Buffer overflows

JOÃO PAULO BARRACA

BO - 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

BO - 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

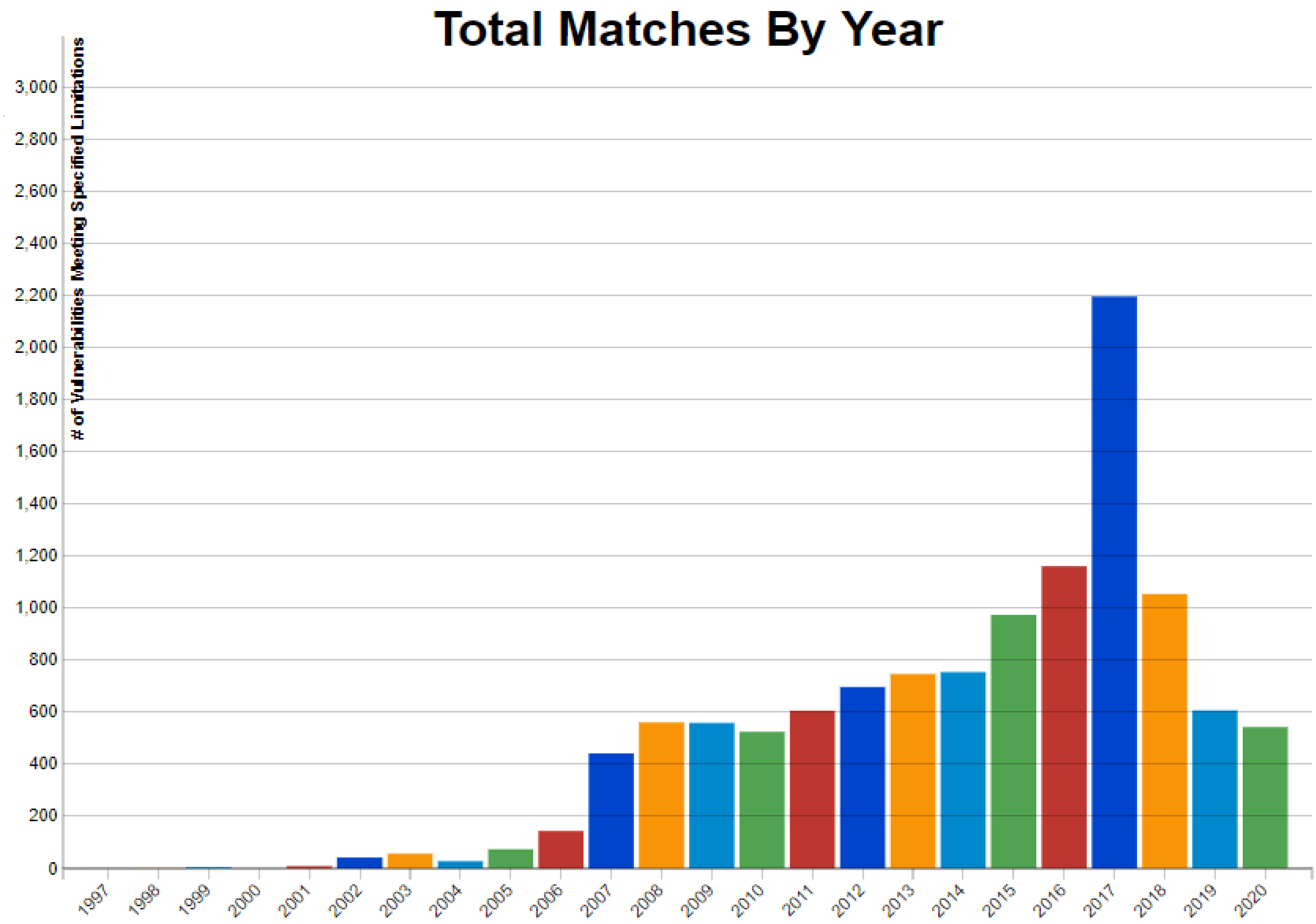
CWE-122: Heap-based Buffer Overflow: overwrite over data in the Heap Segment

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

```
# 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

AAAAAAAAAAAAAAAAd AAAAAAAAAAd
```

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 (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));
```

```
$ ./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.

```c
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
117098867967446225O
```

