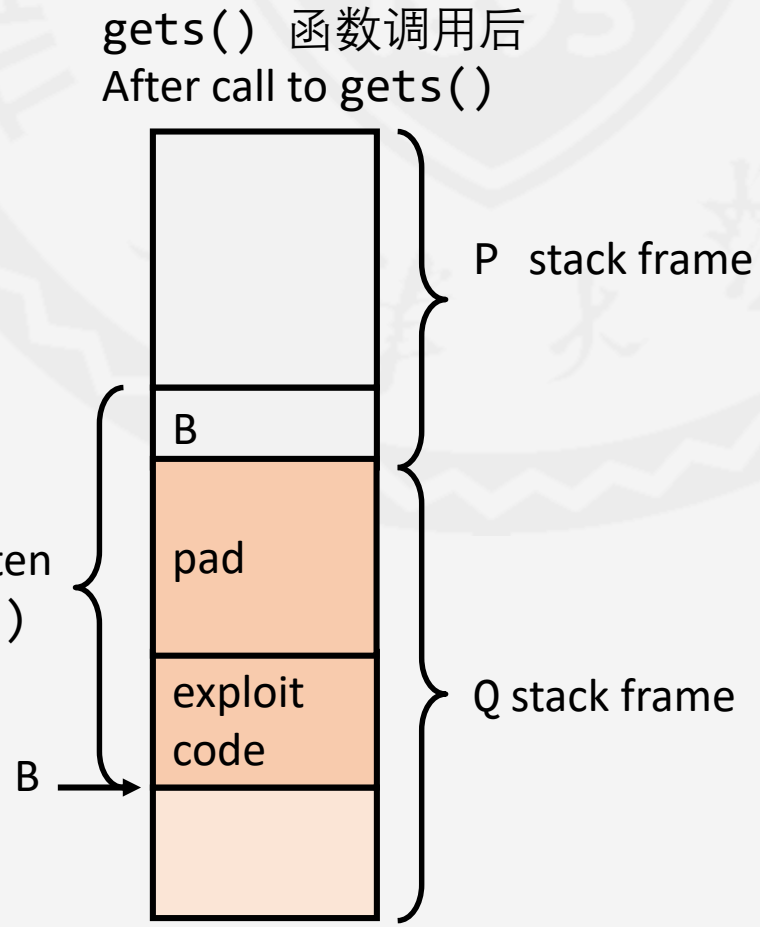
## 代码注入攻击
## Code Injection Attacks

- 输入的字符串中包含着可执行的字节编码
  Input string contains byte representation of executable code
- 将返回地址A覆盖为缓冲区的地址B
  Overwrite return address A with address of buffer B
- 当过程 Q 执行 ret 指令，将会跳转到攻击代码
  When Q executes ret, will jump to exploit code

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



# 基于缓冲区溢出的攻击
# Exploits Based on Buffer Overflows

- 缓冲区溢出的漏洞允许远程计算机在受害计算机上执行任意代码
  Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines
- 这在实际的程序中十分常见
  Distressingly common in real programs
  - 程序员不断地犯着同样的错误 ☹
    Programmers keep making the same mistakes ☹
  - 近年来也采取了一些措施让这些攻击变得更加困难
    Recent measures make these attacks much more difficult

# 缓冲区溢出

Buffer Overflow

## 即时通讯软件战争
## IM War

- AOL公司攻击了AIM客户端中的缓冲区溢出漏洞
  AOL exploited existing buffer overflow bug in AIM clients
- 攻击代码：向服务器发送4个字节的签名信息（这些字节位于AIM客户端的某个位置）
  Exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server
- 当微软实现了对这个签名的匹配后，AOL修改了签名的位置（生成了新的签名）
  When Microsoft changed code to match signature, AOL changed signature location

# 缓冲区溢出

Buffer Overflow

- 一封未知来源的电子邮件结束了这场战争

  An email from an unknown source ended the war

- 随后发现这份邮件来自于微软内部

  It was later determined that this email originated from within Microsoft

- 工程师的职业操守

  Professional ethics of engineers

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

***It was later determined that this email originated from within Microsoft!***

# 本章内容

Topic

- 内存布局

  Memory Layout

- 缓冲区溢出

  Buffer Overflow

  - 漏洞

    Vulnerability

  - 对抗

    Thwarting

# 对抗缓冲区溢出攻击
# Thwarting Buffer Overflow Attacks

- 避免溢出漏洞的出现
  Avoid overflow vulnerabilities
- 使用系统级保护策略
  Employ system-level protections
- 让编译器使用栈“金丝雀”
  Have compiler use “stack canaries”

