Buffers 1 - Stack

JOÃO PAULO BARRACA

Buffer Overflow - According to CAPEC-100

- **Targets improper or missing bounds checking on buffer operations**
  - typically triggered by input injected by an adversary.
- **An adversary is able to write past the boundaries of allocated buffer regions in memory**
- **Causes a program crash or potentially redirection of execution as per the adversaries' choice.**
  - Denial of Service
  - (Remote) Code Execution

Buffer Overflow - Scope

- **CWE-119 is extremely broad as there are many types of BO**

- **Characteristics of a BO**
  - Type of access: Read or Write
  - Type of memory: stack, heap
  - Location: before or after the buffer
  - Reason: iteration, copy, pointer arithmetic, memory clear, mapping

Other Direct Child CWEs

CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-125	Out-of-bounds Read
CWE-466	Return of Pointer Value Outside of Expected Range
CWE-786	Access of Memory Location Before Start of Buffer
CWE-787	Out-of-bounds Write
CWE-788	Access of Memory Location After End of Buffer
CWE-805	Buffer Access with Incorrect Length Value
CWE-822	Untrusted Pointer Dereference
CWE-823	Use of Out-of-range Pointer Offset
CWE-824	Access of Uninitialized Pointer
CWE-825	Expired Pointer Dereference

Relevant CWEs with specific types

CWE-120: Classic Buffer Overflow: copy without checking the size of the input

CWE-121: Stack-based Buffer Overflow: overwrite over data in the Stack Segment

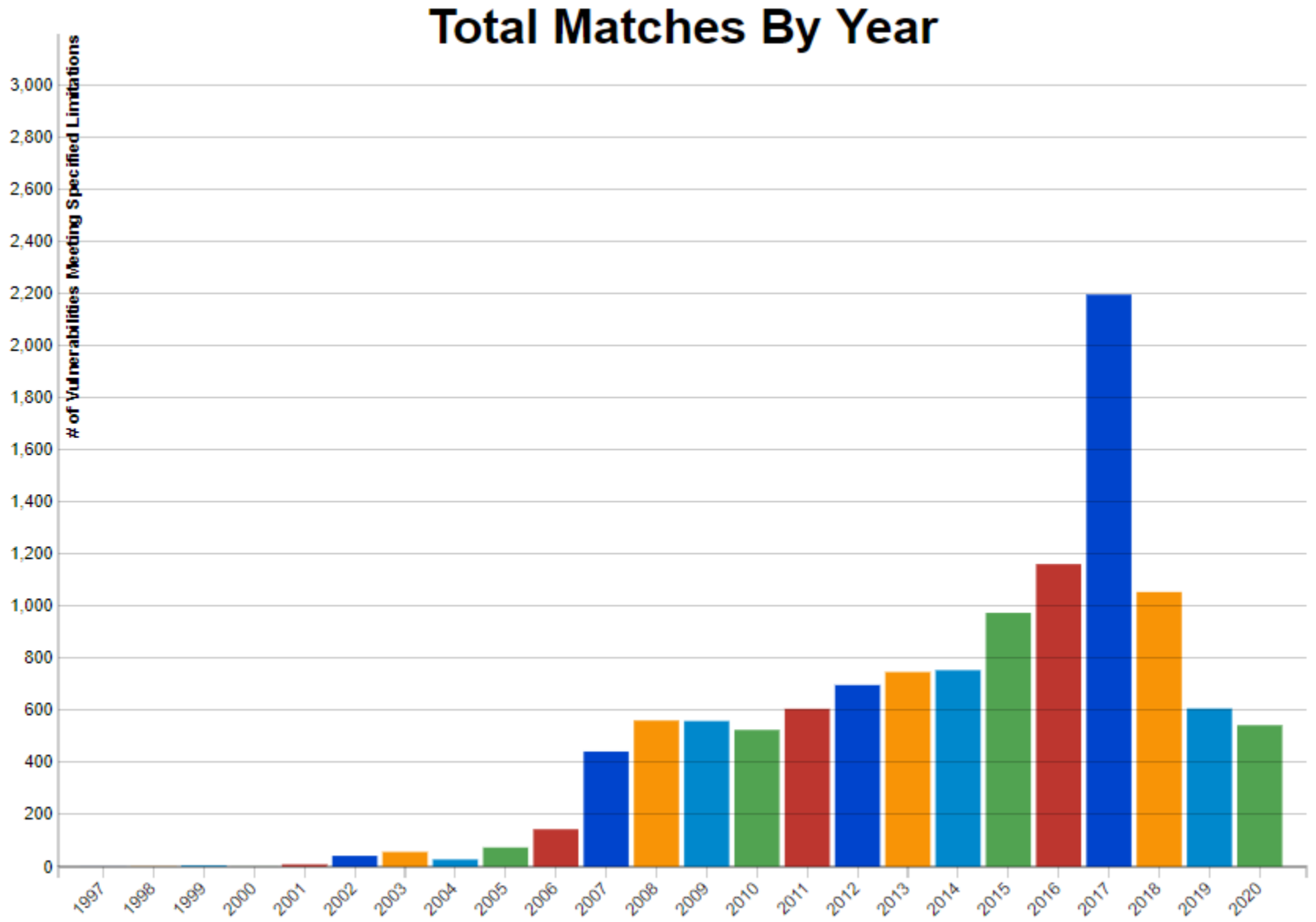
CWE-122: Heap-based Buffer Overflow: overwrite over data in the Heap Segment

CWE-123: Write-what-where Condition: ability to write to any memory of choice

CWE-124: Buffer Underwrite ('Buffer Underflow'): Write to memory before the buffer

CWE-126: Buffer Over-read: Read after the buffer ends (e.g., using an index)

CWE-127: Buffer Under-read: Read before the buffer start (e.g., using an index)



Popularity at NVD

Popularity decline

- **Better tools to check for the vulnerability**
  - Static/Dynamic Code analysis
- **Dissemination of bound checking mechanisms in compilers**
  - Standard in most distributions and enabled by default
  - Still lacking in embedded devices
- **Increasingly higher adoption of higher layer languages**
  - Extensive use and Open Sources libraries improves security
  - Security focused languages such as Rust

Potentially Vulnerable Software

- **Any software that gets information from external sources**
  - Sockets, PIPEs and other IPC
  - Files
  - Program arguments
  - Environment Variables
- **Software developed in languages with direct memory access**
  - Mostly C and C++ (or at least with most devastating impact)
  - But also: Go when using “unsafe”, PHP, Python, Java, etc…

Dominant prevalence

- **Anything that was made in a language with access to memory**
  - Server software packages (nginx, apache, mysql, …)

- **Embedded and IoT devices**
  - Due to lack of compiler support
  - Due to lack of hardware capabilities

… in python

```
$ cat bo_1.py

message = "Hello World"

buffer = [None] * 10


print(message)

for i in range(15):

        buffer[i] = 'A'


print(message)
```

```
$ python3 bo_1.py

Hello World

Traceback (most recent call last):

  File "bo_1.py", line 7, in <module>

    buffer[i] = 'A'

IndexError: list assignment index out of
range
```

… in C

```c
#include <stdio.h>

void main(int argc, char* argv[]){
    char message[] = "Hello World";
    int buffer[5];
    int i;

    printf("%s\n", message);
    for(i = 0;i < 15; i++) {
        buffer[i] = 'A';
    }
    printf("%s\n", message);
}
```

```
$ ./bo_1

Hello World

AAAAAAAAAAAAAAAd AAAAAAAAAd
```

Vulnerabilities in languages (mostly C/C++)

- **Not memory safe: programmers can read/write memory freely and are not constrained by the address or size of the variables**
  - Great flexibility, but huge risk as mistakes lead to accessing memory that otherwise should not be accessed
  - C/C++ compilers have freedom to optimize code and even sometimes undefined behavior

- **Memory safe languages intercept such errors, raising errors**
  - Program will crash (DoS), but impact is limited

```
// Correct usage
printf("%d\n", *value);

// Reading memory after the variable
printf("%d\n", *(value + 4));

// Reading memory before the variable
printf("%d\n", *(value - 4));
```

Vulnerabilities in languages (mostly C/C++)

- **Not memory safe: programmers can read/write memory freely and are not constrained by the address or size of the variables**
  - Great flexibility, but huge risk as mistakes lead to accessing memory that otherwise should not be accessed
  - C/C++ compilers have freedom to optimize code and even sometimes undefined behavior

- **Memory safe languages intercept such errors, raising errors**
  - Program will crash (Availability Compromised), but impact is limited

```
// Correct usage
printf("%d\n", *value);

// Reading memory after the variable
printf("%d\n", *(value + 4));

// Reading memory before the variable
printf("%d\n", *(value - 4));
```

```
$ ./not_memory_safe
42
0
32767
```

Vulnerabilities in languages (mostly C/C++)

- **Not type safe: memory content can be reinterpreted as required by the programmer**
  - Casts may be arbitrarily allowed and not checked
- **Type safe languages do not allow reinterpretation, or only safe reintrepertation**
  - Cast a byte to int is safe, a buffer to int is not.

```
int value = 42;

// Correct usage
printf("%d\n", value);

// Cast to variable with different storage
printf("%f\n", *((double*) &value));

// Cast to variable with different size
printf("%llu\n", *((unsigned long long*) &value));
```

Vulnerabilities in languages (mostly C/C++)

- **Not type safe: memory content can be reinterpreted as required by the programmer**
  - Casts may be arbitrarily allowed and not checked
- **Type safe languages do not allow reinterpretation, or only safe reinterpretation**
  - Cast a byte to int is safe, a buffer to int is not.

```
int value = 42;

// Correct usage
printf("%d\n", value);

// Cast to variable with different storage
printf("%f\n", *((double*) &value));

// Cast to variable with different size
printf("%llu\n", *((unsigned long long*) &value));
```

```
$ ./not_type_safe
42
0.000000
1170988679674462250
```

Vulnerabilities in languages (mostly C/C++)

- **Dynamically allocated memory has no implicit management mechanism**
  - Programmer must allocate and deallocate all memory
  - Programmer must know how memory was allocated
  - Programmer must free memory only after there is no other reference

```
char* buffer = (char*) malloc(10);
char* str = buffer;

free(buffer);

// Write after free (and write beyond buffer)
memcpy(str, "Hello World!!!!", 15);

// Read after free (and read beyond buffer)
printf("%s\n", str);
```

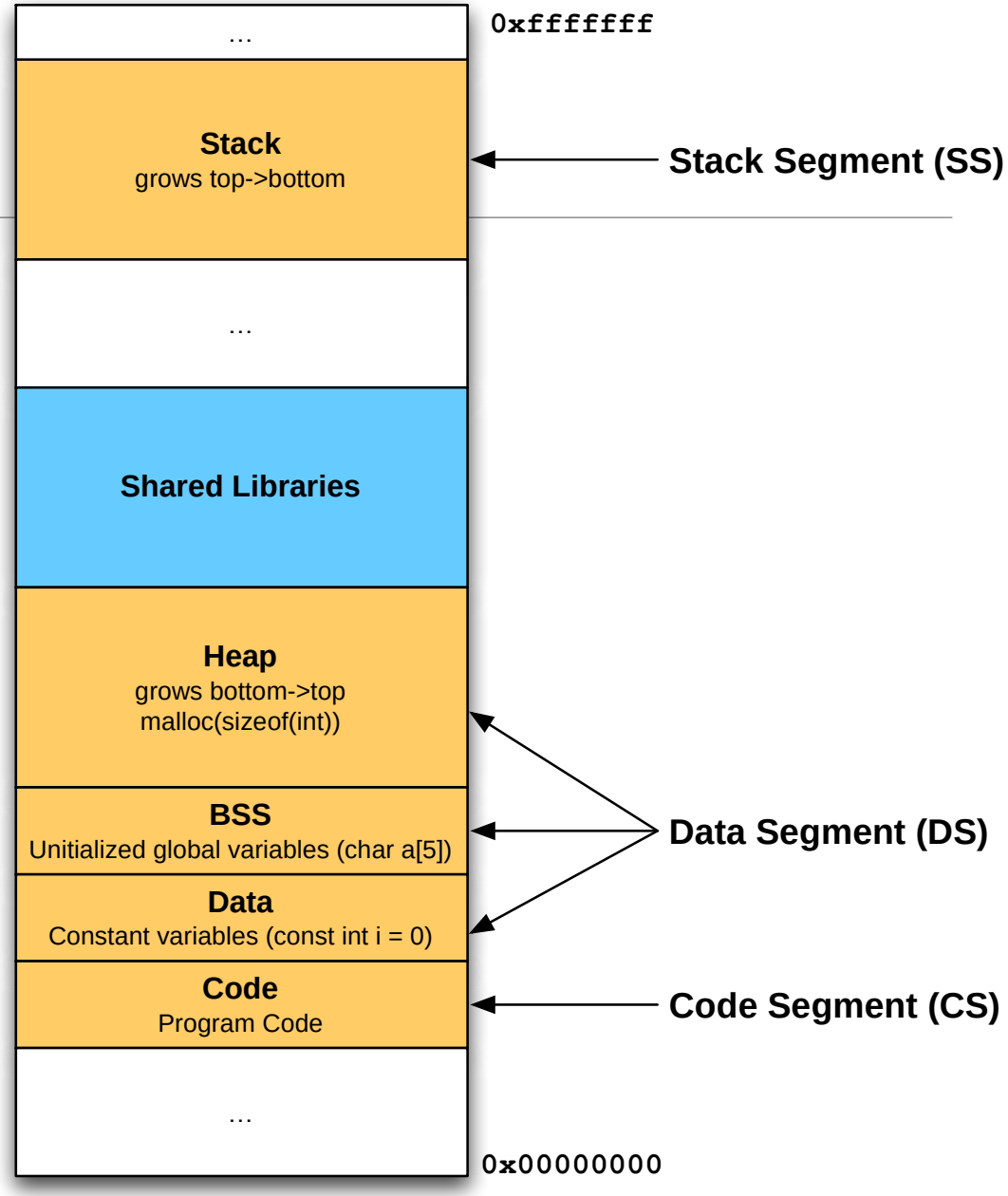
```
$ ./dynamic_memory
Hello World!!!!
```