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);
```

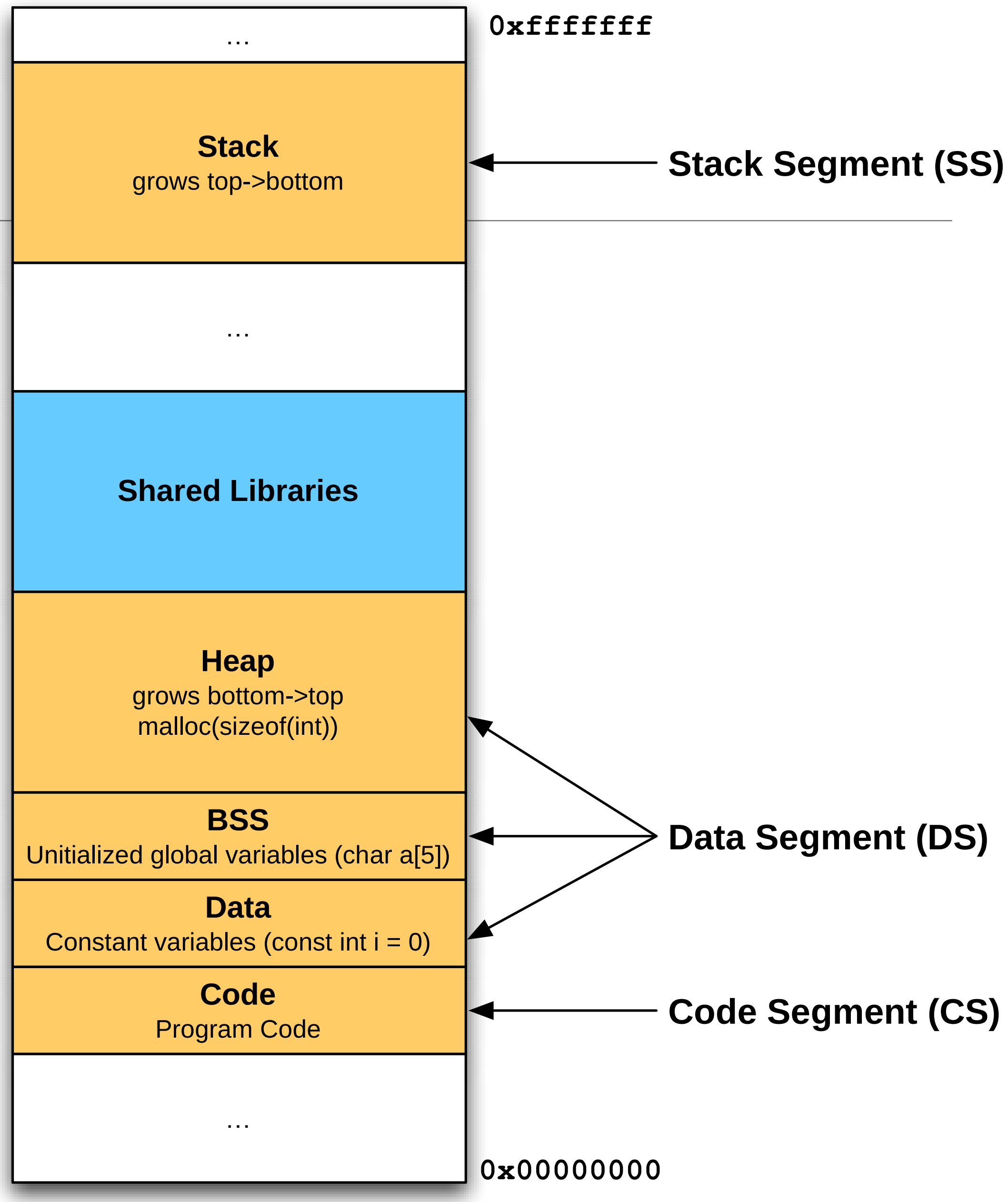
```
$ ./dynamic_memory
Hello World!!!!
```

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

