# Systems Programming

## Stack Buffer Overflow

**Byoungyoung Lee**
**Seoul National University**
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

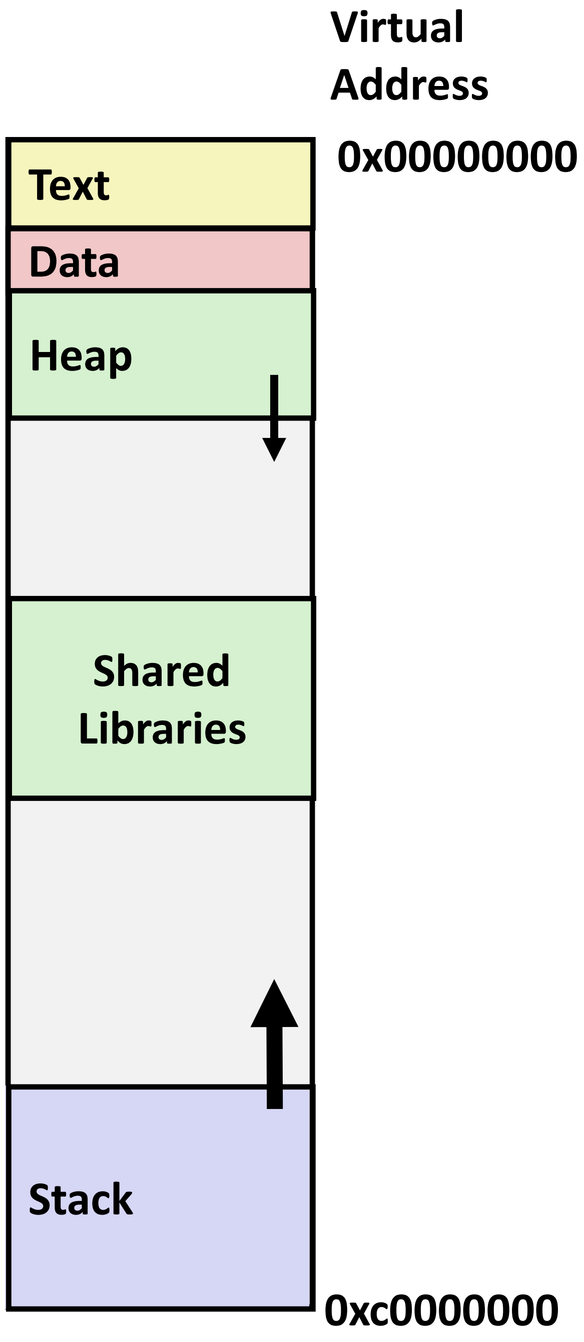
Lecture slides are prepared based on materials provided by CSAPP authors.

# Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

# x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack
  - e.g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`
- **Data**
  - Statically allocated data
  - e.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

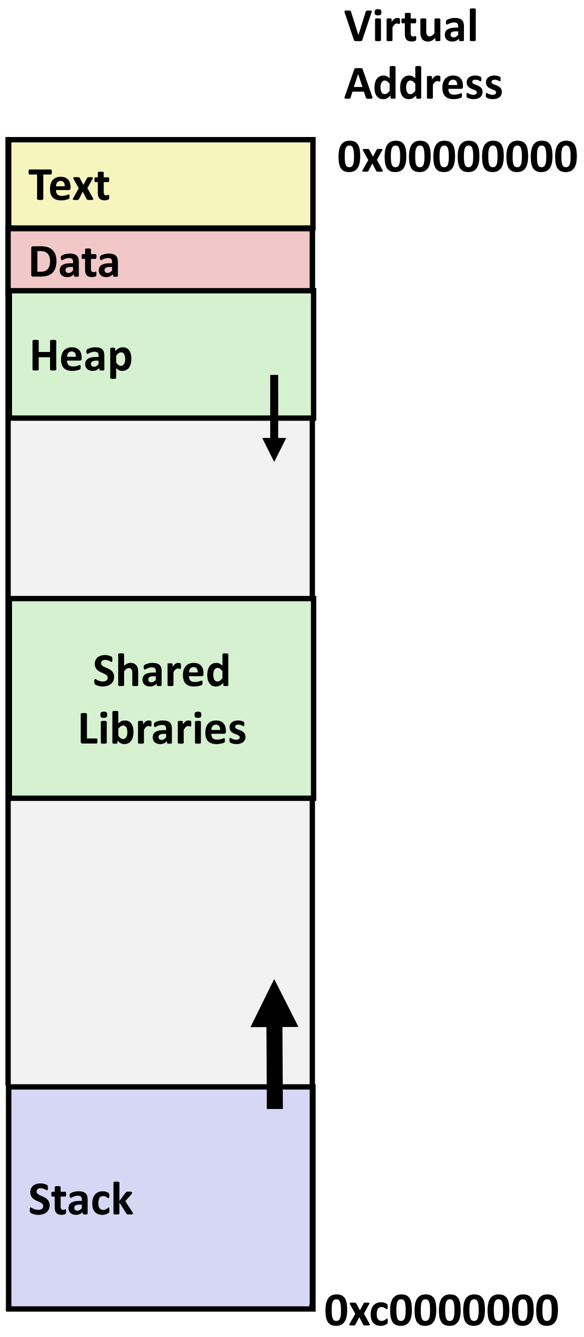



0000

# Memory Layout: 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 *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
}
```

*Q. Where does everything go?*



# Memory Layout: Example

**rwxp:**

- **Read**
- **Write**
- **Execute**
- **Private**