Why? Memory Structure 101

- **Kernel organizes memory in pages**
  - Typically 4096 bytes
- **Processes operate in a Virtual Memory Space**
  - Mapped to real pages, which can be in RAM or Swapped
- **Kernel splits program in several segments**
  - Increases security
    - segment based permissions
  - Increases performance
    - some are dynamic: invalidated when program terminates
    - some are static: can be retained, speed repeated startup

Memory Structure

- **SS: Local variables and execution flow**
- **Shared Libraries: .so/dlls loaded.**
  - Addresses are shared between programs
- **Heap: memory allocated with malloc/new**
- **BSS: Global Variables**
- **Data: Constants**
- **Code: Actual instructions**

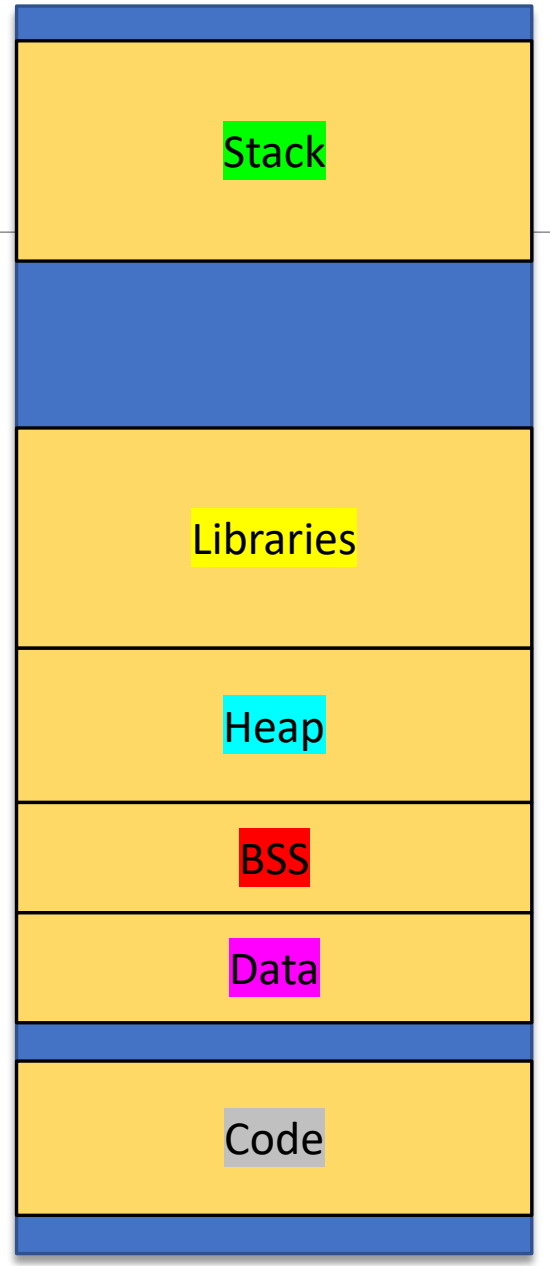


mem.c (available in course web page)

- **Simple program showing the memory map of itself**
- **Features:**
  - Prints the address of objects of different types
    - Argument
    - Dynamic memory with malloc
    - Global Variable
    - Constant
    - Function
  - Prints the memory maps as exposed in /proc/self/maps
  - Creates a recursive function and prints the address of local variables
  - Crashes with a Stack Overflow

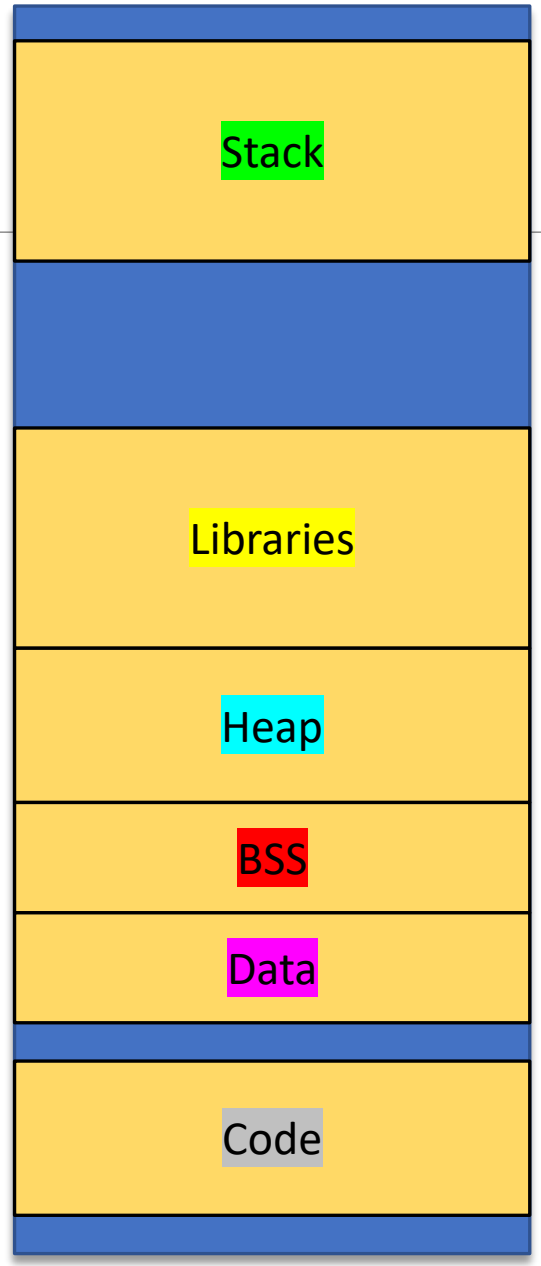
mem.c

```
Internal Variables (Page = 4096)

&argc  = bfeb8590 -> stack = bfeb8000

malloc = 08435008 -> heap  = 08435000

bssvar = 0804a034 -> bss   = 0804a000

cntvar = 08048920 -> const = 08048000

&main  = 0804865c -> text  = 08048000
```



mem.c

```
Content of /proc/self/maps
08048000-08049000 r-xp 00000000 08:01 26845750   /home/s/mem
08049000-0804a000 r--p 00000000 08:01 26845750   /home/s/mem
0804a000-0804b000 rw-p 00001000 08:01 26845750   /home/s/mem
08435000-08456000 rw-p 00000000 00:00 0          [heap]
b7616000-b7617000 rw-p 00000000 00:00 0
b7617000-b776a000 r-xp 00000000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776a000-b776b000 ---p 00153000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776b000-b776d000 r--p 00153000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776d000-b776e000 rw-p 00155000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776e000-b7771000 rw-p 00000000 00:00 0
b777e000-b7782000 rw-p 00000000 00:00 0
b7782000-b7783000 r-xp 00000000 00:00 0          [vdso]
b7783000-b779e000 r-xp 00000000 08:01 1565567    /lib/ld-2.11.1.so
b779e000-b779f000 r--p 0001a000 08:01 1565567    /lib/ld-2.11.1.so
b779f000-b77a0000 rw-p 0001b000 08:01 1565567    /lib/ld-2.11.1.so
bfe99000-bfeba000 rw-p 00000000 00:00 0          [stack]
```



0xFFFFFFFF
Top of memory

Start of Memory
0x00000000

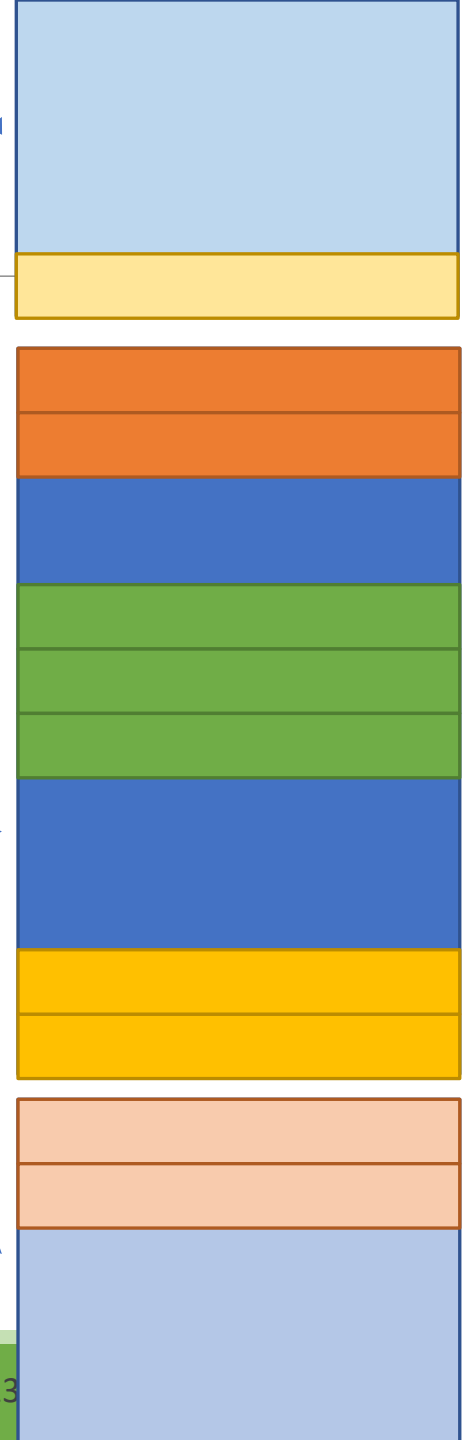
mem.c

```
Stack evolution:
foo [000]: &argc  = bfeb8140 -> stack = bfeb8000
foo [001]: &argc  = bfdb8110 -> stack = bfdb8000
foo [002]: &argc  = bfcb80e0 -> stack = bfcb8000
foo [003]: &argc  = bfbb80b0 -> stack = bfbb8000
foo [004]: &argc  = bfab8080 -> stack = bfab8000
foo [005]: &argc  = bf9b8050 -> stack = bf9b8000
foo [006]: &argc  = bf8b8020 -> stack = bf8b8000
foo [007]: &argc  = bf7b7ff0 -> stack = bf7b7000
foo [008]: &argc  = bf6b7fc0 -> stack = bf6b7000
Segmentation fault
```

Stack organization

- **Stack is organized by frames, one for each function call**
  - Memory reserved for the function to use as it requires

```
void main(){
   foo();
}

void foo(){
   bar();
}
```