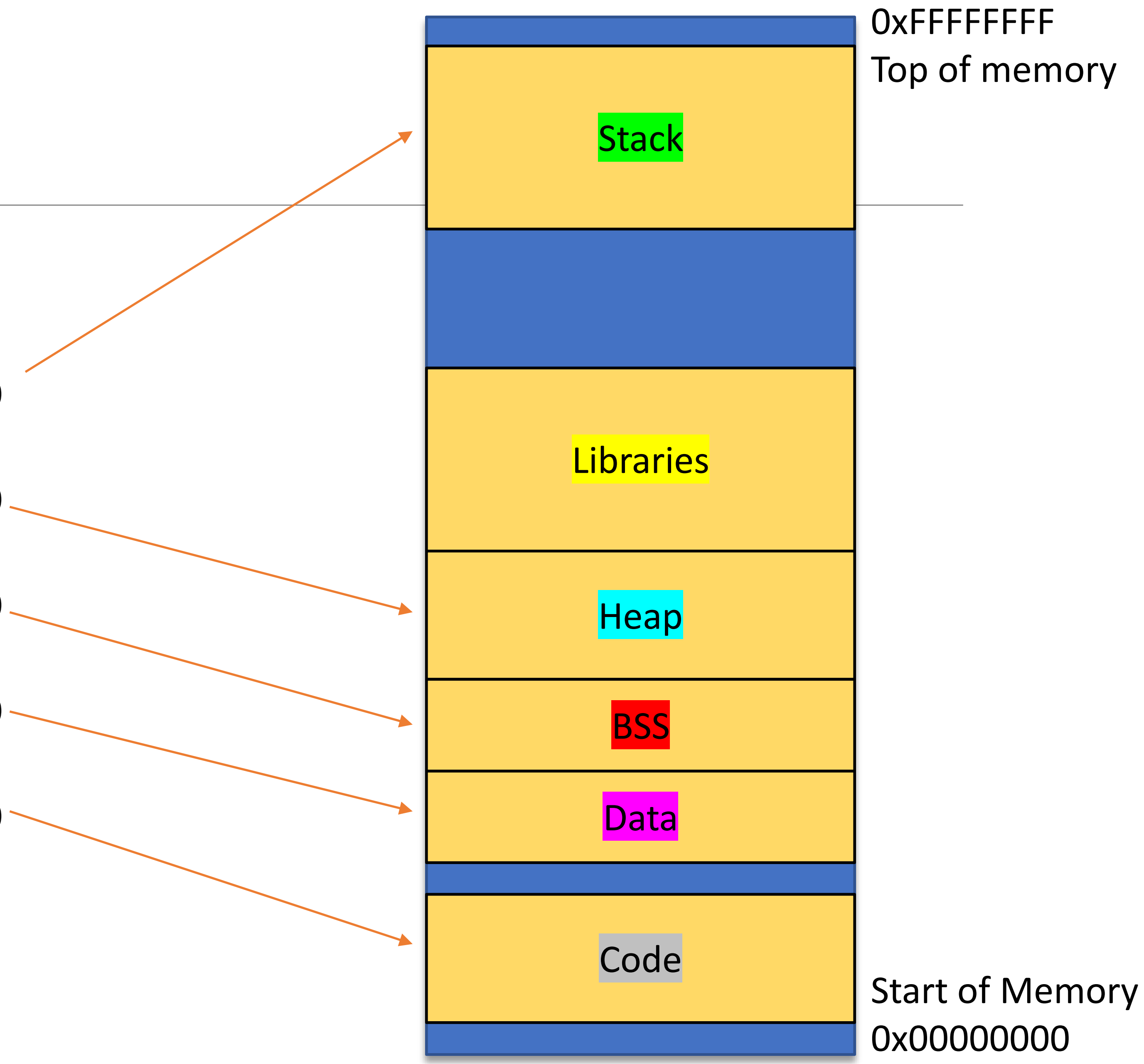


mem.c (available in course web page)

- **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

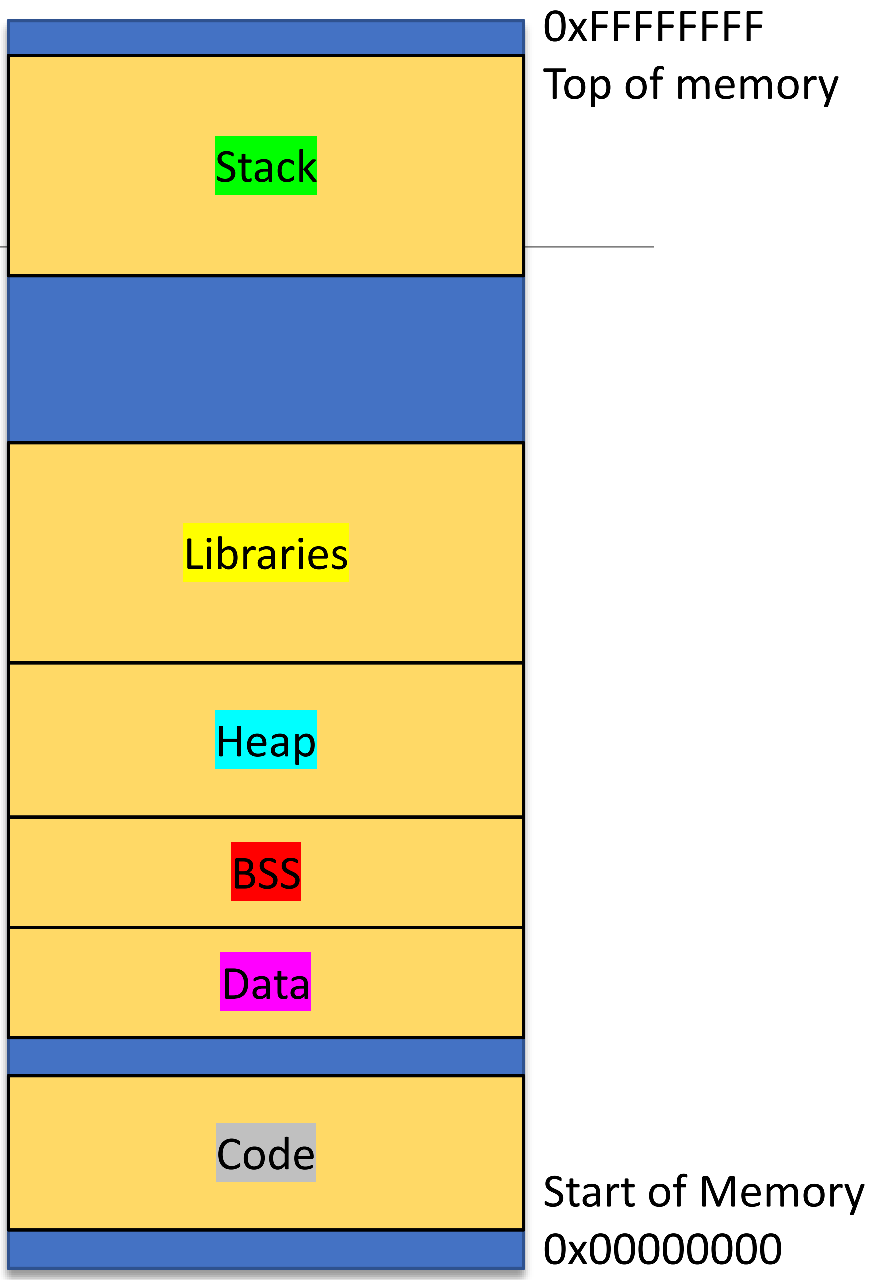
```
Internal Variables (Page = 4096)