```
└─$ cat /proc/self/maps
562bf109a000-562bf109c000 r--p 00000000 08:20 5834                       /usr/bin/cat
562bf109c000-562bf10a0000 r-xp 00002000 08:20 5834                       /usr/bin/cat
562bf10a0000-562bf10a2000 r--p 00006000 08:20 5834                       /usr/bin/cat
562bf10a2000-562bf10a3000 r--p 00007000 08:20 5834                       /usr/bin/cat
562bf10a3000-562bf10a4000 rw-p 00008000 08:20 5834                       /usr/bin/cat
562bf1930000-562bf1951000 rw-p 00000000 00:00 0                          [heap]
7f6e25d7b000-7f6e25d9d000 rw-p 00000000 00:00 0
7f6e25d9d000-7f6e25df4000 r--p 00000000 08:20 135110                     /usr/lib/locale/C.utf8/LC_CTYPE
7f6e25df4000-7f6e25df5000 r--p 00000000 08:20 135617                     /usr/lib/locale/C.utf8/LC_NUMERIC
7f6e25df5000-7f6e25df6000 r--p 00000000 08:20 136387                     /usr/lib/locale/C.utf8/LC_TIME
7f6e25df6000-7f6e25df7000 r--p 00000000 08:20 135104                     /usr/lib/locale/C.utf8/LC_COLLATE
7f6e25df7000-7f6e25df8000 r--p 00000000 08:20 135592                     /usr/lib/locale/C.utf8/LC_MONETARY
7f6e25df8000-7f6e25df9000 r--p 00000000 08:20 135520                     /usr/lib/locale/C.utf8/LC_MESSAGES/SYS_LC_MESSAGES
7f6e25df9000-7f6e25dfa000 r--p 00000000 08:20 136376                     /usr/lib/locale/C.utf8/LC_PAPER
7f6e25dfa000-7f6e260e3000 r--p 00000000 08:20 136432                     /usr/lib/locale/locale-archive
7f6e260e3000-7f6e260e6000 rw-p 00000000 00:00 0
7f6e260e6000-7f6e2610e000 r--p 00000000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e2610e000-7f6e262a3000 r-xp 00028000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e262a3000-7f6e262fb000 r--p 001bd000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e262fb000-7f6e262fc000 ---p 00215000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e262fc000-7f6e26300000 r--p 00215000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e26300000-7f6e26302000 rw-p 00219000 08:20 1121                       /usr/lib/x86_64-linux-gnu/libc.so.6
7f6e26302000-7f6e2630f000 rw-p 00000000 00:00 0
7f6e2630f000-7f6e26310000 r--p 00000000 08:20 135616                     /usr/lib/locale/C.utf8/LC_NAME
7f6e26310000-7f6e26311000 r--p 00000000 08:20 135103                     /usr/lib/locale/C.utf8/LC_ADDRESS
7f6e26311000-7f6e26312000 r--p 00000000 08:20 136378                     /usr/lib/locale/C.utf8/LC_TELEPHONE
7f6e26312000-7f6e26313000 r--p 00000000 08:20 135112                     /usr/lib/locale/C.utf8/LC_MEASUREMENT
7f6e26313000-7f6e2631a000 r--s 00000000 08:20 151853                     /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7f6e2631a000-7f6e2631c000 rw-p 00000000 00:00 0
7f6e2631c000-7f6e2631e000 r--p 00000000 08:20 1259                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f6e2631e000-7f6e26348000 r-xp 00002000 08:20 1259                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f6e26348000-7f6e26353000 r--p 0002c000 08:20 1259                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f6e26353000-7f6e26354000 r--p 00000000 08:20 135111                     /usr/lib/locale/C.utf8/LC_IDENTIFICATION
7f6e26354000-7f6e26356000 r--p 00037000 08:20 1259                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f6e26356000-7f6e26358000 rw-p 00039000 08:20 1259                       /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd16103000-7ffd16124000 rw-p 00000000 00:00 0                          [stack]
7ffd16155000-7ffd16159000 r--p 00000000 00:00 0                          [vvar]
7ffd16159000-7ffd1615b000 r-xp 00000000 00:00 0                          [vdso]
```

# Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection

# Stack buffer-overflow: Example

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

#define NAME_LEN 32

void copy_name(char *src) {
    char name[NAME_LEN];
    strcpy(name, src);
    printf("My name is %s\n", name);
    return;
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return -1;

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