# 缓冲区溢出

Buffer Overflow

## 避免溢出漏洞的出现
## Avoid overflow vulnerabilities

- 例如：使用限制字符串长度的标准库函数
  For example, use library routines that limit string lengths
  - 使用 `fgets` 代替 `gets`
    `fgets` instead of `gets`
  - 使用 `strncpy` 代替 `strcpy`
    `strncpy` instead of `strcpy`
  - 不使用 `scanf` 的 `%s` 转换模式获取字符串
    Don't use `scanf` with `%s` conversion specification
    - 使用 `fgets` 读取字符串
      Use `fgets` to read the string
    - 或者使用 `%ns`，n 选择一个合适的整数
      Or use `%ns` where n is a suitable integer

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

# 缓冲区溢出

Buffer Overflow

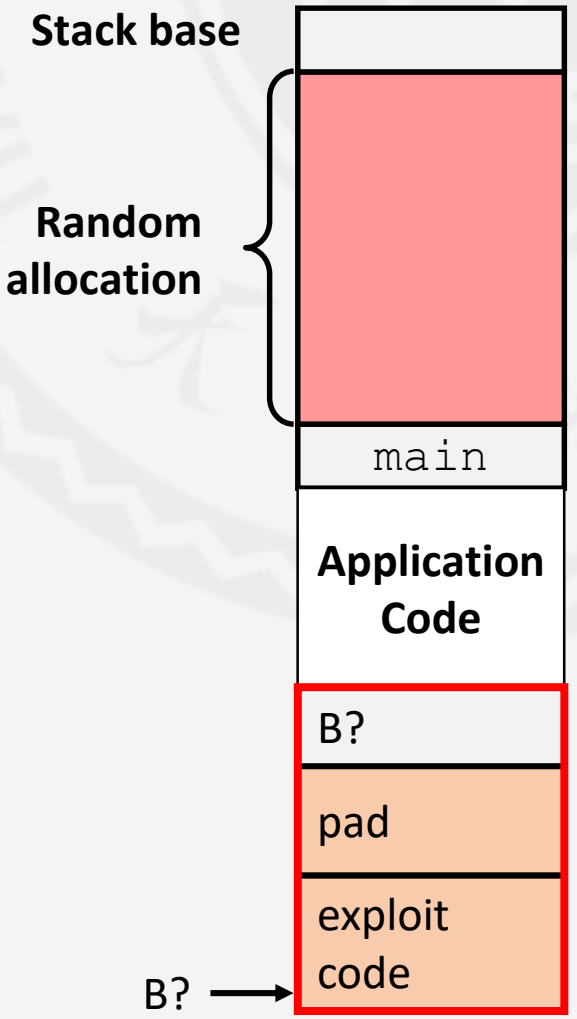
## 系统级的保护机制
## System-Level Protections

- 栈随机化
  Randomized stack offsets
  - 在程序的开头，分配一个随机大小的栈空间
    At start of program, allocate random amount of space on stack
  - 这会移动整个程序的栈地址
    Shifts stack addresses for entire program
  - 使得“黑客”很难预测注入代码的准确位置
    Makes it difficult for hacker to predict beginning of inserted code
- 例如：五次执行内存分配时缓冲区B的地址
  E.g.: 5 executions of memory allocation for buffer B

  `0x7ffe4d3be87c` `0x7fff75a4f9fc`

  `0x7ffeadb7c80c` `0x7ffeaea2fdac`

  `0x7ffcd452017c`
- 每一次运行，栈的位置都不一样
  Stack repositioned each time program executes