&argc  = bfeb8590 -> stack = bfeb8000

malloc = 08435008 -> heap  = 08435000

bssvar = 0804a034 -> bss   = 0804a000

cntvar = 08048920 -> const = 08048000

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



mem.c

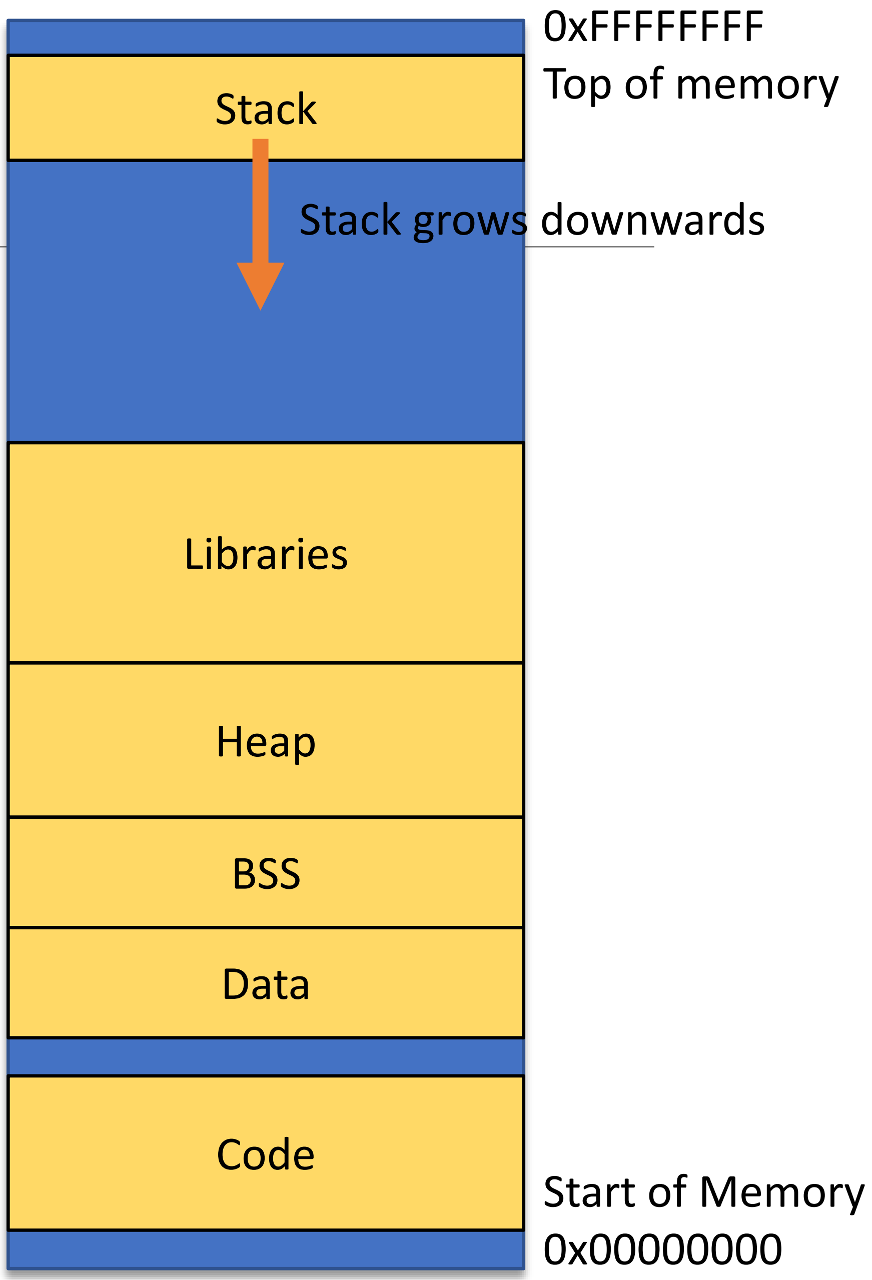
Content of /proc/self/maps

```
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]
```