```
└$ ./bof byoungyoung
My name is byoungyoung
```

```
└$ ./bof byoungyoungbyoungyoungbyoungyoungbyoungyoung
My name is byoungyoungbyoungyoungbyoungyoungbyoungyoung
[1]    29364 segmentation fault  ./bof byoungyoungbyoungyoungbyoungyoungbyoungyoung
```

# Such Problems are a BIG Deal

- **Generally called a "buffer overflow"**
  - When exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the major technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack

**Reference:**
**http://www.aquamanager.com**

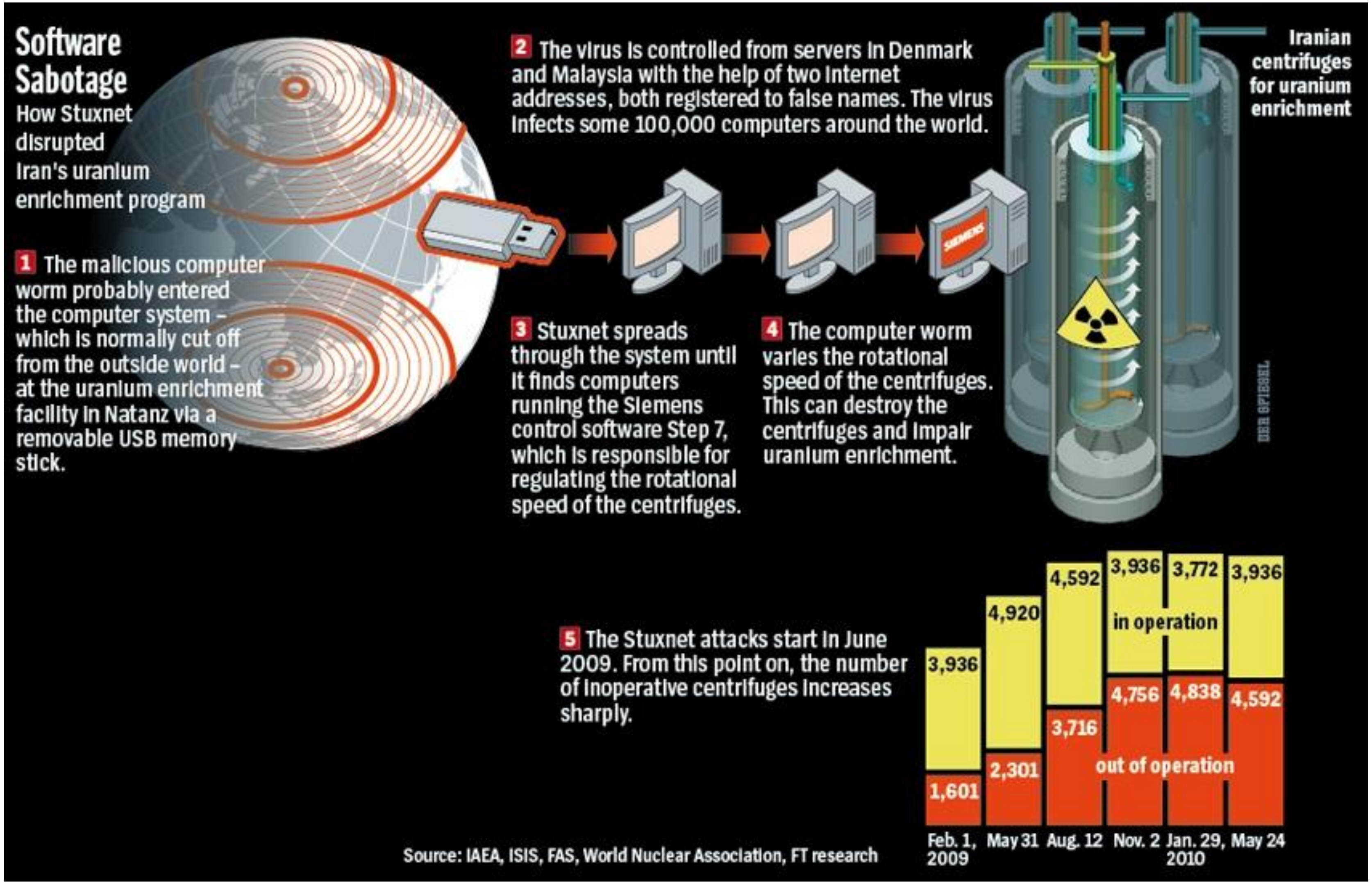
# Exploits Based on Buffer Overflows

- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- **Surprisingly common in real programs**
  - Programmers keep making the same mistakes ☹
  - Recent mitigation techniques make these attacks much more difficult
- **Examples across the decades**
  - Original "Morris worm" (1988)
  - Code Red worm (2001)
  - Stuxnet (2005~2010)
  - Heartbleed (2012~2014)
  - … and many, many more
    - Most of Chrome/Firefox/Safari exploits
    - Most iOS Jailbreak, Android rooting
- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!

# Example: the original Morris worm (1988)

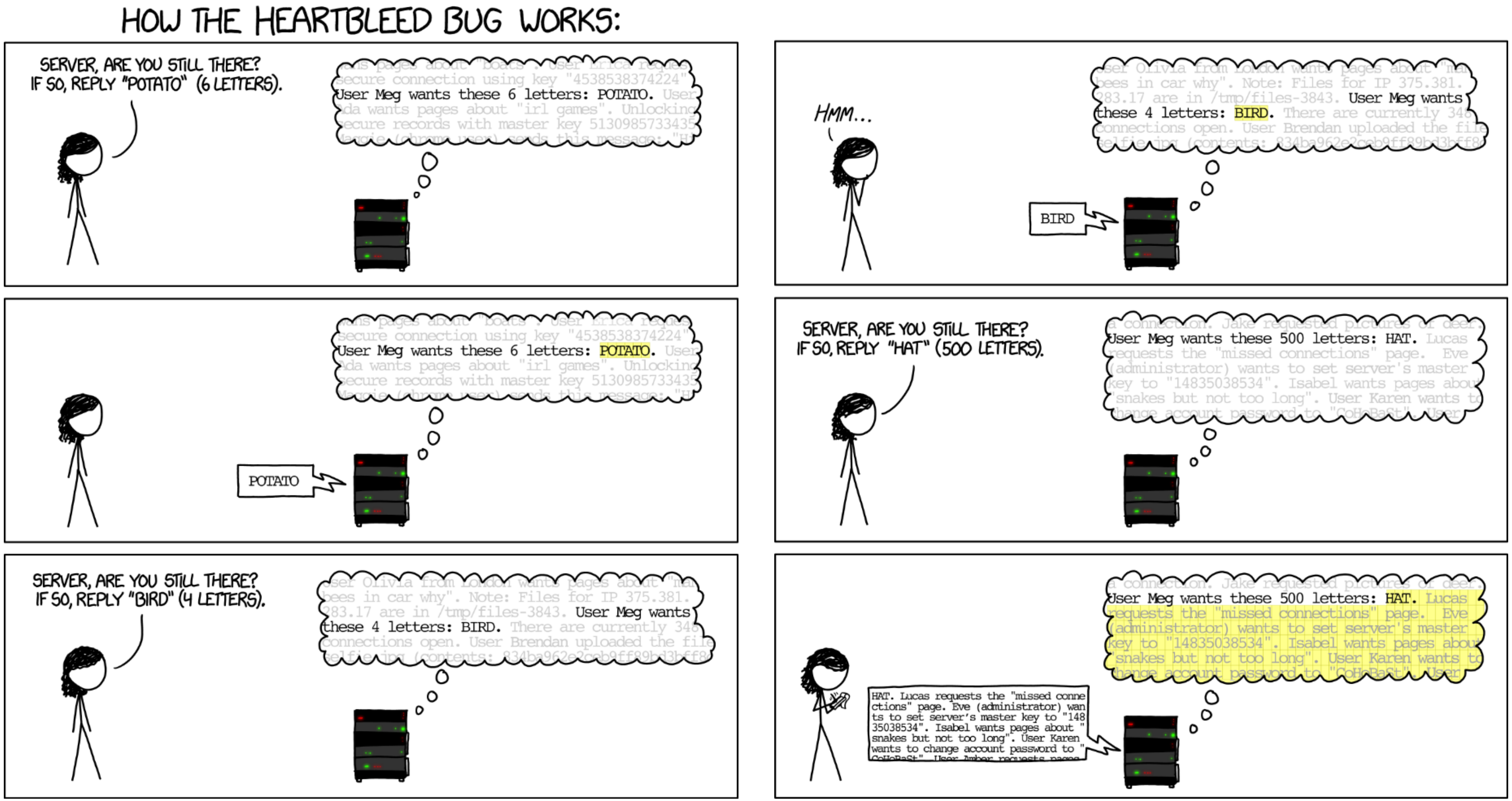
- **Exploited a few vulnerabilities to spread**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger byoungyoung@snu.ac.kr`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger`** ***`"exploit-code  padding  new-return-address"`***
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet ☺ )
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted, and then…
    - https://en.wikipedia.org/wiki/Robert_Tappan_Morris

# Stuxnet




Reference: https://www.extremetech.com

# Heartbleed




https://xkcd.com/1354/

# Let’s go back to the example

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

#define NAME_LEN 32

void copy_name(char *src) {
    char name[NAME_LEN];
    strcpy(name, src);
    printf("My name is %s\n", name);
    return;
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return -1;

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

```
└─$ ./bof byoungyoung
My name is byoungyoung
```

```
└─$ ./bof byoungyoungbyoungyoungbyoungyoungbyoungyoung
My name is byoungyoungbyoungyoungbyoungyoungbyoungyoung
[1]    29364 segmentation fault  ./bof byoungyoungbyoungyoungbyoungyoungbyoungyoung
```

# When does it start complaining?