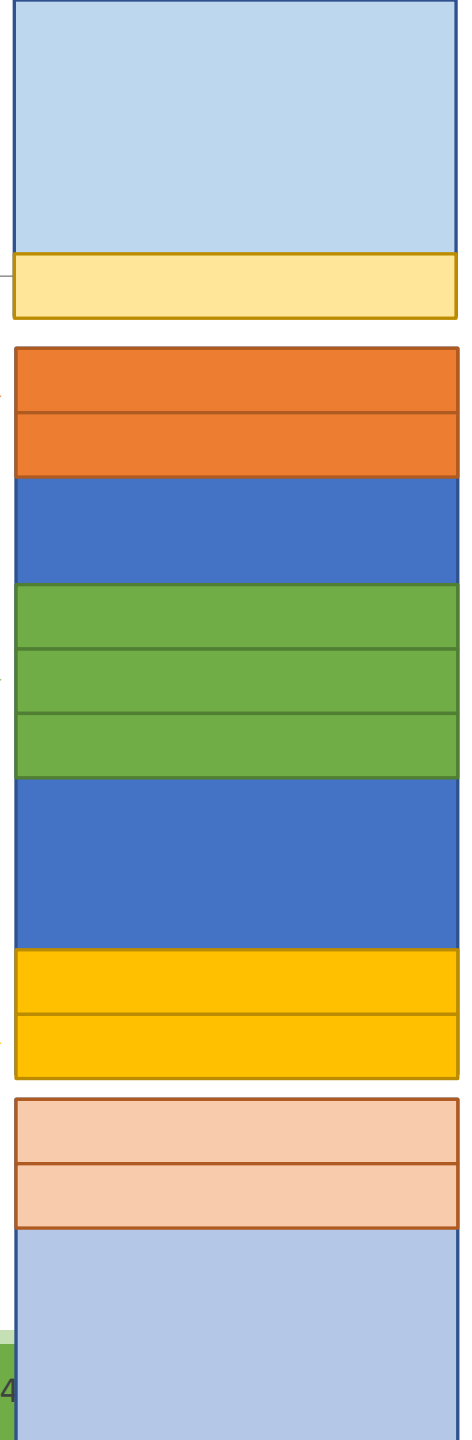


- **Each stack frame stores:**
  - Return Information
  - Local Variables
  - Arguments to following functions (x32: all, x64: +5th)

Stack organization

- **Stack is organized by frames, one for each function call**
  - Memory reserved for the function to use as it requires

- **Each stack frame stores:**
  - Return Information
  - Local Variables
  - Arguments to following functions

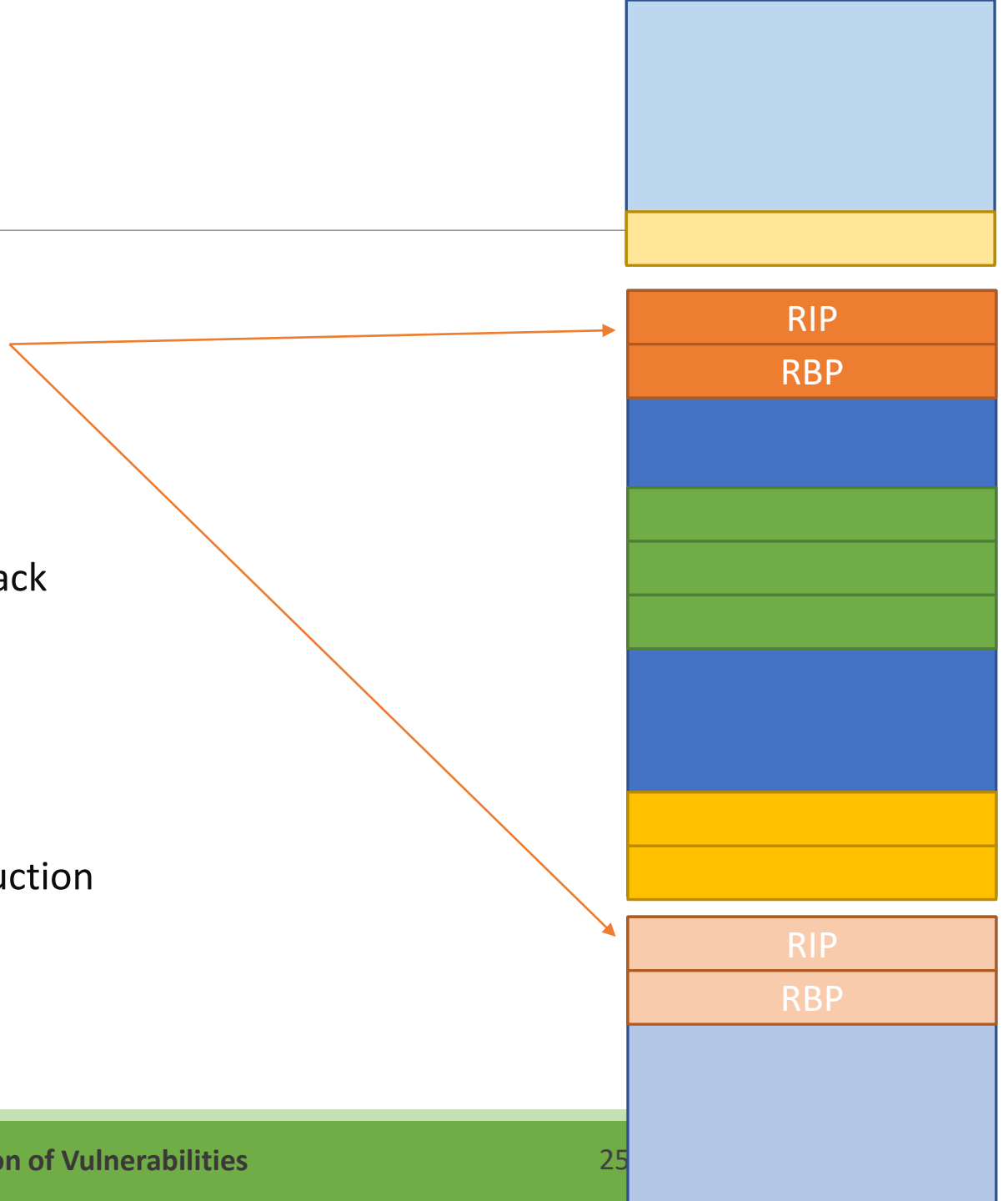


Stack organization

- **Return information has 2 major objectives**
  - Chaining frames as new functions are called
  - Return to the next instruction after the function ends

- **Frame chaining**
  - When a function is called, the address of the current stack frame (Register RBP in x64) is push to the frame
  - When the function ends, RBP is popped
    - Caller function has its frame restored

- **Function chaining**
  - When a function is called, the address of the next instruction is push to the stack (RIP register)
  - When a function ends, that address is popped
    - Execution resumes at the caller function



mem_local.c (available in course web page)

- **Prints the address to several variables**
  - Local variables declared in the main function
  - Arguments passed to the foo function
  - Local variables in the foo function

```
main
argc    : 0x7fffd6baeddc
argv    : 0x7fffd6baeed8

foo
a       : 0x7fffd6baed8c
local_a: 0x7fffd6baed9b
buffer : 0x7fffd6baeda0
local_b: 0x7fffd6baed9c
```

```c
char foo(int a,){
    char local_a = 3;
    char buffer[16];
    int local_b = 5;

    printf("%p\n", &a);
    printf("%p\n", &local_a);
    printf("%p\n", &buffer);
    printf("%p\n", &local_b);

    buffer[0] = local_a;
    return buffer[0];
}

int main(int argc, char* argv[]){
    printf("%p\n", &argc);
    printf("%p\n", argv);

    return foo(argc);
}
```

mem_local.c – Conclusions

- **Stack frame grows from higher addresses to lower addresses**
  - Main has variables at 0xbaedb.
  - Foo has variables at 0xbaed6-8.

```
main
argc    : 0x7fffd6baeddc
argv    : 0x7fffd6baeed8

foo
a       : 0x7fffd6baed8c
local_a: 0x7fffd6baed9b
buffer : 0x7fffd6baeda0
local_b: 0x7fffd6baed9c
```

```c
char foo(int a,){
    char local_a = 3;
    char buffer[16];
    int local_b = 5;

    printf("%p\n", &a);
    printf("%p\n", &local_a);
    printf("%p\n", &buffer);
    printf("%p\n", &local_b);

    buffer[0] = local_a;
    return buffer[0];
}

int main(int argc, char* argv[]){
    printf("%p\n", &argc);
    printf("%p\n", argv);

    return foo(argc);
}
```

mem_local.c – Conclusions

- **Declaration order doesn't matter!**
- **Compiler will place variables are he seems adequate**
  - Will keep information aligned
  - May create empty spaces
  - May deploy additional protection mechanisms (canaries)

```
main
argc    : 0x7fffd6baeddc
argv    : 0x7fffd6baeed8

foo
a       : 0x7fffd6baed8c
local_a: 0x7fffd6baed9b  (3rd)
buffer : 0x7fffd6baeda0  (1st)
local_b: 0x7fffd6baed9c  (2nd)
```

```c
char foo(int a,){
    char local_a = 3;
    char buffer[16];
    int local_b = 5;

    printf("%p\n", &a);
    printf("%p\n", &local_a);
    printf("%p\n", &buffer);
    printf("%p\n", &local_b);

    buffer[0] = local_a;
    return buffer[0];
}

int main(int argc, char* argv[]){
    printf("%p\n", &argc);
    printf("%p\n", argv);

    return foo(argc);
}
```

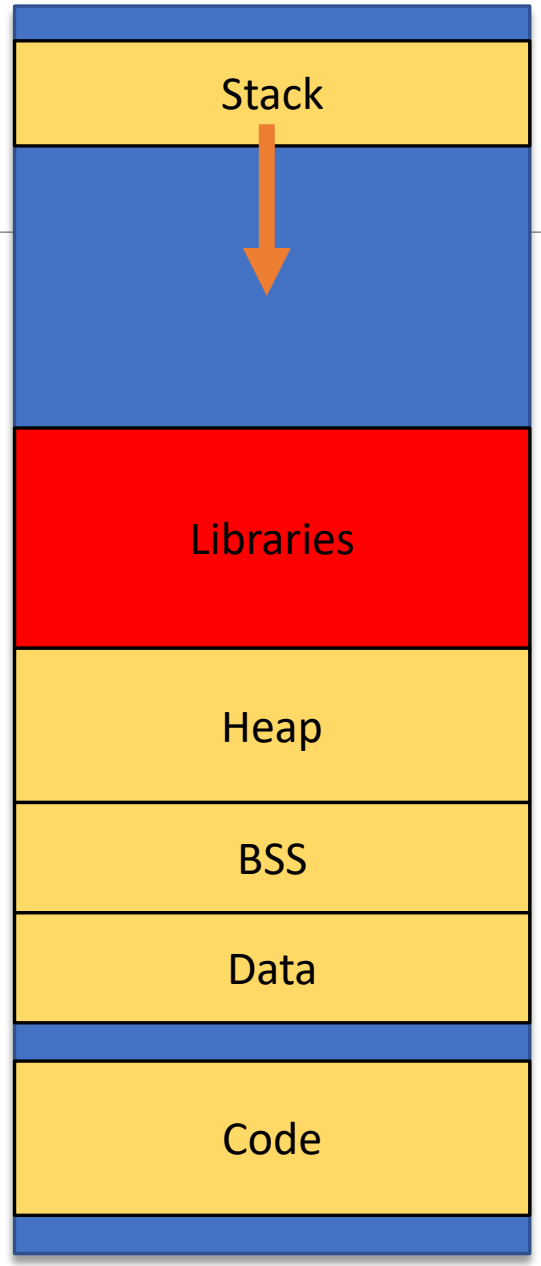
mem.c

Q: How much can it grow?

mem.c

1. **Until a limit imposed by the SO is reached. Ex:**
   - glibc i386, x86_64 7.4 MB
   - Tru64 5.1 5.2 MB
   - Cygwin 1.8 MB
   - Solaris 7..10 1 MB
   - MacOS X 10.5 460 KB
   - AIX 5 98 KB
   - OpenBSD 4.0 64 KB
   - HP-UX 11 16 KB

2. **Until vital memory is overwritten**
   …mostly in embedded (IoT) devices



0xFFFFFFFF
Top of memory

Stack grows downwards

Start of Memory
0x00000000

CWE-120 Classic Overflow

- **Given an input buffer, data is copied without checking its size**
  - If destination buffer is larger than input data, nothing bad happens
  - If destination buffer is smaller than input data, memory is overwritten
- **Impact: memory is overwritten**
  - Mostly affects local variables
  - May change the execution flow
    - Change of local control variables
    - Change of stored Instruction Pointer
  - May be used to inject external code
- **Solution: take in consideration the size of the destination buffer!**

Classic Overflow – prog 1

- **Description:**
  - Reads the username from the command line
  - Input is stored in variable **`username`**
  - Variable can hold strings up to 31 chars
  - **`scanf`** function has no limit on input size
  - **`printf`** will print the content

```c
//classic/prog_1.c
//gcc -O0 -fno-stack-protector -o prog_1 prog_1.c

#include <stdio.h>

int main() {
        char username[32];
        puts("username:");
        scanf("%s", username);
        printf("Welcome %s!\n", username);
        return 0;
}
```

- **Shows a simple write beyond boundaries**
  - **`printf`** also shows a read beyond boundaries

Classic Overflow – prog 1

- **Reading more than 31 chars will result in overwriting the memory after the `username`**
  - There are no other variables, so this will be stack structures (addressed later)
- **`printf` will print chars up to 0x00**
  - potentially printing program memory
  - Function is insecure as there are no explicit boundaries except the actual string content