mem.c

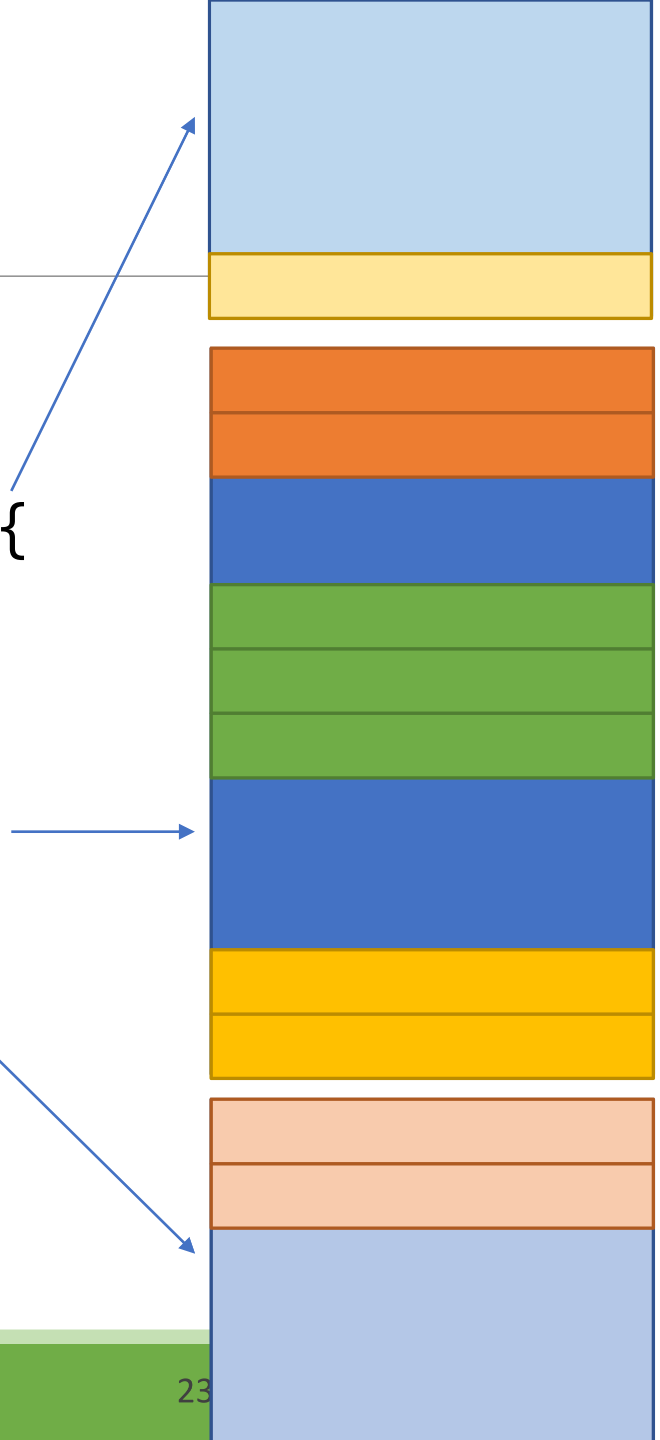
```
Stack evolution:
foo [000]: &argc  = bfeb8140 -> stack = bfeb8000
foo [001]: &argc  = bfdb8110 -> stack = bfdb8000
foo [002]: &argc  = bfcb80e0 -> stack = bfcb8000
foo [003]: &argc  = bfbb80b0 -> stack = bfbb8000
foo [004]: &argc  = bfab8080 -> stack = bfab8000
foo [005]: &argc  = bf9b8050 -> stack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



Stack organization

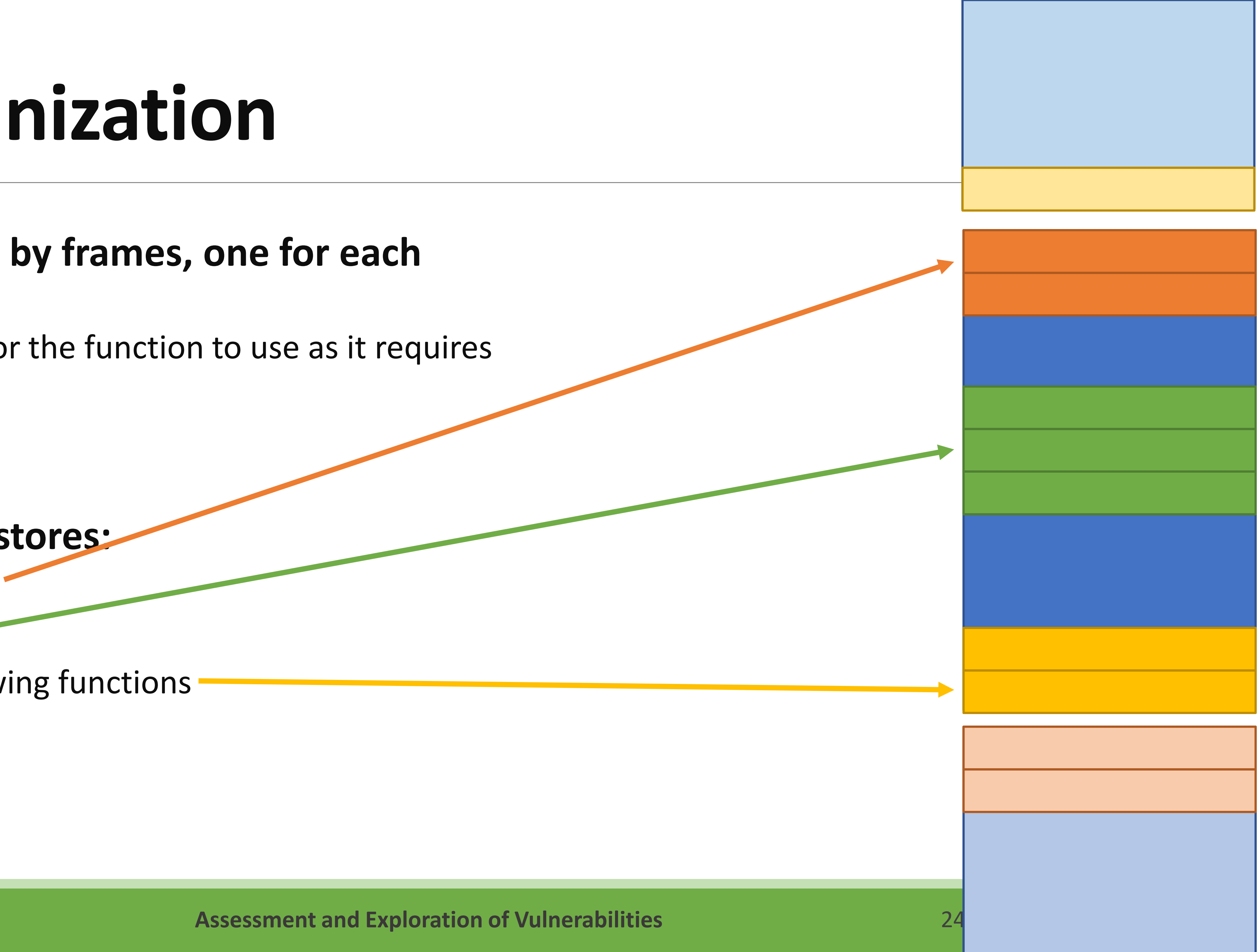
- **Stack is organized by frames, one for each function call**