```
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof a
My name is a
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aa
My name is aa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aaa
My name is aaa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aaaaa
My name is aaaaa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aaaaaaaa
My name is aaaaaaaa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
My name is aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[1]    7391 segmentation fault  ./bof aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$ ./bof aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
My name is aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[1]    7467 segmentation fault  ./bof aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
┌blee@DESKTOP-TBSBE9P ~/class/class-systems-programming/random-stuffs/buffer-overflow ‹main›
└$
```

**This is not quite smart….**

# Being smarter with pwntools

```
import pwn

for i in range(64):
    print()
    print("Trying len %d" % i)
    pwn.process(["./bof", "a" * i]).recvall()
```

```
└$ ./check-break-len.py

Trying len 0
[+] Starting local process './bof': pid 11643
[+] Receiving all data: Done (12B)
[*] Process './bof' stopped with exit code 0 (pid 11643)

Trying len 1
[+] Starting local process './bof': pid 11646
[+] Receiving all data: Done (13B)
[*] Process './bof' stopped with exit code 0 (pid 11646)
```

```
Trying len 31
[+] Starting local process './bof': pid 11781
[+] Receiving all data: Done (43B)
[*] Process './bof' stopped with exit code 0 (pid 11781)

Trying len 32
[+] Starting local process './bof': pid 11787
[+] Receiving all data: Done (44B)
[*] Process './bof' stopped with exit code -11 (SIGSEGV) (pid 11787)

Trying len 33
[+] Starting local process './bof': pid 11790
[+] Receiving all data: Done (45B)
[*] Process './bof' stopped with exit code -11 (SIGSEGV) (pid 11790)
```

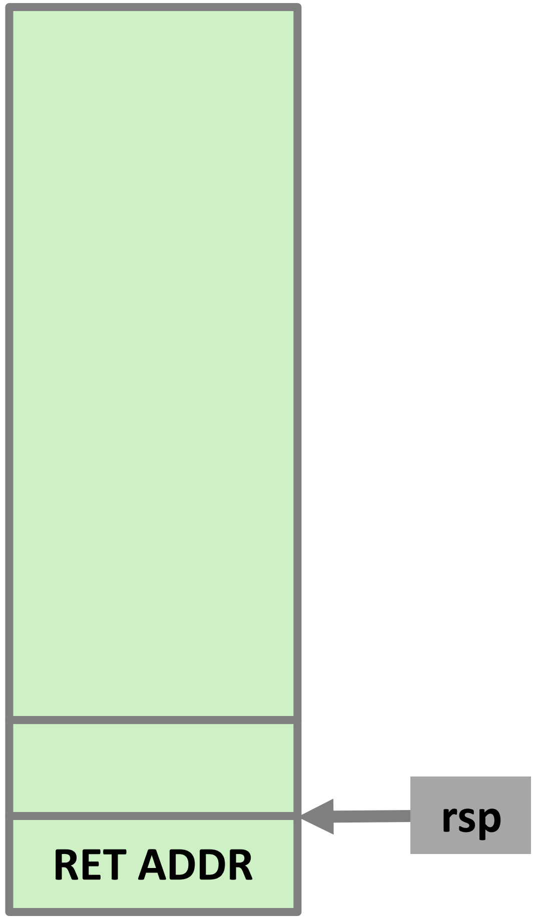
**It starts complaining if the length is 32 or more.**

# Can you be more precise when it starts breaking?

```
0000000000001245 <copy_name>:
    1245:       55                      push   %rbp
    1246:       48 89 e5                mov    %rsp,%rbp
    1249:       48 83 ec 30             sub    $0x30,%rsp
    124d:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
    1251:       48 8b 55 d8             mov    -0x28(%rbp),%rdx
    1255:       48 8d 45 e0             lea    -0x20(%rbp),%rax
    1259:       48 89 d6                mov    %rdx,%rsi
    125c:       48 89 c7                mov    %rax,%rdi
    125f:       e8 cc fd ff ff          call   1030 <strcpy@plt>
    1264:       48 8d 45 e0             lea    -0x20(%rbp),%rax
    1268:       48 89 c6                mov    %rax,%rsi
    126b:       48 8d 05 b3 0d 00 00    lea    0xdb3(%rip),%rax
    1272:       48 89 c7                mov    %rax,%rdi
    1275:       b8 00 00 00 00          mov    $0x0,%eax
    127a:       e8 d1 fd ff ff          call   1050 <printf@plt>
    127f:       90                      nop
    1280:       c9                      leave
    1281:       c3                      ret
```

- **The assembly of copy_name() should have an answer!**
- **Let’s read assembly …**

# Why does it break at 32?




stack

main()

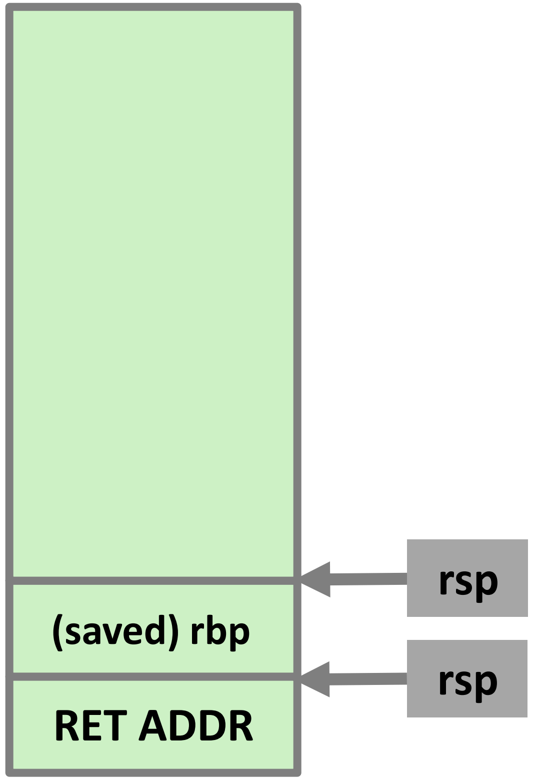
```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
cmpl   $0x1,-0x4(%rbp)
jg     0x129e <main+28>
mov    $0xffffffff,%eax
jmp    0x12b6 <main+52>
mov    -0x10(%rbp),%rax
add    $0x8,%rax
mov    (%rax),%rax
mov    %rax,%rdi
call   0x1245 <copy_name>
mov    $0x0,%eax
leave
ret
```

copy_name()

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x28(%rbp)
mov    -0x28(%rbp),%rdx
lea    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
call   0x1030 <strcpy@plt>
lea    -0x20(%rbp),%rax
mov    %rax,%rsi
lea    0xdb3(%rip),%rax
mov    %rax,%rdi
mov    $0x0,%eax
call   0x1050 <printf@plt>
nop
leave
ret
```

- call instruction pushes the return address
  - The address of the call's next instruction
- rdi holds the first parameter of copy_name() (i.e., `char *src`)