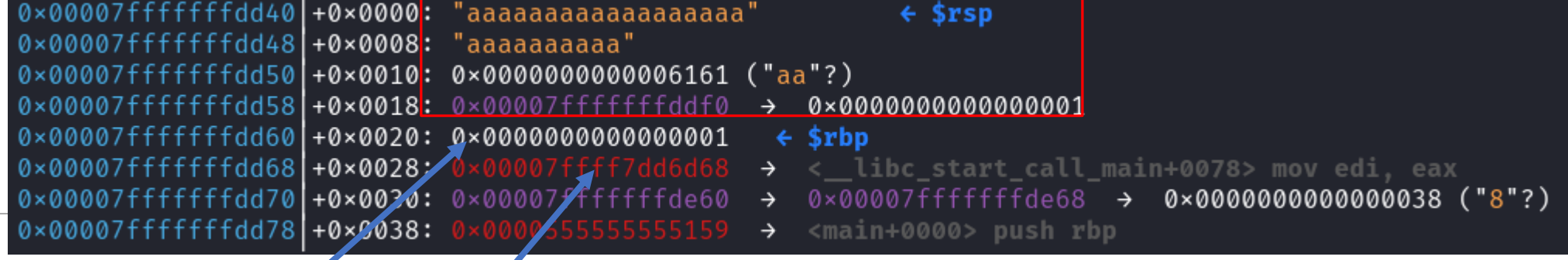
```c
//classic/prog_1.c
//gcc -O0 -fno-stack-protector -o prog_1 prog_1.c

#include <stdio.h>

int main() {
        char username[32];
        puts("username:");
        scanf("%s", username);
        printf("Welcome %s!\n", username);
        return 0;
}
```

Exercise: classic/prog 1

- **Install `gef`: `bash -c "$(curl -fsSL https://gef.blah.cat/sh)"`**
- **Compile the binary: gcc -g -O0 -fno-stack-protector -o prog_1 prog_1.c**
- **Analyze the execution with different payloads**
  - Print register: **`p $rsp`** or variable address **`p &username`**
  - Check stack information: **`info frame`**
- **Determine**
  - What is the stack base address?
  - Where is the return information?
  - How many bytes can be entered without overflow?
  - How many bytes can be written without damage?
  - What happens when an overflow is achieved?

```
$rax   : 0x1
$rbx   : 0x00007fffffffde78  →  0x00007fffffffe1f5  →  "/media/sf_labs/bo/prog_1"
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x00007fffffffdd40  →  "aaaaaaaaaaaaaaaa"
$rbp   : 0x00007fffffffdd60  →  0x0000000000000001
$rsi   : 0xa
$rdi   : 0x00007fffffffd800  →  0x0000000000000000
$rip   : 0x000055555555518b  →  <main+0032> lea rax, [rbp-0x20]
$r8    : 0xa
$r9    : 0xffffffff
$r10   : 0x000055555555600f  →  0x6d6f636c65570073 ("s"?)
$r11   : 0x00007ffff7f3c8e0  →  0x0002000200020002
$r12   : 0x0
$r13   : 0x00007fffffffde88  →  0x00007fffffffe20e  →  "COLORFGBG=15;0"
$r14   : 0x00007ffff7ffd000  →  0x00007ffff7ffe2e0  →  0x0000555555554000  →   jg 0x555555554047
$r15   : 0x0000555555557dd8  →  0x0000555555555110  →  <__do_global_dtors_aux+0000> endbr64
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x00007fffffffdd40|+0x0000: "aaaaaaaaaaaaaaaa"        ← $rsp
0x00007fffffffdd48|+0x0008: "aaaaaaaaaa"
0x00007fffffffdd50|+0x0010: 0x0000000000006161 ("aa"?)
0x00007fffffffdd58|+0x0018: 0x00007fffffffddf0  →  0x0000000000000001
0x00007fffffffdd60|+0x0020: 0x0000000000000001   ← $rbp
0x00007fffffffdd68|+0x0028: 0x00007ffff7dd6d68  →  <__libc_start_call_main+0078> mov edi, eax
0x00007fffffffdd70|+0x0030: 0x00007fffffffde60  →  0x00007fffffffde68  →  0x0000000000000038 ("8"?)
0x00007fffffffdd78|+0x0038: 0x0000555555555159  →  <main+0000> push rbp

   0x55555555517e <main+0025>      mov    rdi, rax
   0x555555555181 <main+0028>      mov    eax, 0x0
   0x555555555186 <main+002d>      call   0x555555555050 <__isoc99_scanf@plt>
●→ 0x55555555518b <main+0032>      lea    rax, [rbp-0x20]
   0x55555555518f <main+0036>      mov    rsi, rax
   0x555555555192 <main+0039>      lea    rax, [rip+0xe78]        # 0x555555556011
   0x555555555199 <main+0040>      mov    rdi, rax
   0x55555555519c <main+0043>      mov    eax, 0x0
   0x5555555551a1 <main+0048>      call   0x555555555040 <printf@plt>

      2
      3  int main() {
      4          char username[32];
      5          puts("username:");
      6          scanf("%s", username);
                        // username=0x00007fffffffdd40  →  "aaaaaaaaaaaaaaaa"
●→    7          printf("Welcome %s!\n", username);
      8          return 0;
      9  }
     10

[#0] Id 1, Name: "prog_1", stopped 0x55555555518b in main (), reason: BREAKPOINT

[#0] 0x55555555518b → main()
```

Saved $BP
Saved $PC



- **What is the stack base address?**
  - `info frame: x7fffffffdd70`
  - `p $rbp: 0x7fffffffdd60`
- **Where is the return information?**
  - Just before `$rbp`
- **How many bytes can be entered without overflow?**
  - `sizeof(username) - 1`
- **How many bytes can be written without damage?**
  - 31
  - It could have been different due to padding.
- **What happens when an overflow is achieved?**
  - Saved **$RBP** is overwritten and then Saved **$PC** is overwritten
  - In this case, 31 ‘a’ were provided and an additional \0 was added at 0x..dd52

Classic Overflow – classic/prog_2

- **Flow:**
  - Asks for username and password
  - Validates credentials
  - Asks for message
  - If user authenticated, access is granted
- **Issues:**
  - Several uncontrolled reads
  - All variables may overwrite other
- **Demonstrates overwrite of local variables**
  - Each vulnerable variable may overwrite others above

```
int main() {
        char allowed = 0;
        char password[8];
        char username[8];
        char message[32];

        puts("username:");
        scanf(“%s”, username);
        puts("password:");
        scanf(“%s”, password);
        allowed = strcmp("admin", username) + \
              strcmp("topsecrt", password);

        puts("message:");
        scanf(“%s”, message);

        printf("user=%s pass=%s result=%d\n", username, \
                  password, allowed);

        if(allowed == 0)
                printf("Access granted. Message sent!\n");
        else
                printf("Access denied\n");

        return 0;
}
```

Classic Overflow – classic/prog_2

- **Variable order will determine how it can be exploited**
  - Implementation dependent
- **`message` is the prime suspect as it is written after the evaluation is done**
- **Can also change an internal decision (flow inside the function) by writing over the `allowed` variable**

```
int main() {
        char allowed = 0;
        char password[8];
        char username[8];
        char message[32];

        puts("username:");
        scanf("%s", username);
        puts("password:");
        scanf("%s", password);
        allowed = strcmp("admin", username) + \
              strcmp("topsecrt", password);

        puts("message:");
        scanf("%s", message);

        printf("user=%s pass=%s result=%d\n", username, \
                   password, allowed);

        if(allowed == 0)
                printf("Access granted. Message sent!\n");
        else
                printf("Access denied\n");

        return 0;
}
```

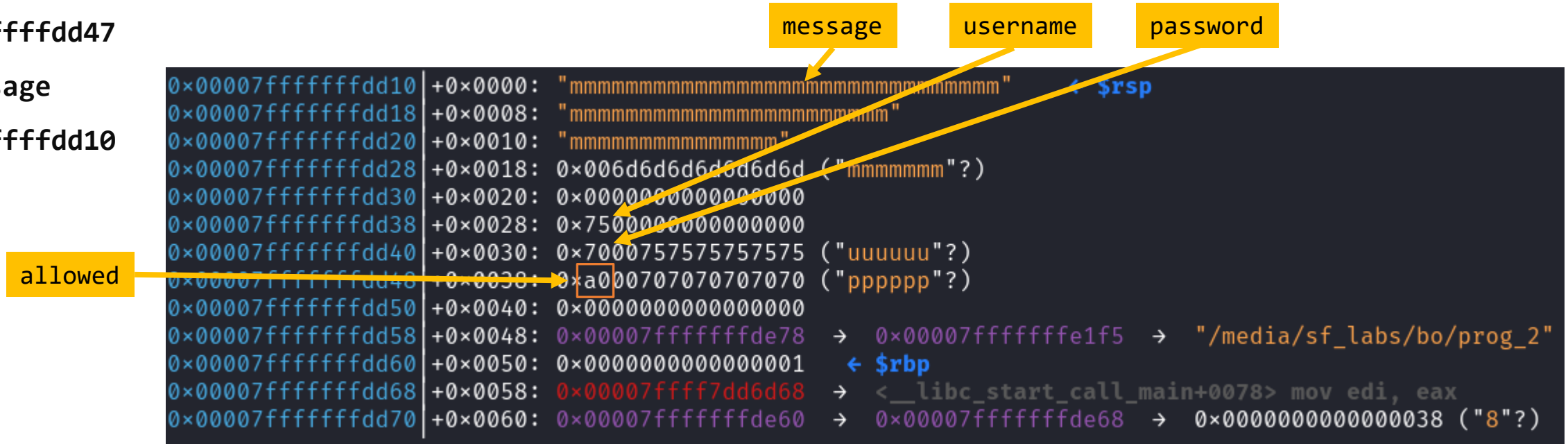
Classic Overflow – classic/prog_2

```
p &allowed
0x7fffffffdd4f
p &username
0x7fffffffdd3f
p &password
0x7fffffffdd47
p &message
0x7fffffffdd10
```

Memory grows from top to bottom

message can be used to overwrite everything!!!

There is padding between the **message** and the **user**



Exercise: classic/prog 2

- **Compile the binary: `gcc -g -O0 -fno-stack-protector -o prog_2 prog_2.c`**
- **Analyze the execution with different payloads**
  - Print register: **`p $rsp`** or variable address **`p &username`**
  - Check stack information: **`info frame`**
- **Determine**
  - What is the stack base address?
  - Where is the return information?
  - How many bytes can be entered to the message without overflow?
  - How many bytes can be written without damage?
  - What happens when an overflow is achieved?
  - How can the decision be subverted?

CWE-126: Buffer Over-read

- **The software reads from a buffer and reference memory locations after the targeted buffer.**
  - using buffer access mechanisms such as indexes or pointers

- **Impact: Allows access to otherwise private data**

- **Most common with:**
  - Casts between structures with different sizes
  - Copy of data without considering the actual size, assuming a general size
  - Copy of data based on corrupted metadata
  - Erasure of `\0` in null terminated strings

Buffer Over-read – overread1.c

- **Program flow:**
  - Program reads a string without boundary checks
  - Memory is manipulated
  - A message is printed

- **Demonstrates a read beyond bounds with `printf`**

- **Impact: private data (message) is disclosed to users**

```
int main(int argc, char* argv[]){
        char message[32];
        char buffer[8];

        printf("Password: ");
        scanf(“%s”, buffer);

        sprintf(message, "Secret message");

        if(strcmp(buffer, "password") == 0)
{
                printf("%s\n", message);
        }else{
                printf("Password %s is
incorrect\n", buffer);
        }
}
```

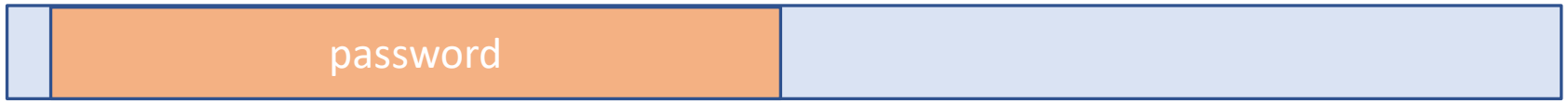
Buffer Over-read – overread1

- **Vulnerability:**
  - In some situations, the password may overflow the buffer, and further memory operations erase the **\0** character
  - Further **printf** of a message will include additional memory

```
int main(int argc, char* argv[]){
        char message[32];
        char buffer[8];

        printf("Password: ");
        scanf("%s", buffer);

        sprintf(message, "Secret message");

        if(strcmp(buffer, "password") == 0)
{
                printf("%s\n", message);
        }else{
                printf("Password %s is
incorrect\n", buffer);
        }
}
```