  - Memory reserved for the function to use as it requires
- **Each stack frame stores:**
  - Return Information
  - Local Variables
  - Arguments to following functions (x32: all, x64: +5th)

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

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



Stack organization

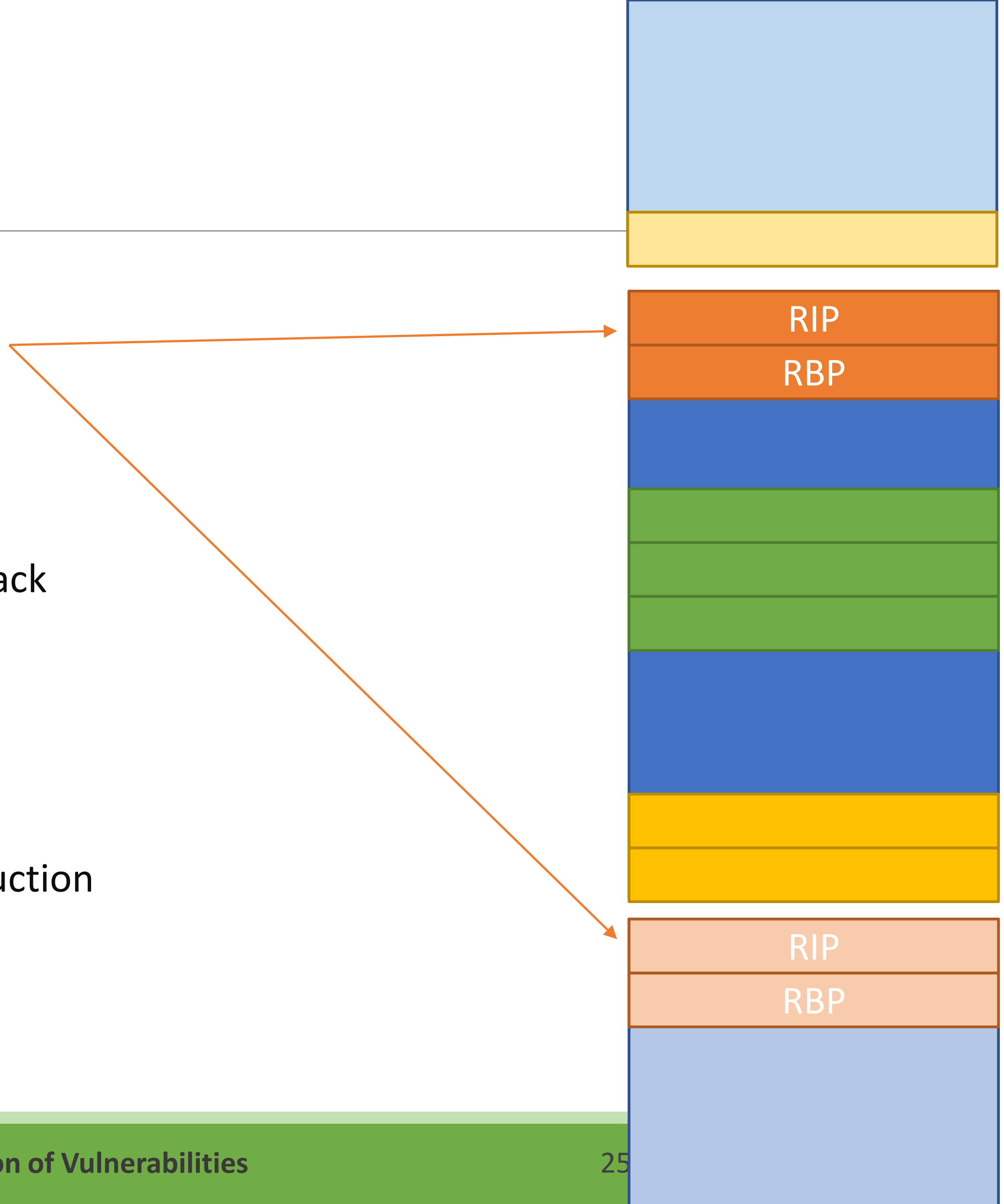
- **Stack is organized by frames, one for each function call**
  - Memory reserved for the function to use as it requires



- **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 it's 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