# Why does it break at 32?



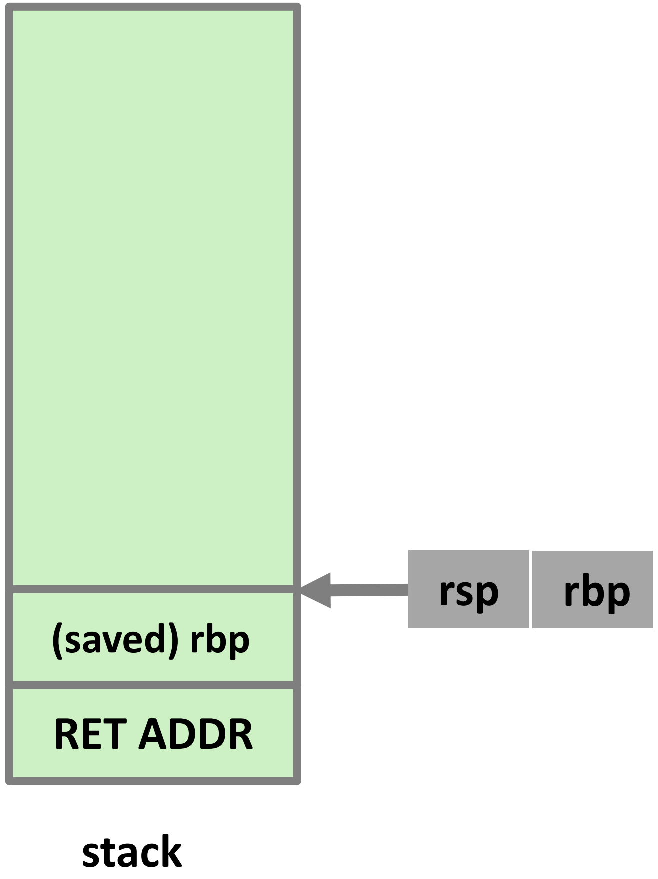

stack

**copy_name()**

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x28(%rbp)
mov    -0x28(%rbp),%rdx
lea    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
call   0x1030 <strcpy@plt>
lea    -0x20(%rbp),%rax
mov    %rax,%rsi
lea    0xdb3(%rip),%rax
mov    %rax,%rdi
mov    $0x0,%eax
call   0x1050 <printf@plt>
nop
leave
ret
```

- `push rbp` is part of the function prolog.
- It saves the stack frame pointer (i.e., `rbp`) of the caller (which is `main()`)
- This saved stack frame pointer will be restored later when executing `leave`.

# Why does it break at 32?

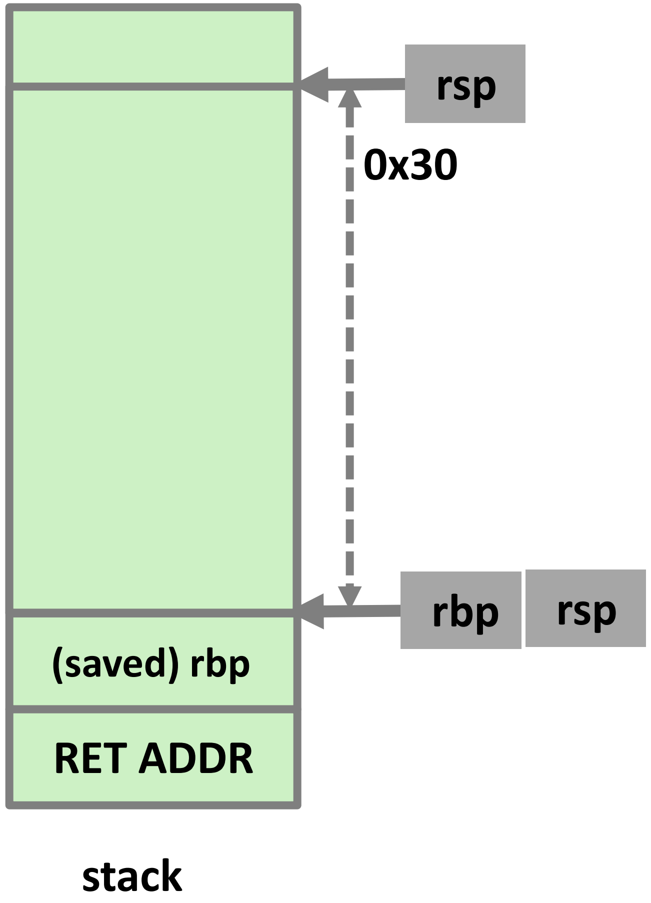



copy_name()

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x28(%rbp)
mov    -0x28(%rbp),%rdx
lea    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
call   0x1030 <strcpy@plt>
lea    -0x20(%rbp),%rax
mov    %rax,%rsi
lea    0xdb3(%rip),%rax
mov    %rax,%rdi
mov    $0x0,%eax
call   0x1050 <printf@plt>
nop
leave
ret
```

- `mov rbp, rsp` is also part of the function prolog.
- This updates the stack frame pointer
  - such that `rbp` accordingly points to the stack frame pointer of `copy_name()`
  - which previously pointed to the tack frame pointer of `main()`

# Why does it break at 32?

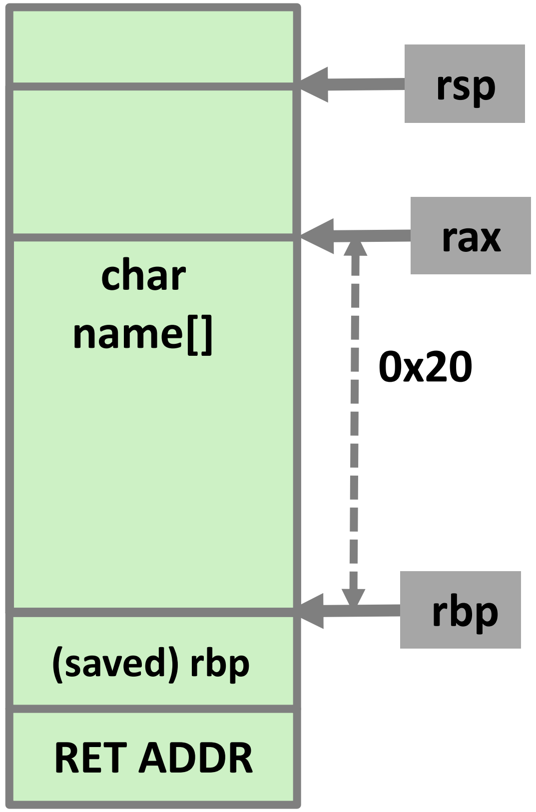



copy_name()

```
push   %rbp
mov    %rsp,%rbp
sub    $0x30,%rsp
mov    %rdi,-0x28(%rbp)
mov    -0x28(%rbp),%rdx
lea    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
call   0x1030 <strcpy@plt>
lea    -0x20(%rbp),%rax
mov    %rax,%rsi
lea    0xdb3(%rip),%rax
mov    %rax,%rdi
mov    $0x0,%eax
call   0x1050 <printf@plt>
nop
leave
ret
```

- This subtraction w.r.t. `rsp` is allocating the local space for copy_name()