Buffer Over-read – overread1

- Exercise: Determine what conditions trigger the vulnerability, and what is the impact.
- Write overflow



- Memory manipulation erases the password Null terminator (`\0`)

- Read overflow

Buffer Over-read – server.c

- **Program Flow**
  - Receives a message to a buffer
  - Prints the buffer
  - Returns the buffer through the socket

- **Vulnerability:**
  - Send doesn't respect buffer sizes and will use a buffer larger than expected
  - **`printf`** has no notion of string size and will print everything up to **`\0`**

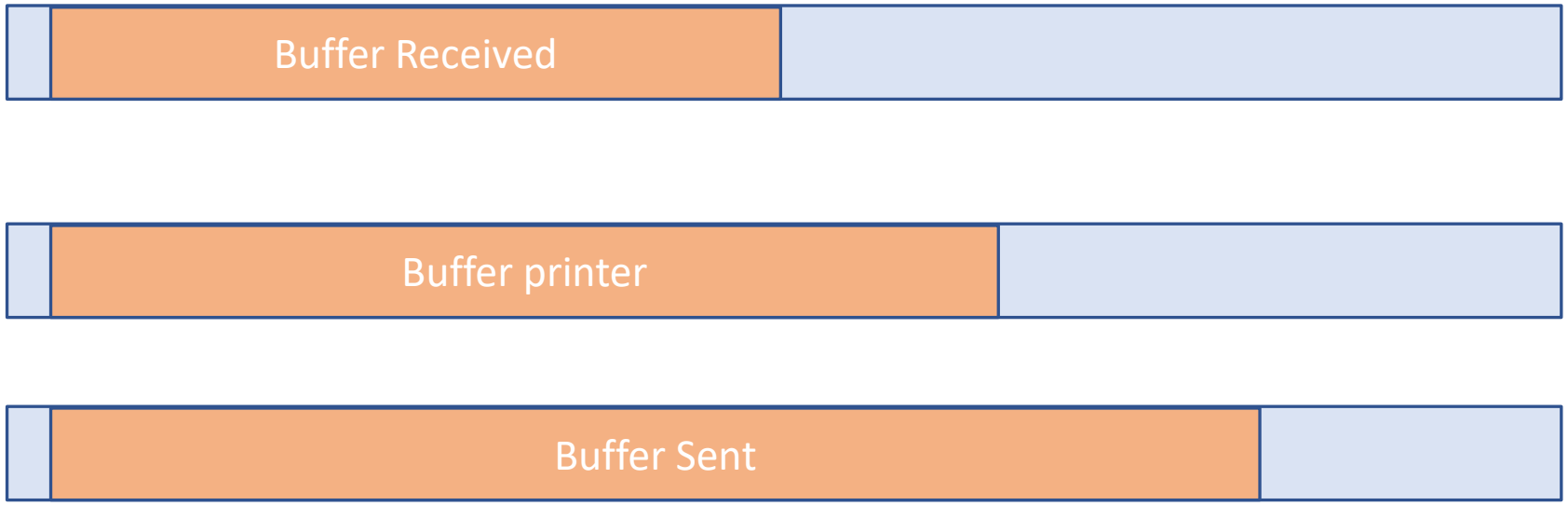
- **Impact: existing memory contents will be sent to clients**

```
while(1){
      n = recvfrom(sockfd, buffer, 32, NULL, &cliaddr, &len);
      printf("%s\n", buffer);
      sendto(sockfd, buffer, MESSAGE_SIZE, NULL, &cliaddr, len);
}
```

Buffer Over-read – server.c

- **Exercise: Determine what conditions trigger the vulnerability, and what is the impact.**

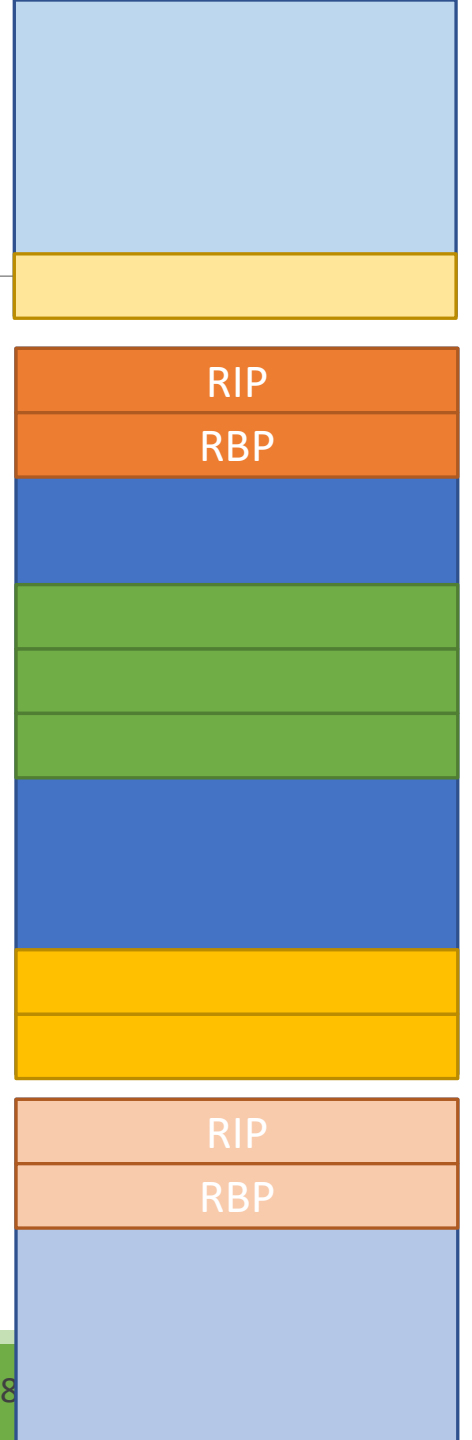
- **Variable structure:**



Stack Overflow

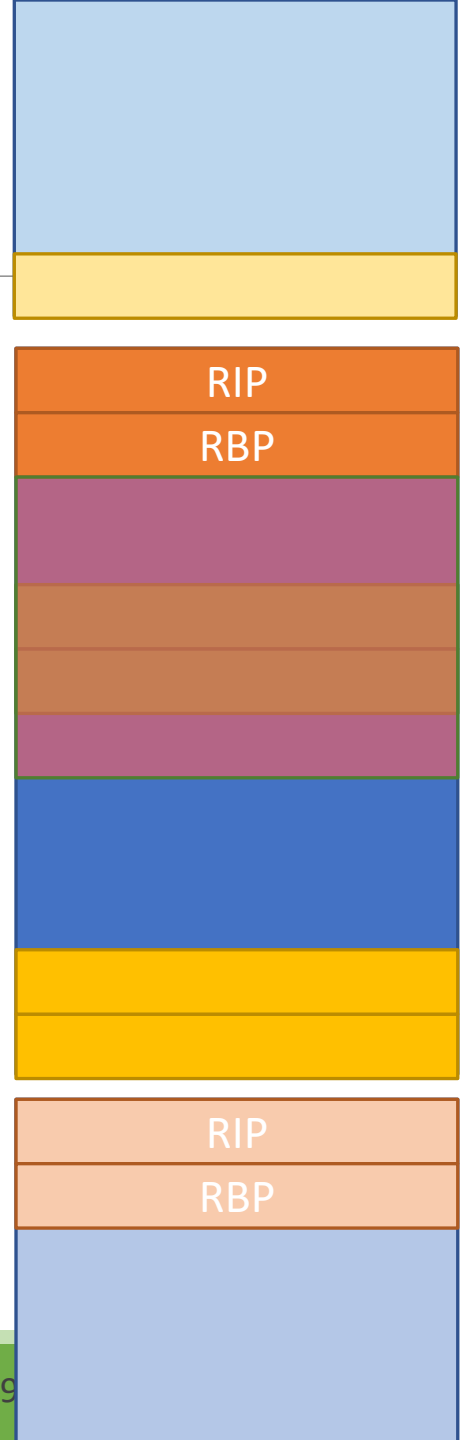
Stack Based Vulnerabilities

- **Stack can be subverted to conduct attacks**
  - it contains local variables (which store user injected data)
  - the program execution flow is kept in the stack
- **Mostly:**
  - Denial of Service: program crashes
  - Memory disclosure: attacker gains access to previous frames
  - Change program flow
  - Injection of malicious code



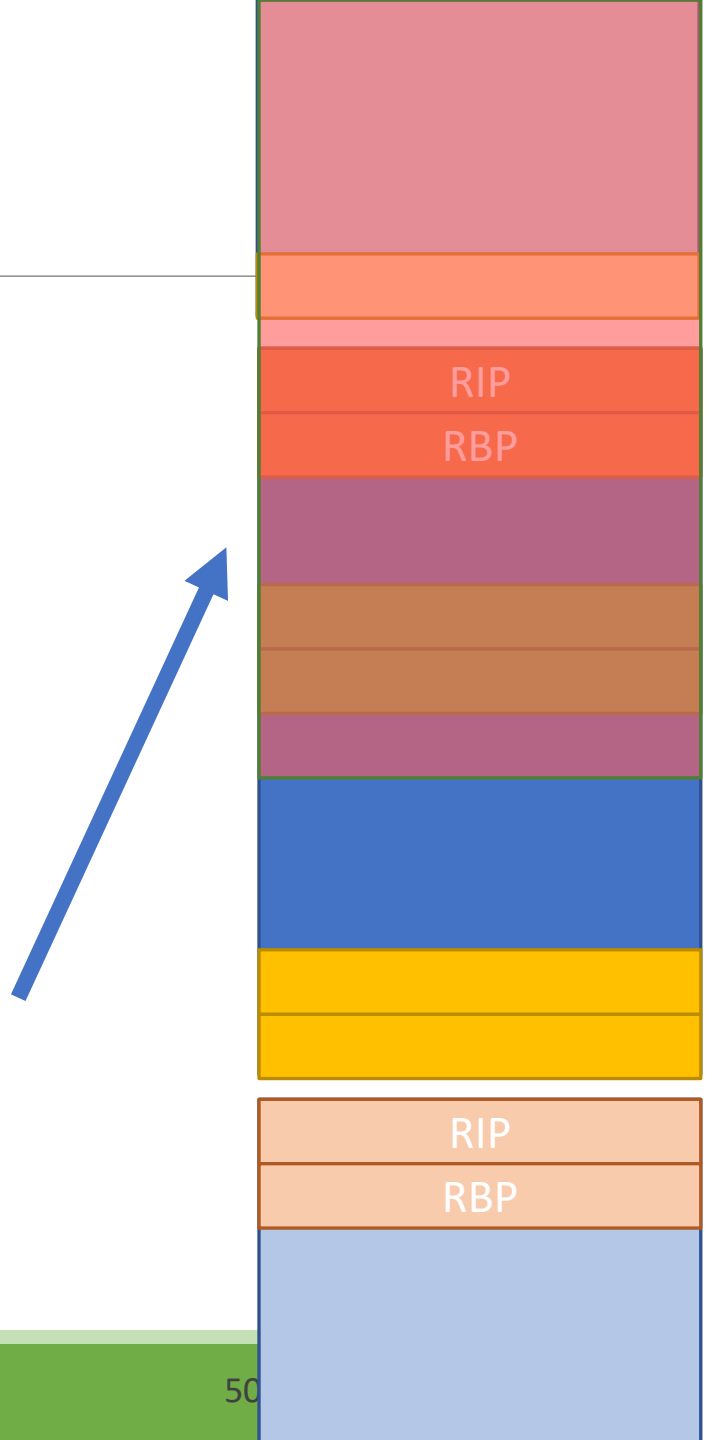
Stack Based Vulnerabilities

- **Recap…**
- **Local variables will overwrite others**
  - Can change data stored
  - Can lead to local memory disclosure
  - Can change local decisions if they depend of stored data



Stack Based Vulnerabilities

- **Recap…**
- **Local variables will overwrite others**
  - Can change data stored
  - Can lead to local memory disclosure
  - Can change local decisions if they depend of stored data
- **Further writing will overwrite flow information**
  - If done blindly, program will crash (why?)
- **It affects frames from previous functions**