```

```
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
```

```
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

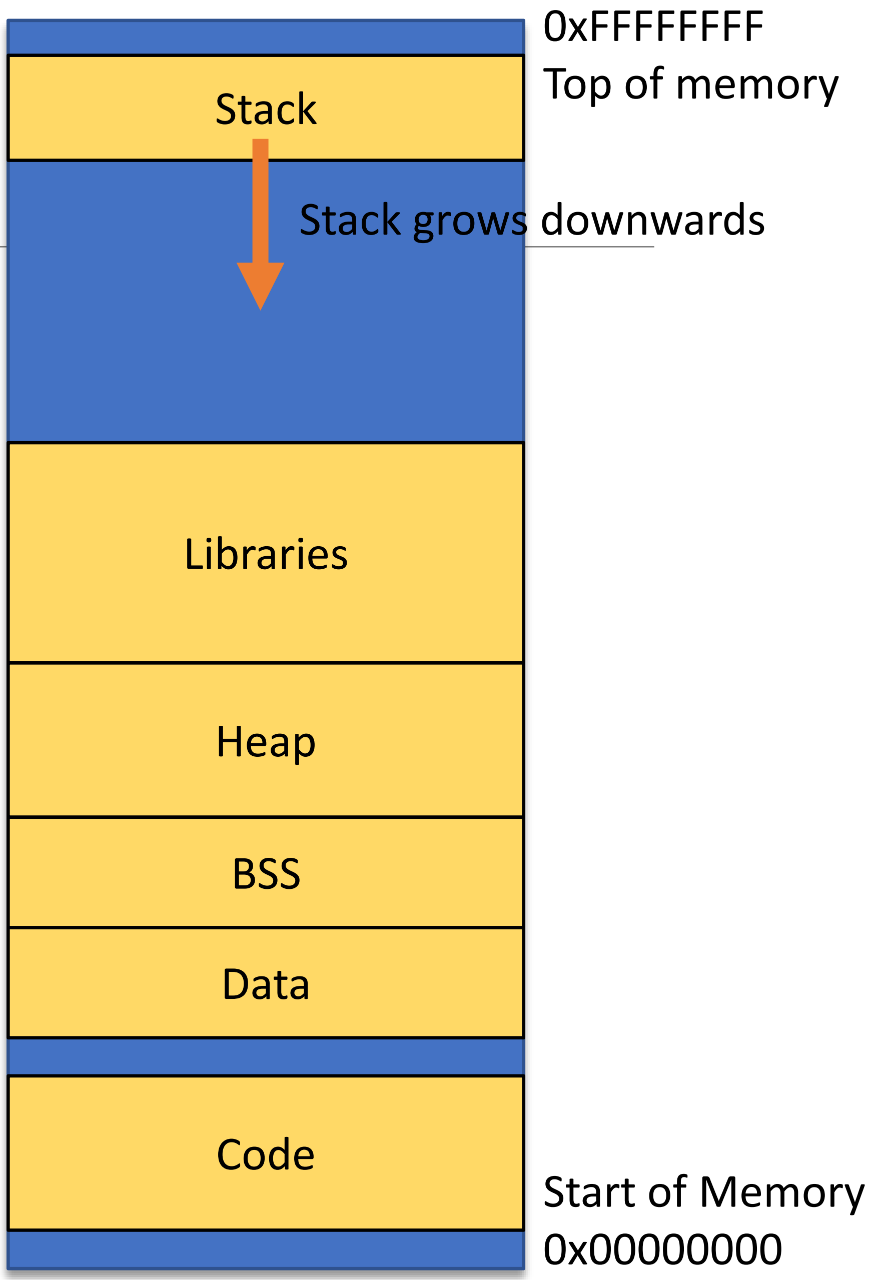
- **Declaration order doesn't matter!**
- **Compiler will place variables are he seems adequate**
  - Will keep information aligned
  - May create empty spaces
  - May deploy additional protection mechanisms (canaries)

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

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