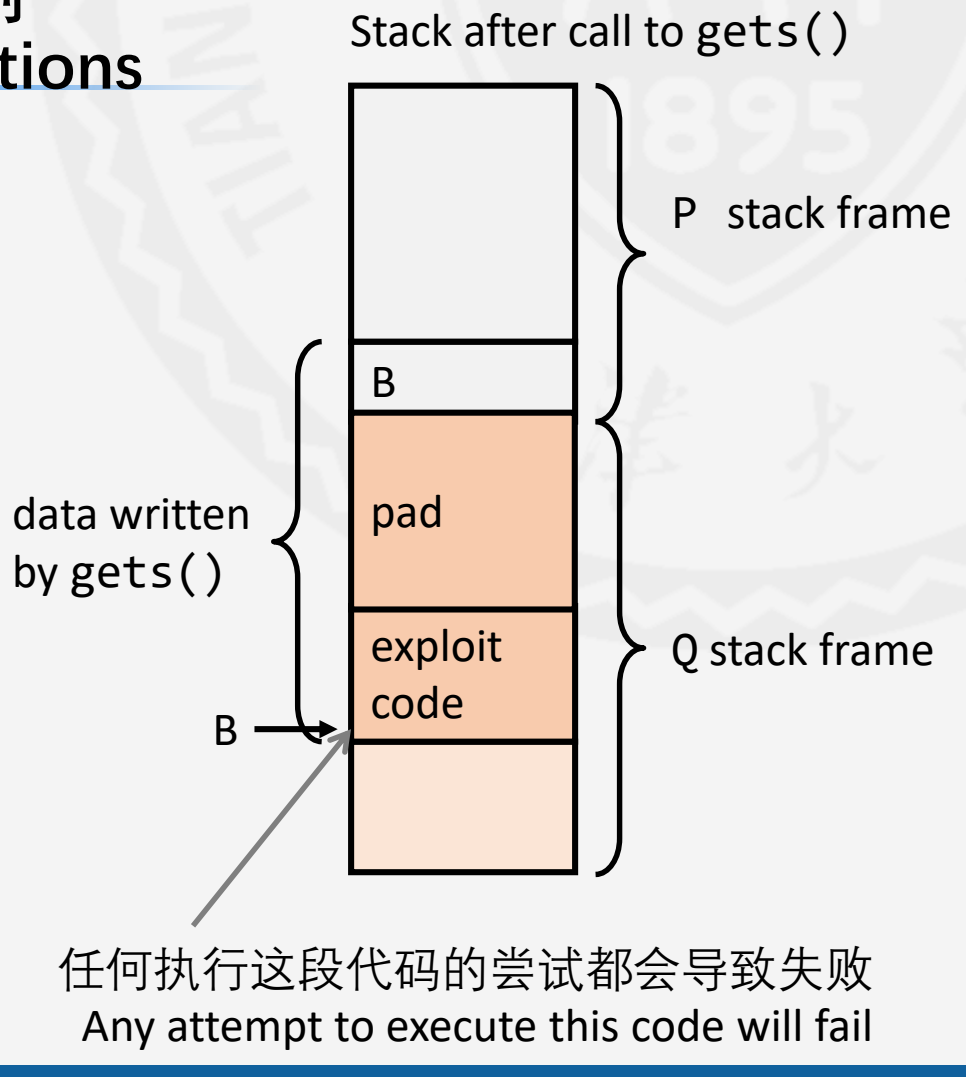


# 缓冲区溢出

Buffer Overflow

## 系统级的保护机制
## System-Level Protections

- 不可执行段 （限制可执行代码的区域）
  Nonexecutable code segments
  - 在传统的x86中，只能将内存区域标记为“可读”和“可写”
    At start of program, allocate random amount of space on stack
    - 任何“可读” 的内存区域都可以执行代码
      Can execute anything readable
  - x86-64增加了显示的“可执行”权限
    X86-64 added  explicit “execute” permission
  - 把栈标识为“不可执行”
    Stack marked as non-executable
  - “虚拟内存” 一章会详细说明这个问题
    The chapter “Virtual Memory” will give the explanation in detail



# 缓冲区溢出

Buffer Overflow

## 栈“金丝雀”
## Stack Canaries

- 思想
  Idea
  - 在栈中，缓冲区的后面紧接着放置一个特殊值（“金丝雀”）
    Place special value (“canary”) on stack just beyond buffer
  - 在函数返回前，检查这个值是否被破坏
    Check for corruption before exiting function
- GCC 实现
  GCC Implementation
  - 编译选项： **`-fstack-protector`**
    Compiler option: **`-fstack-protector`**
  - 现在这个选项是默认的（早期版本这个选项的关闭的）
    Now the default (disabled earlier)

小知识：“金丝雀”源于历史上用这种鸟在煤矿中感知有毒气体
Tips： The term “canary” refers to the historic use of these birds to detect the presence of dangerous gases in coal mine.