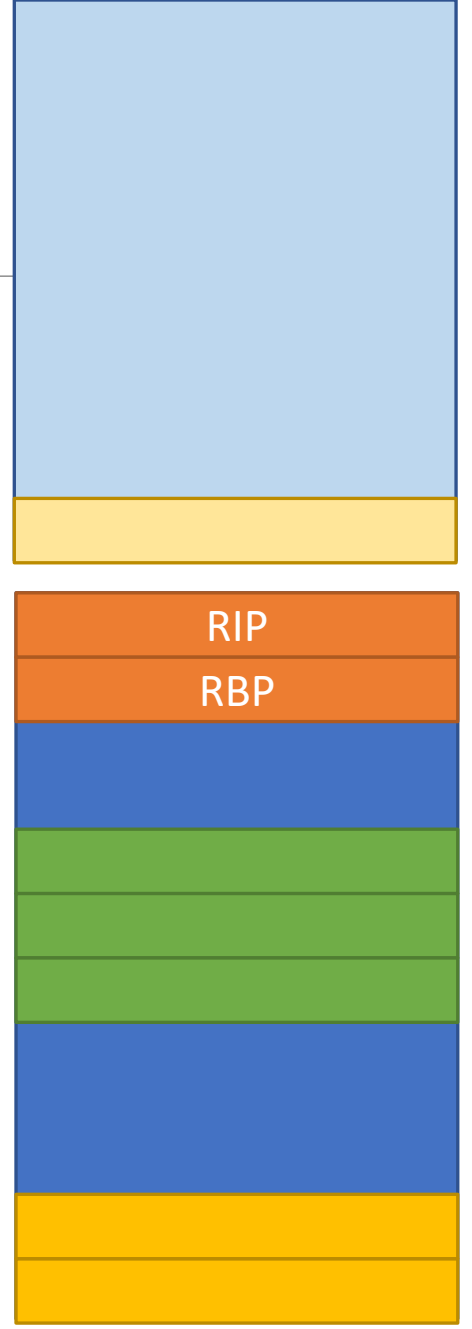


Stack Smashing

- **What about writing the correct values to the stack?**
  - Some value to RBP
  - An address belonging to the process in RIP

- **Well… when the message ends the flow will be restored**
  - That is… **stored RBP and stored RIP are loaded into the registers**
  - The stack frame will start at RBP
  - Program jump to the address in RIP

- **If the addresses aren't in a mapped area, program will receive a `SIGSEGV`**



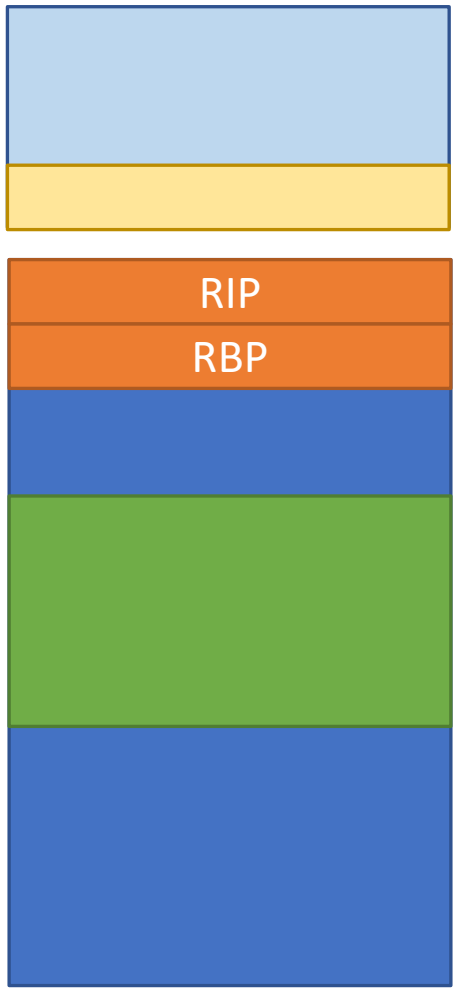
Stack: program_flow.c

- **Program flow:**
  - Reads data from file
  - Calls **foo** function with size and buffer
  - **foo** has an overflowing **memcpy**
  - **secret** function is never called
- **Attack: Overflow the buffer**
  - writing over stored $**RBP**
  - writing over stored $**RIP**, placing **&secret** there
- **Consider ASLR to be disabled**

```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack: program_flow.c

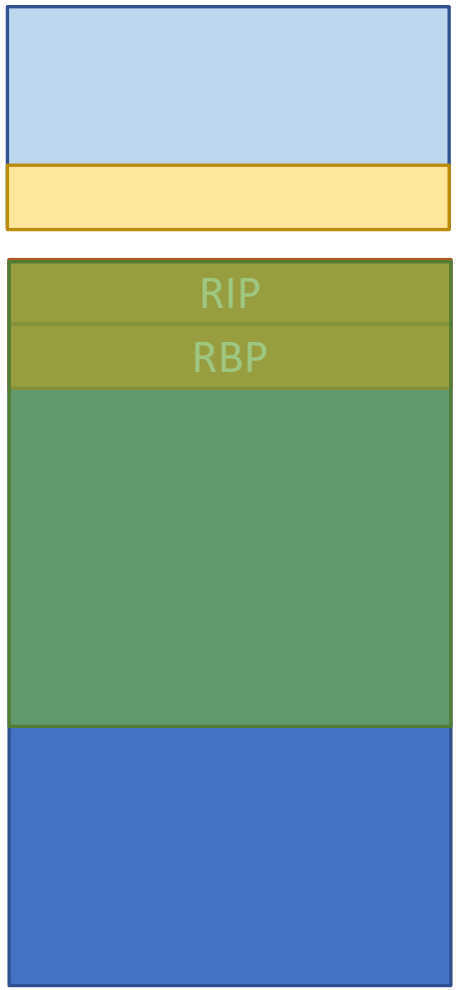
- **Main stack**
- **Foo stack**
  - Stored program flow
  - **buffer[8]**
- **Secret has no stack!**




```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack: program_flow.c

- **Attack strategy**
  - Overwrite buffer over **RBP/RIP**
- **How to find the addresses?**
  - If we have the source code: **`printf("%p\n", secret);`**
  - If we don't: **gdb** or bruteforce




```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack: program_flow.c

```
$ ./program_flow payload

0x8001209



$ gdb program_flow payload

gdb$ br main

gdb$ run

gdb$ print &secret

gdb$ 5 = (void (*)()) 0x8001209 <secret>
```

Value to inject **program** vs **gdb** may yield different values!

```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack Smashing – program_flow.c

- Typical flow


```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack: program_flow.c

- Flow subverted to secret()


```c
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    fclose(fp);

    foo(size, buffer);
    return 0;
}
```

Stack: program_flow.c

```
$ program_flow payload

0x8001209

Secret message
```

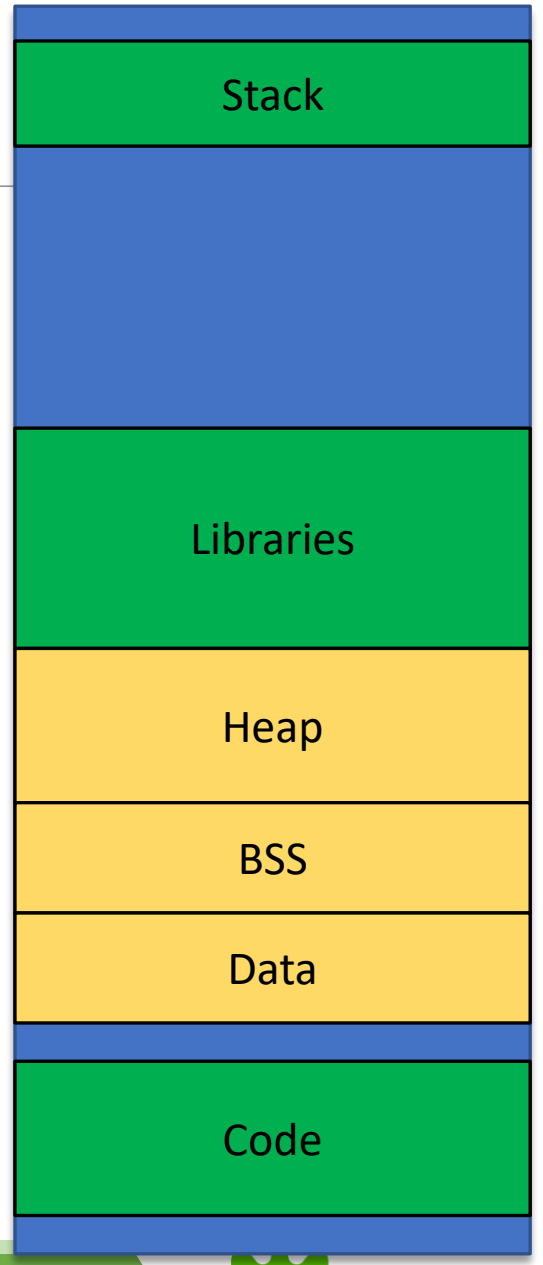
With the correct payload, secret() is called

Q: What payload?

```
void secret(){
    printf("Secret message\n");
    exit(0);
}
char foo(int size, char* arg){
    char buffer[8];
    memcpy(buffer, arg, size);
    return buffer[0];
}
int main(int argc, char* argv[]){
    char buffer[64];
    printf("%p\n", &secret);

    FILE *fp = fopen(argv[1], "r");
    int size = fread(buffer, 1, 64, fp);
    e(fp);

    ize, buffer);
    n 0;
```

Stack: return_to_libc.c

- **Instead of returning to a program function it is possible to jump to other locations**
  - In theory, any segment allocated to the program
  - In practice, permission mechanisms limit the available segments
- **Segments for libraries have several generic libraries**
  - And relevant functions: `system()`
- **Stack can be executable**
  - but it isn't on recent systems



Stack: return_to_libc.c

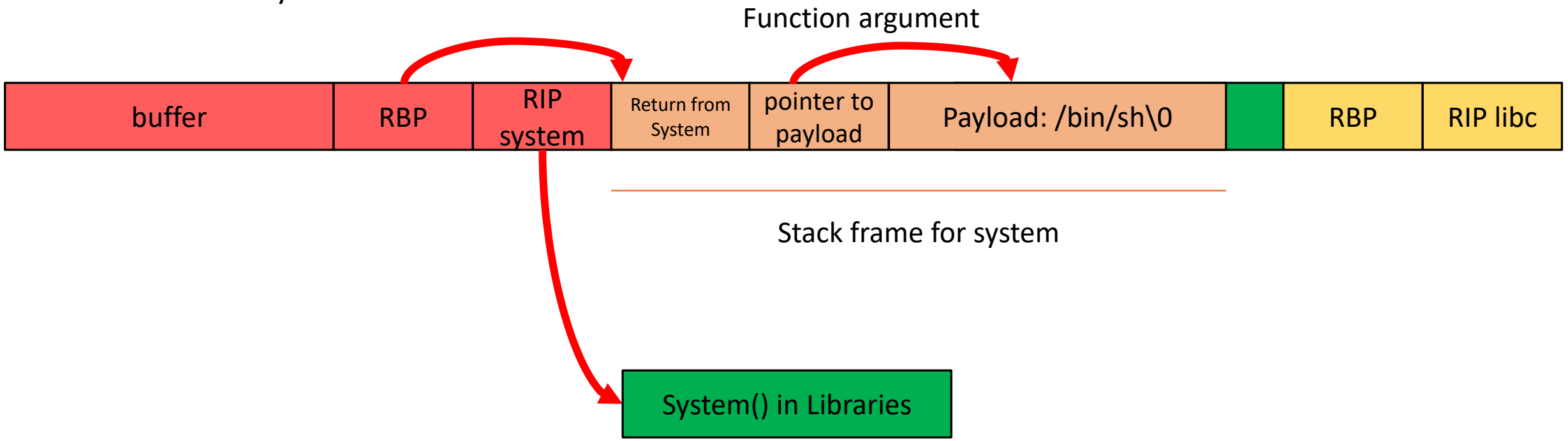
Typical Flow

Return to libc

- Build “fake” Stack frame and call system() with one argument
  - Argument is the command to execute (e.g. a reverse shell)
- Must take in consideration calling convention
  - Which is architecture dependent

Stack: return_to_libc.c (32bits)

- **Arguments are passed in the stack**
  - Approach: store values to the stack so that system is called with a **shellcode**
    - Then call system



Countermeasures: Data Executable Prevention

- **Non Executable Stack (NX) (Data Executable Prevention)**
  - Most binaries do not allow running code from Stack
  - Stack segments are marked as Non Executable (NX bit)
    - IP cannot jump to it
    - Injection of a shellcode is not possible

- **Introduced in recent OS, but can be disabled**
  - Not ubiquitous on embedded devices
  - Binaries must opt-in!

Countermeasures: Canaries

- **Uses references values after local variables to detect overflow**
  - Value is placed when the function starts
  - Value is compared before function exits
  - Program is interrupted if values do not match

- **Stack canaries:**

Countermeasures: Canaries

Without Canaries

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
lea     rax, -10[rbp]
mov     rsi, rax
lea     rdi, .LC0[rip]
mov     eax, 0
call    __isoc99_scanf@PLT
lea     rax, -10[rbp]
mov     rdi, rax
call    puts@PLT
nop
leave
ret
```

With Canaries

```
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     rax, QWORD PTR fs:40
        mov     QWORD PTR -8[rbp], rax
        xor     eax, eax
        lea     rax, -18[rbp]
        mov     rsi, rax
        lea     rdi, .LC0[rip]
        mov     eax, 0
        call    __isoc99_scanf@PLT
        lea     rax, -18[rbp]
        mov     rdi, rax
        call    puts@PLT
        nop
        mov     rax, QWORD PTR -8[rbp]
        xor     rax, QWORD PTR fs:40
        je      .L2
        call    __stack_chk_fail@PLT
L2:
        leave
        ret
```

Gets value from fs:40
Stores value at rbp-8 (inside stack frame)

Fetches value
Xor with reference at fs:40
Exit or crash

Countermeasures: Canaries

- **`-fno-stack-protector`:** disables stack protection. (What we have been using)

- **`-fstack-protector:`** enables stack protection for vulnerable functions that contain:
  - A character array larger than 8 bytes.
  - An 8-bit integer array larger than 8 bytes.
  - A call to `alloca()` with either a variable size or a constant size bigger than 8 bytes.