# Why does it break at 32?



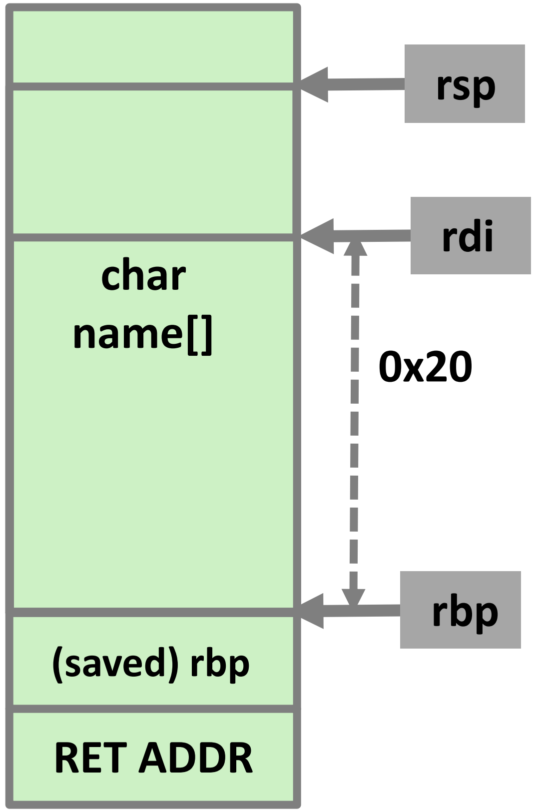

stack

copy_name()

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
call    0x1030 <strcpy@plt>
lea     -0x20(%rbp),%rax
mov     %rax,%rsi
lea     0xdb3(%rip),%rax
mov     %rax,%rdi
mov     $0x0,%eax
call    0x1050 <printf@plt>
nop
leave
ret
```

- Using `lea`, the base address of `char name[]` is stored in rax
  - rax == `rbp-0x20`.

# Why does it break at 32?




stack

copy_name()

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
call    0x1030 <strcpy@plt>
lea     -0x20(%rbp),%rax
mov     %rax,%rsi
lea     0xdb3(%rip),%rax
mov     %rax,%rdi
mov     $0x0,%eax
call    0x1050 <printf@plt>
nop
leave
ret
```

Do you see now why the program starts complaining when the string size is 32?

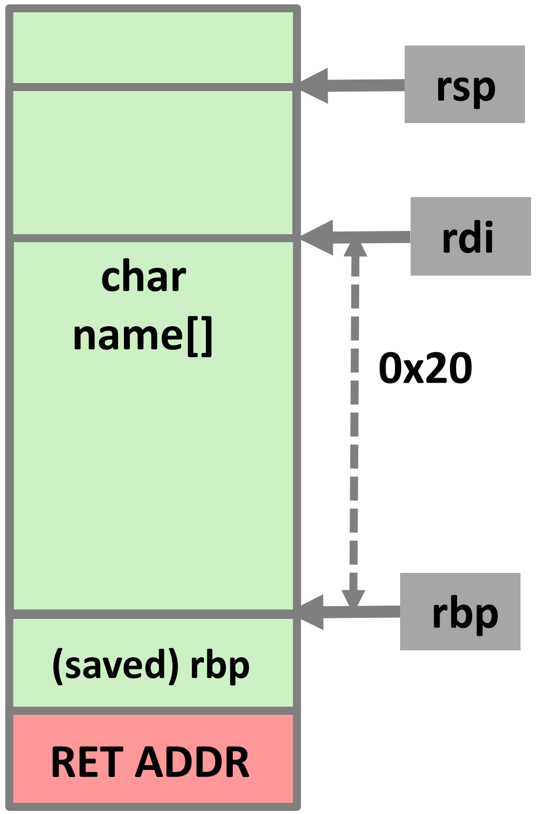
# Exploiting Stack Buffer Overflows

- **Overwriting the return address, you can control "RIP"**
  - Means you can control "where to execute"

- **But how would you execute your own malicious code?**
  - (1) Jump to the existing (malicious) code in the victim program
  - (2) Inject the malicious code
  - (3) return-oriented-programming