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

# 金丝雀：反汇编保护缓冲区代码
# Canaries: Protected Buffer Disassembly

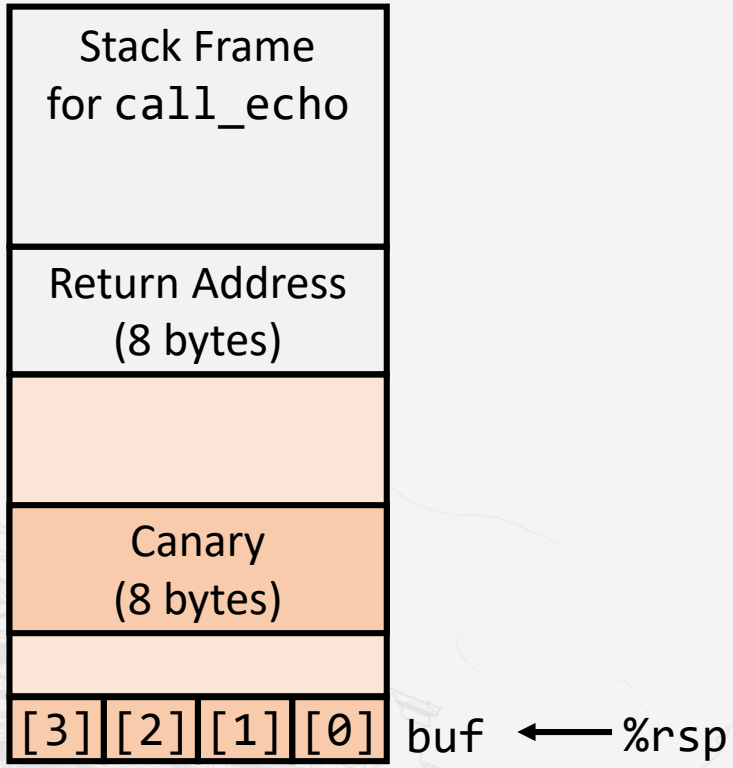
```
echo:
  40072f:   sub    $0x18,%rsp
  400733:   mov    %fs:0x28,%rax
  40073c:   mov    %rax,0x8(%rsp)
  400741:   xor    %eax,%eax
  400743:   mov    %rsp,%rdi
  400746:   callq  4006e0 <gets>
  40074b:   mov    %rsp,%rdi
  40074e:   callq  400570 <puts@plt>
  400753:   mov    0x8(%rsp),%rax
  400758:   xor    %fs:0x28,%rax
  400761:   je     400768 <echo+0x39>
  400763:   callq  400580 <__stack_chk_fail@plt>
  400768:   add    $0x18,%rsp
  40076c:   retq
```

# 缓冲区溢出

Buffer Overflow

## 设置金丝雀
## Setting Up Canary

*调用 gets 前*
*Before call to gets*




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

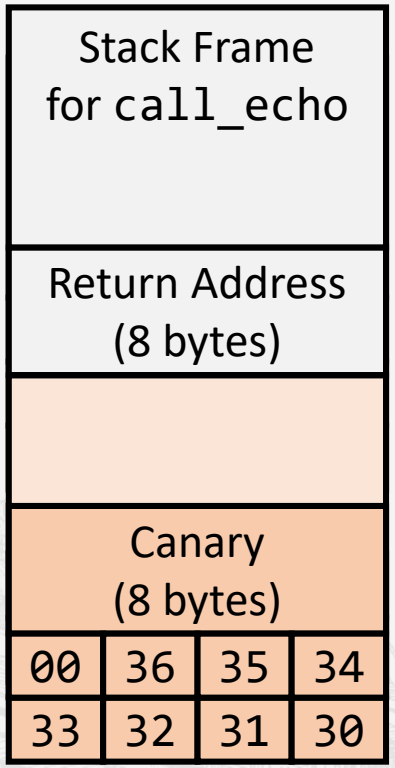
```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

# 缓冲区溢出

Buffer Overflow

## 检查金丝雀
## Checking Canary

*调用gets后*
*After call to gets*




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: *0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax     # Retrieve from stack
    xorq    %fs:40, %rax      # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
.L6:. . .
```