- **`-fstack-protector-strong`:** enables stack protection for vulnerable functions that contain:
  - An array of any size and type.
  - A call to `alloca()`.
  - A local variable that has its address taken.

- **`-fstack-protector-all:`** adds stack protection to all functions regardless of their vulnerability.

Stack: return_to_libc.c (x86_64)

- **x64: first arguments are passed in register: RDI, RSI, RDX, RCX, R8, R9**
  - Approach: load RDI with address of string, jump to system address
  - Problems: cannot jump to stack (due to NX)

- **Improved:**
  - Search any code that loads RDI from stack
    - we can **control the execution flow** after the function exists but we cannot execute code from the stack
  - jump to code that loads RDI from stack
  - Jump to system

ROP - Return Oriented Programming

- **Execute code already present in the program.**
  - Each snippet is composed by some instructions + **RET (called a Gadget)**
  - **RET** pops RIP from the stack
- **Program flow is controlled by values in the stack**
  - Attacker puts values in stack pointing to gadgets
  - When a gadget ends, the code jumps to the next gadget
- **Any program can be constructed as long as there are gadgets available**
  - When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC [1] - Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S.
  - Return-Oriented Programming: Exploits Without Code Injection [2] - Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan.

ROP

- **ROP Attacks: Chain gadgets to execute malicious code.**

- **A gadget is a suite of instructions which end by the branch instruction ret (Intel) or the equivalent on ARM.**

Intel examples:

```
pop eax ; ret
xor ebx, ebx ; ret
```

ARM examples:

```
pop {r4, pc}
str r1, [r0] ; bx lr
```

- **Objective: Use gadgets instead of classical shellcode**
  - Because shellcode cannot be executed from stack
  - Gadgets reside in the program code and can be executed

ROP

- Because x86 instructions aren't aligned, a gadget can contain another gadget.

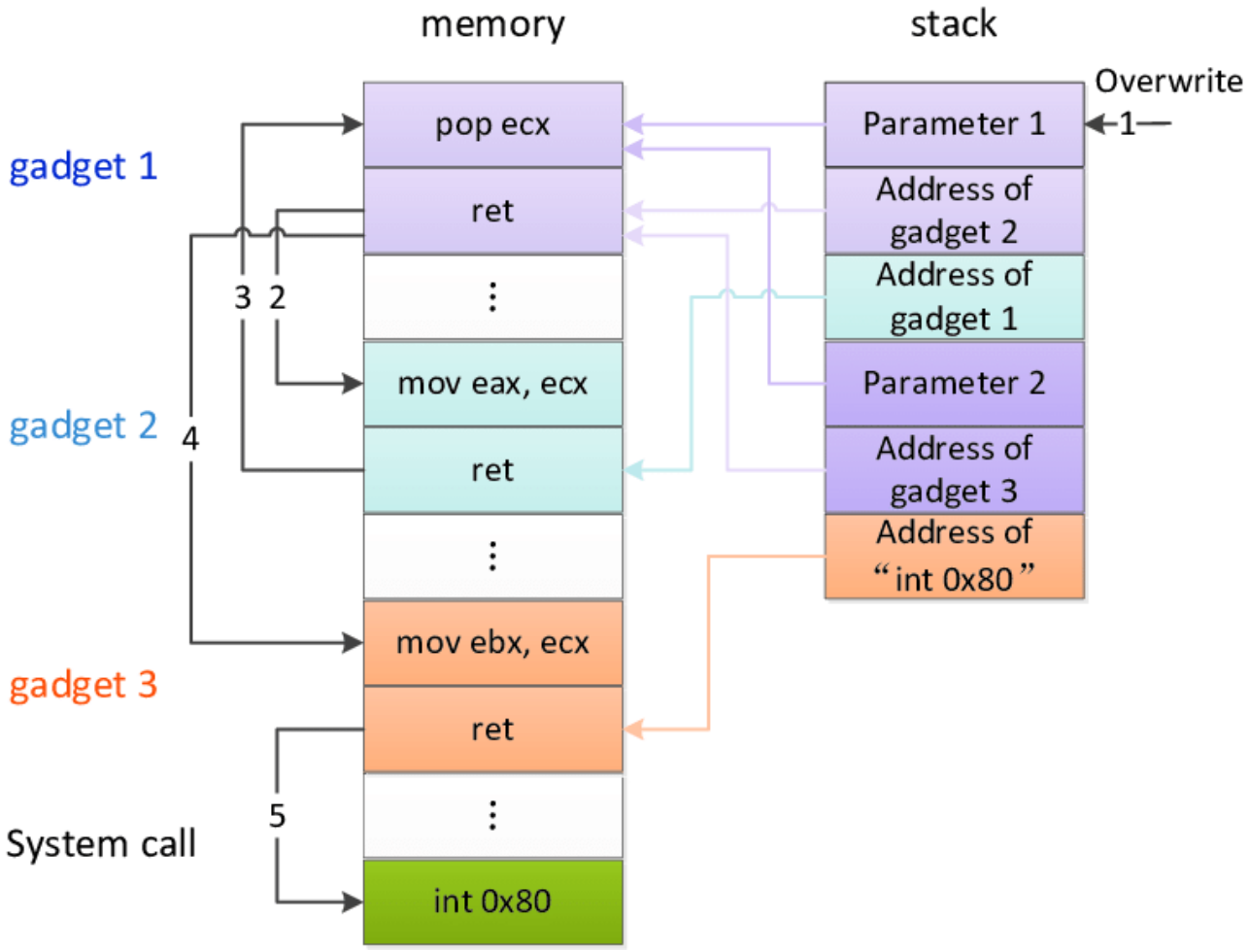
```
f7c7070000000f9545c3 → test edi, 0x7 ; setnz byte ptr [rbp-0x3d] ;
  c7070000000f9545c3 → mov dword ptr [rdi], 0xf000000 ; xchg ebp, eax ; ret
```

- Doesn't work on RISC architectures like ARM, MIPS, SPARC...

```
0x000000000040137d: nop dword ptr [rax]; ret;
0x0000000000401003: or byte ptr [rax - 0x75], cl; add eax, 0x2fed; test rax, rax; je 0x1012; call rax;
0x0000000000401136: or dword ptr [rdi + 0x404080], edi; jmp rax;
0x0000000000401106: out 0x2e, al; add byte ptr [rax], al; hlt; nop dword ptr [rax + rax]; ret;
0x0000000000401374: pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401376: pop r13; pop r14; pop r15; ret;
0x0000000000401378: pop r14; pop r15; ret;
0x000000000040137a: pop r15; ret;
0x0000000000401373: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401377: pop rbp; pop r14; pop r15; ret;
0x00000000004011a9: pop rbp; ret;
0x000000000040131b: pop rbx; pop rbp; ret;
0x000000000040137b: pop rdi; ret;
0x0000000000401379: pop rsi; pop r15; ret;
0x0000000000401375: pop rsp; pop r13; pop r14; pop r15; ret;
0x0000000000401199: push rbp; mov rbp, rsp; call 0x1120; mov byte ptr [rip + 0x2eef], 1; pop rbp; ret;
0x0000000000401072: ret 0x2f;
0x000000000040128f: ret 0x4011;
0x000000000040100d: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x0000000000401100: sbb eax, 0xff004012; adc eax, 0x2ee6; hlt; nop dword ptr [rax + rax]; ret;
0x0000000000401385: sub esp, 8; add rsp, 8; ret;
0x0000000000401001: sub esp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x0000000000401384: sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: sub rsp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401133: test eax, eax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401175: test eax, eax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401132: test rax, rax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401174: test rax, rax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040121a: clc; leave; ret;
0x000000000040110a: hlt; nop dword ptr [rax + rax]; ret;
0x000000000040121b: leave; ret;
0x000000000040113f: nop; ret;
0x0000000000401016: ret;

106 gadgets found
gef➤
```

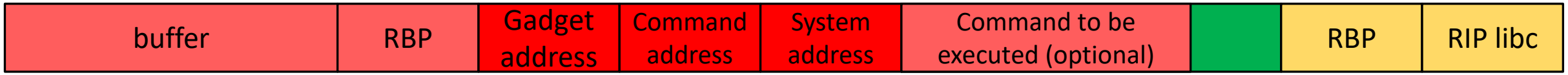
ROP

- **Using ROP, stack is subverted to create a jump sequence. It contains:**
  - Values to be loaded
  - Addresses to other gadgets
  - May also contain arguments to functions called
- **Gadgets are present in program code and loaded libraries**
  - Each function available provides one gadget
  - Plus misaligned access
- **Why?**
  - It can bypass several security mechanisms
  - It can use existing binary data to run arbitrary commands



```
0x0000000000401373: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401377: pop rbp; pop r14; pop r15; ret;
0x00000000004011a9: pop rbp; ret;
0x000000000040131b: pop rbx; pop rbp; ret;
0x000000000040137b: pop rdi; ret;
0x0000000000401379: pop rsi; pop r15; ret;
0x0000000000401375: pop rsp; pop r13; pop r14; pop r15; ret;
0x0000000000401199: push rbp; mov rbp, rsp; call 0x1120; mov byte ptr [rip + 0x2eef], 1; pop rbp; ret;
0x0000000000401072: ret 0x2f;
0x000000000040128f: ret 0x4011;
0x000000000040100d: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x0000000000401100: sbb eax, 0xff004012; adc eax, 0x2ee6; hlt; nop dword ptr [rax + rax]; ret;
0x0000000000401385: sub esp, 8; add rsp, 8; ret;
0x0000000000401001: sub esp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x0000000000401384: sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: sub rsp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401133: test eax, eax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401175: test eax, eax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401132: test rax, rax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401174: test rax, rax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040121a: clc; leave; ret;
0x000000000040110a: hlt; nop dword ptr [rax + rax]; ret;
0x000000000040121b: leave; ret;
0x000000000040113f: nop; ret;
0x0000000000401016: ret;

106 gadgets found
gef➤  rop --search 'pop rdi'
[INFO] Load gadgets from cache
[LOAD] loading ... 100%
[LOAD] removing double gadgets ... 100%
[INFO] Searching for gadgets: pop rdi


0x000000000040137b: pop rdi; ret;
```

Stack: return_to_libc.c (x86_64)