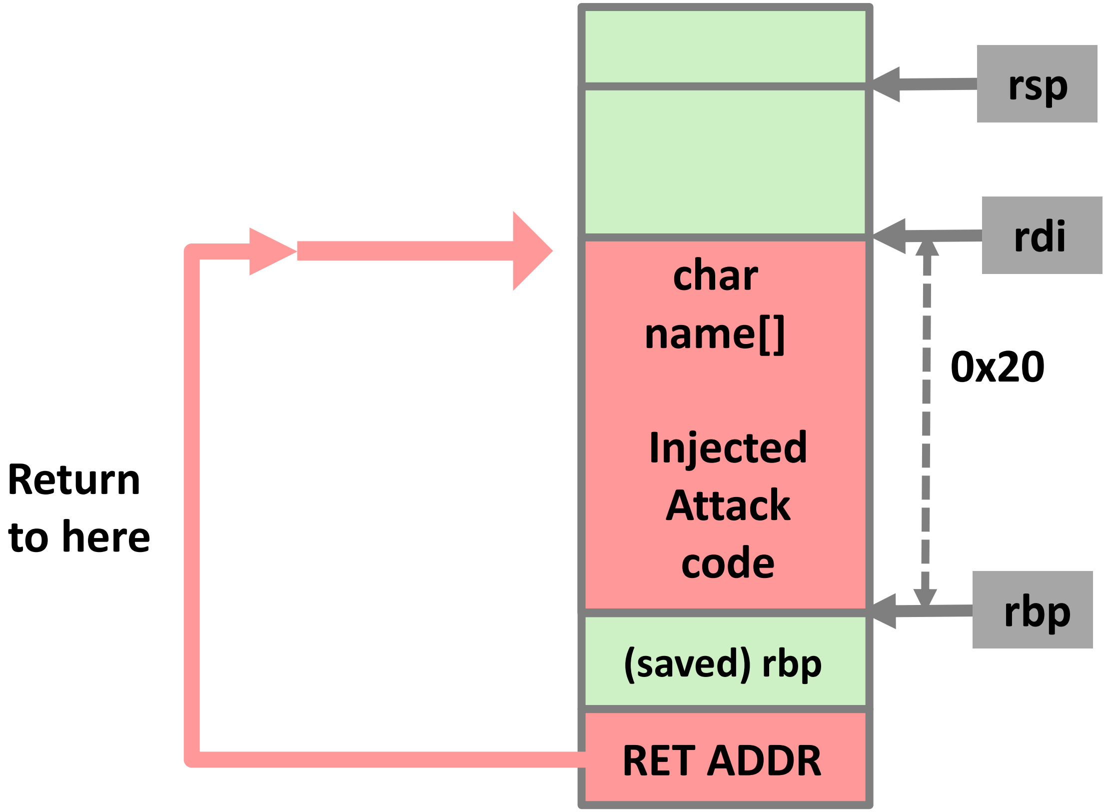
# Buffer Overflow Attacks

```
void print_passwd(void) {
    char c;
    FILE *f;

    f = fopen("passwd.txt", "r");
    if (!f)
        exit(-1);

    write(1, "[**] Password is ", strlen("[**] Password is "));

    while ((c = fgetc(f)) != EOF) {
        write(1, &c, 1);
    }
    write(1, "\n", 1);
    fflush(stdout);
    return;
}

void copy_name(char *src) {
    char name[NAME_LEN];

    strcpy(name, src);

    printf("My name is %s\n", name);
    return;
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return -1;

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




- Overwrite normal return address of copy_name() with the address of some other code!
- When `copy_name returns`, it will jump to the other code (i.e., print_passwd())

# Code Injection Attacks




- Input string contains byte representation of executable code
- Overwrite the return address copy_name() with the address of the name buffer
- When `copy_name` returns, it will jump to the exploit code

# What to Do About Buffer Overflow Attacks

- **Avoid overflow vulnerabilities**
- **Employ system-level protections**
- **Have compiler use “stack canaries”**
- **Lets talk about each…**

# 1. Avoid Overflow Vulnerabilities in Code (!)

- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
  - Secure coding practice!

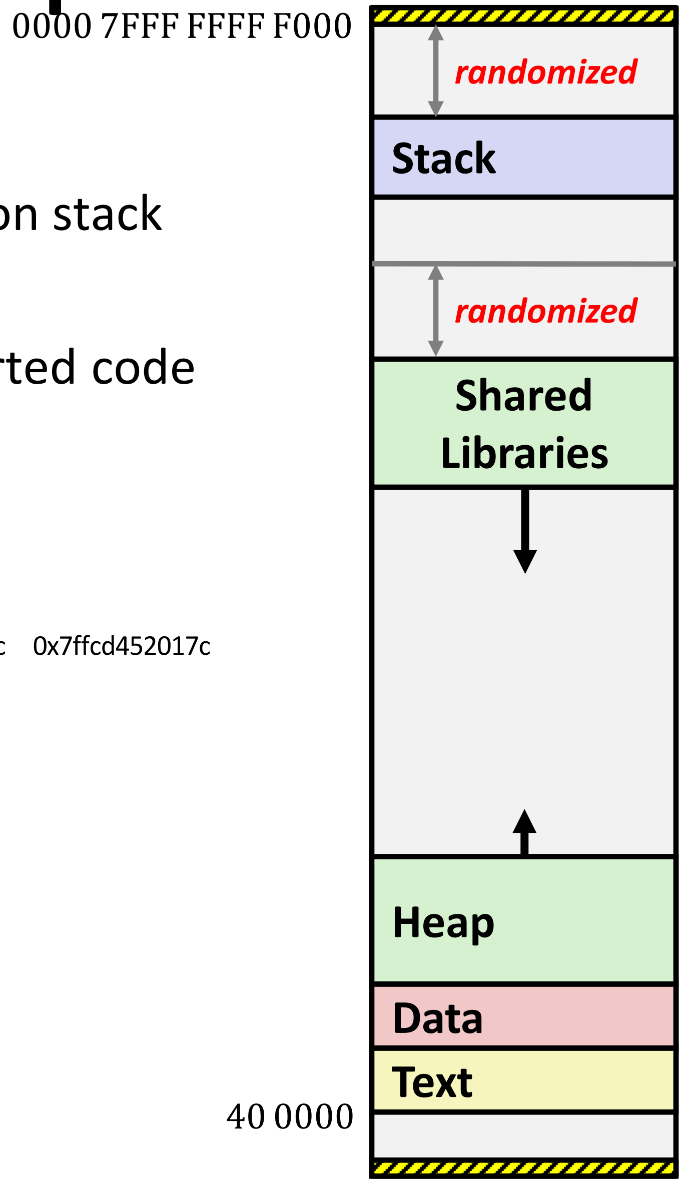
```
/* Echo Line */
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

# 2. System-Level Protections Can Help

- **Randomized stack offsets**
  - At start of program, **allocate random amount of space** on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - e.g., 5 executions of memory allocation code
  - Stack is repositioned each time program executes

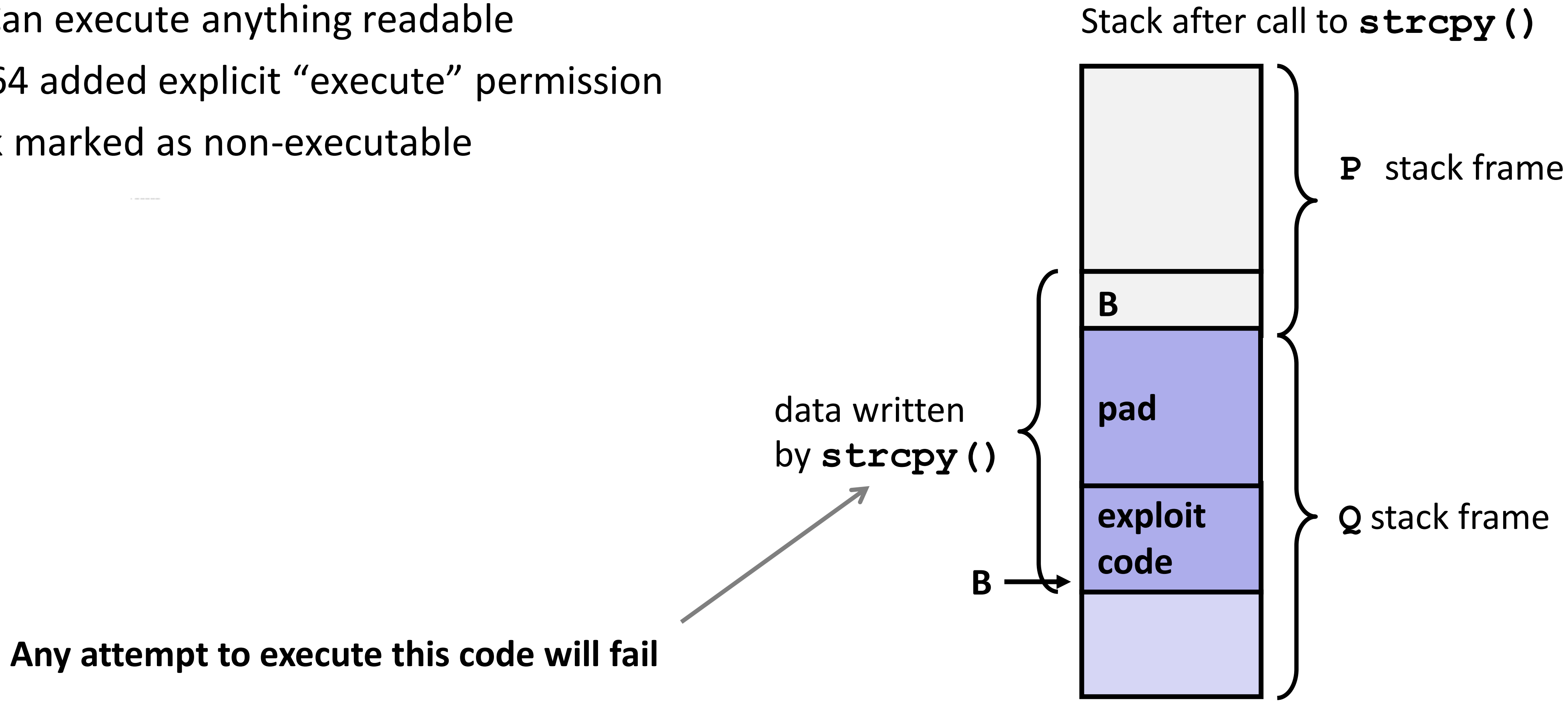
| local | 0x7ffe4d3be87c | 0x7fff75a4f9fc | 0x7ffeadb7c80c | 0x7ffeaea2fdac | 0x7ffcd452017c |
|---|---|---|---|---|---|