# 缓冲区溢出

Buffer Overflow

## 面向返回编程的攻击
## Return-Oriented Programming Attacks

- 挑战（对于“黑客”的）
  Challenge (for hackers)
  - 栈随机化加大了推断缓冲区的位置的难度
    Stack randomization makes it hard to predict buffer location
  - 标记内存区域的不可执行权限增加了注入二进制代码的难度
    Marking stack nonexecutable makes it hard to insert binary code
- 替代策略
  Alternative Strategy
  - 使用现有代码 （不注入）
    Use existing code
    - 例如：stdlib库中的代码
      E.g., library code from stdlib
  - 将代码片段串联在一起以达到预期效果
    String together fragments to achieve overall desired outcome
  - 无法克服栈金丝雀
    Does not overcome stack canaries

- 把“小组件” 构造成程序
  Construct program from gadgets
  - 每个组件的指令序列以 `ret` 指令作为结束
    Sequence of instructions ending in `ret`
  - `ret` 指令的编码是 0xc3
    `ret` encoded by single byte 0xc3
  - 从一个代码片段运行到另一个代码片段运行的位置固定下来
    Code positions fixed from run to run
  - 所有的代码都是具备可执行权限的
    Code is executable

# 缓冲区溢出

Buffer Overflow

## 举例：小组件 #1
## Gadget Example #1

```
long ab_plus_c (long a, long b, long c)
{
    return a*b + c;
}
```

```
0000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

rax ← rdi + rdx

Gadget address = 0x4004d4

利用现有函数结尾处的代码

Use tail end of existing functions

# 举例：小组件 #2
# Gadget Example #2

```
void setval(unsigned *p)
{
    *p = 3347663060u;
}
```

Encodes  movq %rax, %rdi

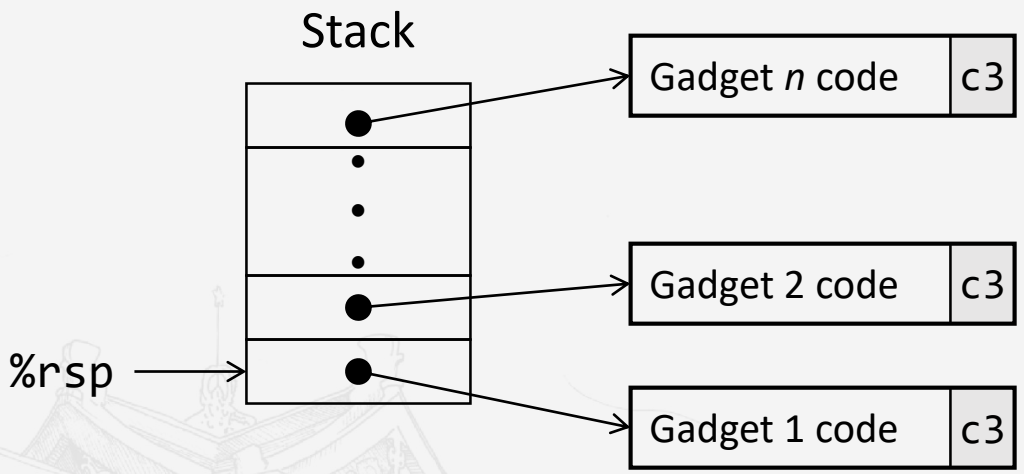
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

Gadget address = 0x4004dc

对字节码的重新利用
Repurpose byte codes

# 面向返回编程攻击的执行
# ROP Execution




- 由 `ret` 指令触发
  Trigger with `ret` instruction
  - 将会执行小组件1
    Will start executing Gadget 1
- 每个小组件最后的 ret 指令会启动下一个“小组件”
  Final ret in each gadget will start next one