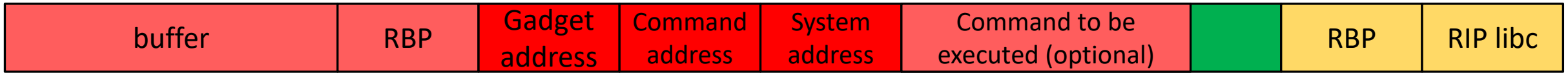
- **Payload strategy:**
  - All addresses are 8 bytes
  - Buffer: padding with 16 bytes (buffer + RBP)
  - Gadget address: ?? -> **`rop --search "pop rdi; ret"`**
    - pop RDI: load command address into RDI
    - ret: load system address into RIP
  - Command address: ?? -> **`grep /bin/sh`**
    - Approaches: **Find a string already in RAM (better);** add the payload after the system address (if required)
  - System address: ?? -> **`print system`**

```
0x000000000040100d: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x0000000000401100: sbb eax, 0xff004012; adc eax, 0x2ee6; hlt; nop dword ptr [rax + rax]; ret;
0x0000000000401385: sub esp, 8; add rsp, 8; ret;
0x0000000000401001: sub esp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x0000000000401384: sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: sub rsp, 8; mov rax, qword ptr [rip + 0x2fed]; test rax, rax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax;
0x000000000040100c: test eax, eax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401133: test eax, eax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401175: test eax, eax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax;
0x000000000040100b: test rax, rax; je 0x1012; call rax; add rsp, 8; ret;
0x0000000000401132: test rax, rax; je 0x1140; mov edi, 0x404080; jmp rax;
0x0000000000401174: test rax, rax; je 0x1180; mov edi, 0x404080; jmp rax;
0x000000000040121a: clc; leave; ret;
0x000000000040110a: hlt; nop dword ptr [rax + rax]; ret;
0x000000000040121b: leave; ret;
0x000000000040113f: nop; ret;
0x0000000000401016: ret;

106 gadgets found
gef➤  rop --search 'pop rdi'
[INFO] Load gadgets from cache
[LOAD] loading ... 100%
[LOAD] removing double gadgets ... 100%
[INFO] Searching for gadgets: pop rdi

[INFO] File: /media/sf_labs/bo/return_to_libc
0x000000000040137b: pop rdi; ret;

gef➤  print system
$2 = {int (const char *)} 0x7ffff7dff8f0 <__libc_system>
gef➤  grep "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/usr/lib/x86_64-linux-gnu/libc.so.6'(0x7ffff7f3a000-0x7ffff7f90000), permission=r--
  0x7ffff7f54e43 - 0x7ffff7f54e4a  →   "/bin/sh"
```

.plt, .got, .got.plt

- **Procedure Linkage Table and Global Offset Table**
  - .**PLT**: Code to relocate symbols
  - .**GOT**: Array with addresses of each symbol requiring relocation
    - .**got** is similar to **.got.plt** but it's writable, while **.got** may be marked as Read Only as a security measure (-z relro)
    - Using a table (GOT) allows patching this table, while keeping libraries in same address, shared to multiple processes
- **Sections required for lazy binding (real time relocation)**
  - Linker needs to resolve the effective address of a code identified by a symbol (e.g **`puts`**)
  - **.plt** and **.got** ensure the symbol location is found and the code jumps around correctly
  - On Linux, the Env Variable `LD_BIND_NOW` forces linking by the linker (on program load)
- **Payload can use that table to obtain addresses to relevant functions**

Stack: return_to_libc.c (x86_64)



- **Payload strategy:**
  - All addresses are 8 bytes
  - Buffer: padding with 16 bytes (buffer + RBP)
  - Gadget address: `0x0040137b`
    - pop RDI: load command address into RDI
    - ret: load system address into RIP
  - /bin/sh address: `0x7ffff7f54e43`
    - Approaches: **Find a string already in RAM (better);** add the payload after the system address (if required)
  - System address: `0x401060`

Stack: return_to_libc.c (x86_64)

- **In some systems, stack must be aligned to 16 bytes and our ROP chain isn't…**
  - Result is a crash in instruction **`movaps`**

- **Solution: add another gadget with only a ret (will pop a value)**
  - Gadget 1: `0x00401016 ; ret`
  - Gadget 2: `0x0040137b ; pop rdi;ret`

Stack: return_to_libc.c (x86_64)

```
106 gadgets found
gef➤  rop --search 'pop rdi'
[INFO] Load gadgets from cache
[LOAD] loading ... 100%
[LOAD] removing double gadgets ... 100%
[INFO] Searching for gadgets: pop rdi

[INFO] File:                    return_to_libc
0x000000000040137b: pop rdi; ret;

gef➤  print system
$2 = {int (const char *)} 0x7ffff7dff8f0 <__libc_system>
gef➤  grep "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/usr/lib/x86_64-linux-gnu/libc.so.6'(0x7ffff7f3a000-0x7ffff7f90000), permission=r--
  0x7ffff7f54e43 - 0x7ffff7f54e4a  →   "/bin/sh"
gef➤  run ./payload
Starting program:                   return_to_libc ./payload
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
secret: 0x4011c2
system: 0x401060
buffer: 0x7fffffffdca8
[Detaching after vfork from child process 695711]
$ whoami
kali
$
```

Stack: return_to_libc.c (x86_64)

- **Exercise: build a ROP chain and get a shell in the program**
  - Use Pwntools
  - It may be useful to disable ASLR for now
    - In gef: **`aslr off`**
    - System wide (as root): **`echo 0 > /proc/sys/kernel/randomize_va_space`**
  - Document the payload

Other attacks: ret2syscall

- **Execute a syscall with an attacker controlled command**
  - Command is controlled by the attacker as it is inserted as a string
  - Require some more gadgets, favoring the use in static binaries
  - Needed:
    - pop rdx; ret
    - pop rdi; ret
    - pop rsi; ret
    - pop rdx; ret
    - mov qword ptr [rax], rdx
    - sys_execve
    - Some memory address for writing the command

RAX	59
RDI	const char *filename
RSI	const char *const argv[]
RDX	const char *const envp[]

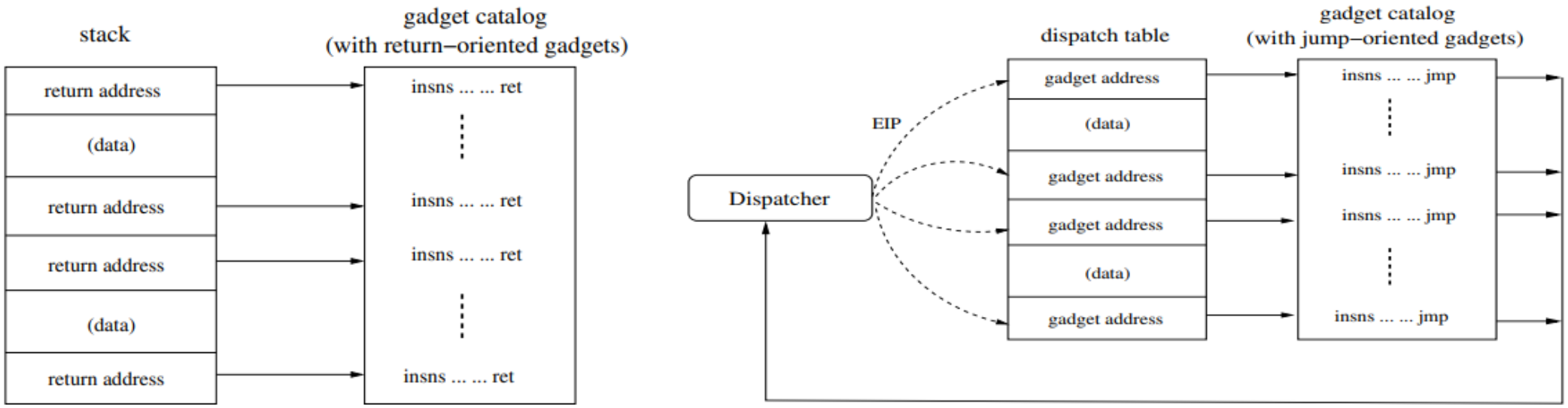
```
Padding + pop_rdx + string\0 + pop_rax + p64(address) + write + pop_rax + p64(0x3b) + pop_rdi +
p64(address) + pop_rsi + p64(0) + pop_rdx + p64(0) + syscall
```

ROP Variants

- **JOP: Jump Oriented Programming**
  - https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf
- **SOP: Jump Oriented Programming**
  - https://www.lst.inf.ethz.ch/research/publications/PPREW_2013/PPREW_2013.pdf
- **BROP: Blind Return Oriented Programming**
  - http://www.scs.stanford.edu/brop/bittau-brop.pdf

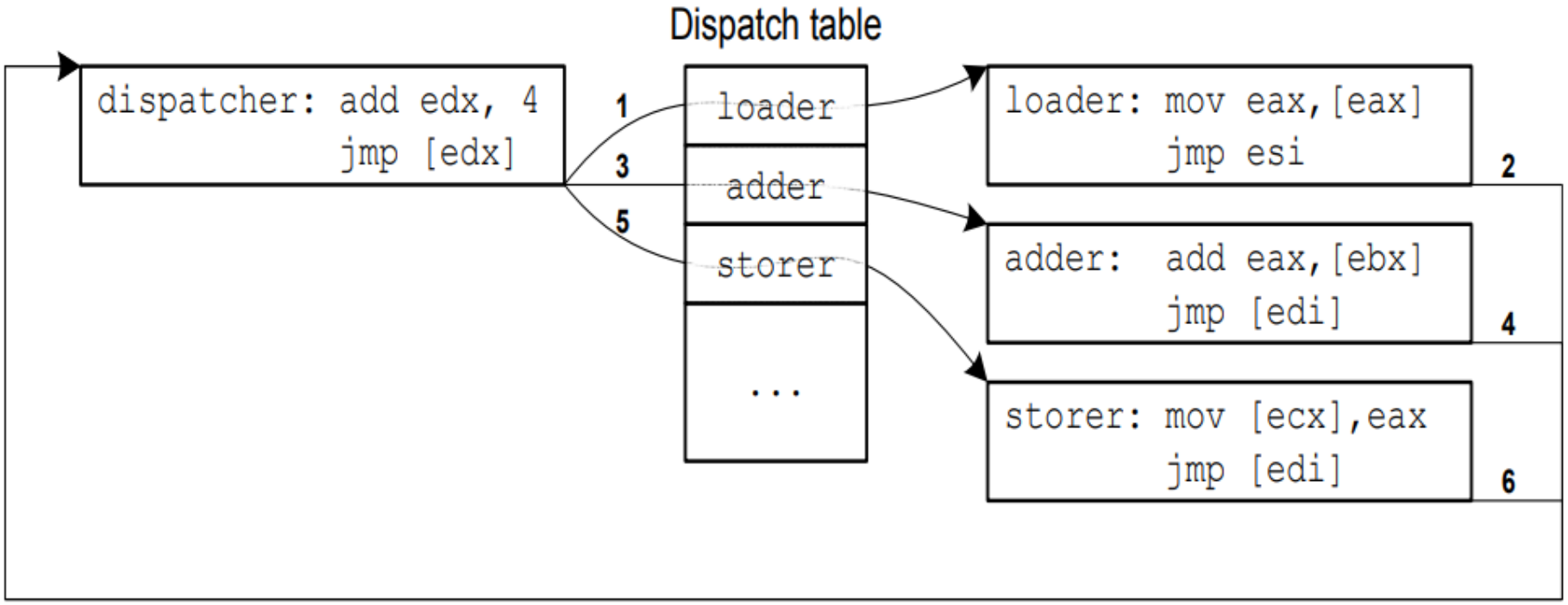
Jump Oriented Programming

- **Explores small gadgets that end with an indirect JMP with a dispatcher**
  - Indirect jmp: jmp [register]
  - Is assumed to be more complex to detect and avoid as interaction is restricted to code and registers
  - Although number of JMP gadgets is smaller, unaligned execution **create jumps** not previously present in the code
  - The program counter is **any register**



Tyler Bletsch, Xuxian Jiang, Vince W. Freeh , Zhenkai Liang "Jump-Oriented Programming: A New Class of Code-Reuse Attack", 2011

Jump Oriented Programming



Tyler Bletsch, Xuxian Jiang, Vince W. Freeh , Zhenkai Liang "Jump-Oriented Programming: A New Class of Code-Reuse Attack", 2011

String Oriented Programming

- **Makes use of a String format bug**
  - Present in the printf family of functions (printf, vprintf, fprintf)
  - Correct: **`printf("%s", str);`**
  - Vulnerable: **`printf(str);`**

- **Format string attacks read/write arbitrary values to arbitrary memory locations**
  - Explore %p, %n, %s,
  - Can be used to dump stack or arbitrary reading of memory
  - Can be used to trigger ROP, JOP attacks by writing values memory
  - Instead of writing sequential chunks, SOP can issue **arbitrary writes**.

- **Two approaches**
  - Direct control flow redirect: Erase return value on stack, jumping to gadget on function end
  - Indirect control flow redirect: Erase a Global Offset Table entry
    - GOT keeps addresses to external symbols as resolved by the linker

Mathias Payer, Thomas R. Gross, String oriented programming: when ASLR is not enough", Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering, 2013

Blind Return Oriented Programming

- **Makes it possible to write exploits without possessing the target's binary.**
  - It requires a stack overflow and a service that restarts after a crash.
  - Based on whether a service crashes
  - Is able to construct a full remote exploit that leads to a shell.

- **The attack remotely leaks gadgets to perform the write system call, after which the binary is transferred from memory to the attacker's socket.**
  - Following that, a standard ROP attack can be carried out.
  - Apart from attacking proprietary services, BROP is very useful in targeting open-source software for which the particular binary used is not public

A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, D. Boneh: Hacking Blind. In Oakland 2014.