  - Address Space Layout Randomization (ASLR)



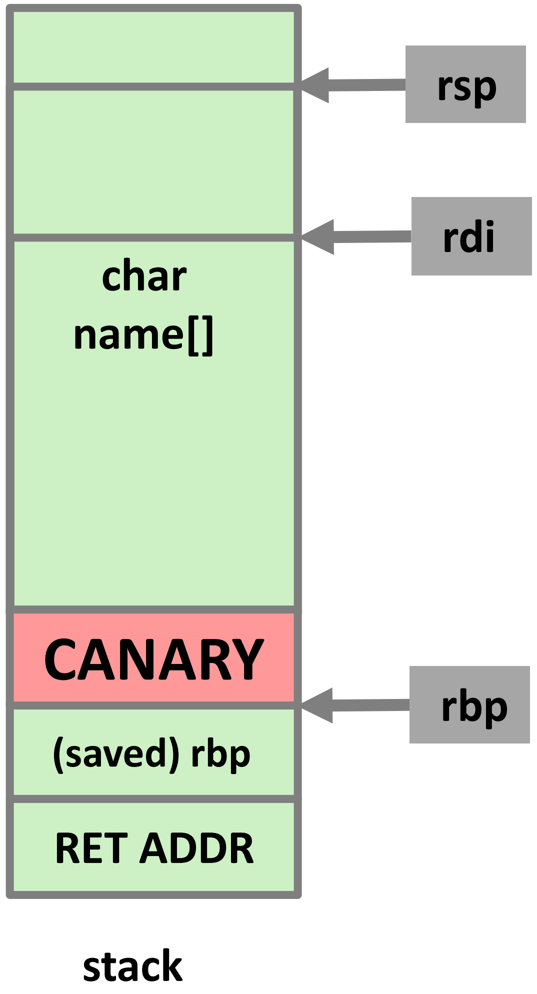
# 2. System-Level Protections Can Help

- **Non-executable code segments**
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - x86-64 added explicit “execute” permission
  - Stack marked as non-executable



# 3. Stack Canaries Can Help

- **Idea**
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default



# 3. Stack Canaries Can Help

**copy_name() : before**

```
push    rbp
mov     rbp,rsp
sub     rsp,0x30
mov     QWORD PTR [rbp-0x28],rdi
mov     rdx,QWORD PTR [rbp-0x28]
lea     rax,[rbp-0x20]
mov     rsi,rdx
mov     rdi,rax
call    0x1030 <strcpy@plt>
lea     rax,[rbp-0x20]
mov     rsi,rax
lea     rax,[rip+0xdb3]        # 0x2025
mov     rdi,rax
mov     eax,0x0
call    0x1050 <printf@plt>
nop
leave
ret
```

**copy_name() : after**

```
push    rbp
mov     rbp,rsp
sub     rsp,0x40
mov     QWORD PTR [rbp-0x38],rdi
mov     rax,QWORD PTR fs:0x28
mov     QWORD PTR [rbp-0x8],rax
xor     eax,eax
mov     rdx,QWORD PTR [rbp-0x38]
lea     rax,[rbp-0x30]
mov     rsi,rdx
mov     rdi,rax
call    0x1030 <strcpy@plt>
lea     rax,[rbp-0x30]
mov     rsi,rax
lea     rax,[rip+0xd94]        # 0x2025
mov     rdi,rax
mov     eax,0x0
call    0x1060 <printf@plt>
nop
mov     rax,QWORD PTR [rbp-0x8]
sub     rax,QWORD PTR fs:0x28
je      0x12b3 <copy_name+94>
call    0x1050 <__stack_chk_fail@plt>
leave
ret
```

- %fs:0x28 is a read-only storage, storing a global canary.
- The global canary is initialized with a random value when the program is loaded.

# Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Marking stack nonexecutable makes it hard to insert binary code

- **Alternative Strategy**
  - Use existing code
    - e.g., library code from stdlib (called “return-to-libc”)
  - Chain those fragments to achieve overall desired outcome

- **Construct “attack logic” from *gadgets***
  - Gadget: any sequence of instructions ending in `ret`
    - `ret`: an instruction encoded by single byte `0xc3`

# Return-oriented-programming (ROP)

- Generalized, a way more powerful version of return-to-libc
- Gadget
  - A sequence of instructions embedded in a victim program
  - Ends with a **return** instruction
  - Each gadget emulates a specific primitive operation
    - e.g., add, mul, mov, jmp, etc.
- ROP
  - Connect multiple gadgets together to perform arbitrary operations

# ROP Example #1 (simple)

- Goal: **Store a constant value `C` to a memory address `A`**
  - How would you setup the registers and stack?
- Given the CPU context
  - `*` denotes the register value that the attacker can control

| Register | Value |
|---|---|
| eip | * |
| esp | 0xbfff0000 |
| eax | * |
| ebx | * |

- Given gadgets

```
G1:
 mov (%eax), %ebx
 ret
```

# ROP Example #2 (chain)

- Goal: **Store a constant `C` to a memory address `A`**
  - How would you setup the registers and stack?
- Given CPU context

| Register | Value |
|---|---|
| eip | * |
| esp | 0xbfff0000 |
| eax | 0 |
| ebx | 0 |

Given gadgets

```
G1                       |G2:                  |G3:
  mov (%eax),            | mov   %eax, A       | mov %ebx, C
  %ebx ret               | ret                 | ret
```

# Summary

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return Oriented Programming