```
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?

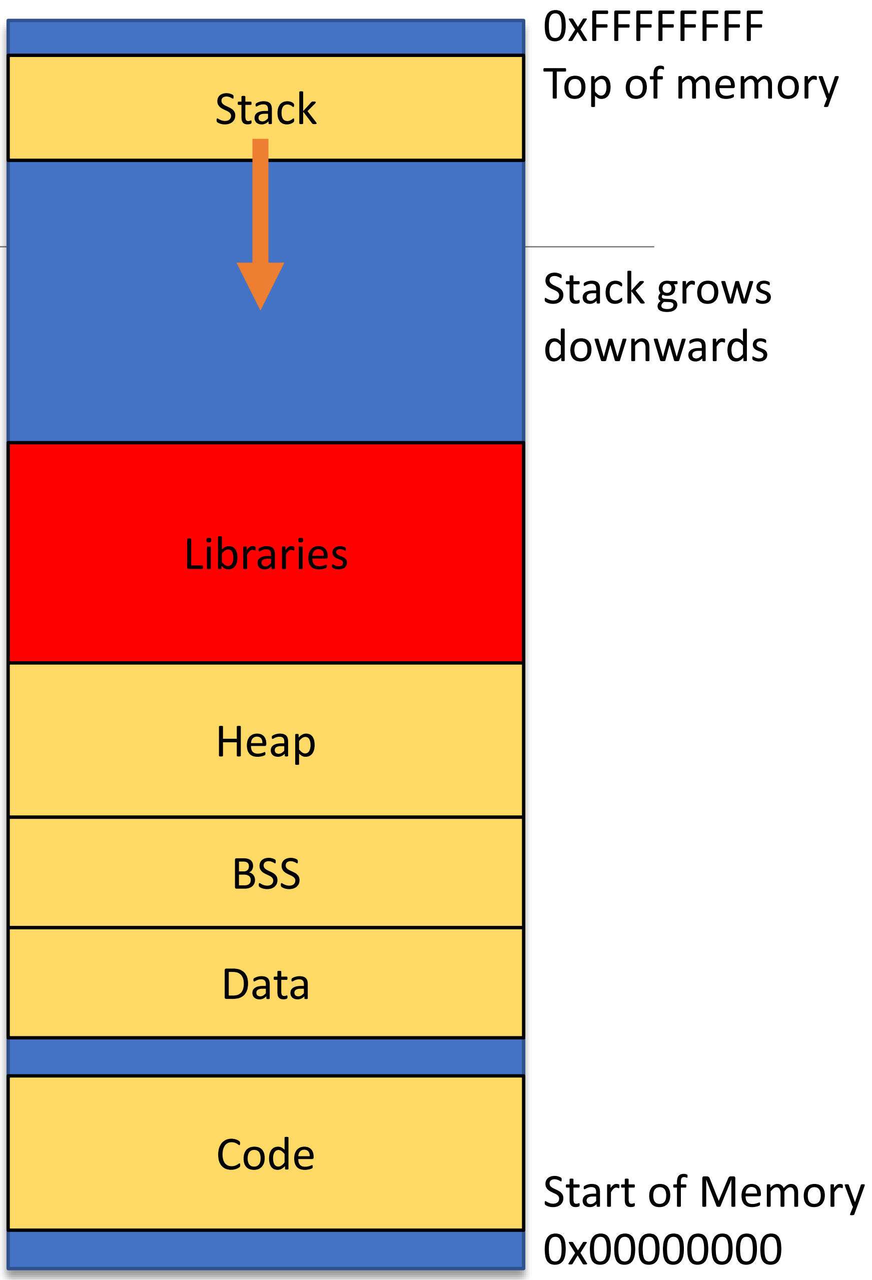


mem.c

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 devices**



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
    - Why 31 and not 32?
  - **`gets`** functions has no limit on input size
  - **`printf`** will print the content

```
//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:");
        gets(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

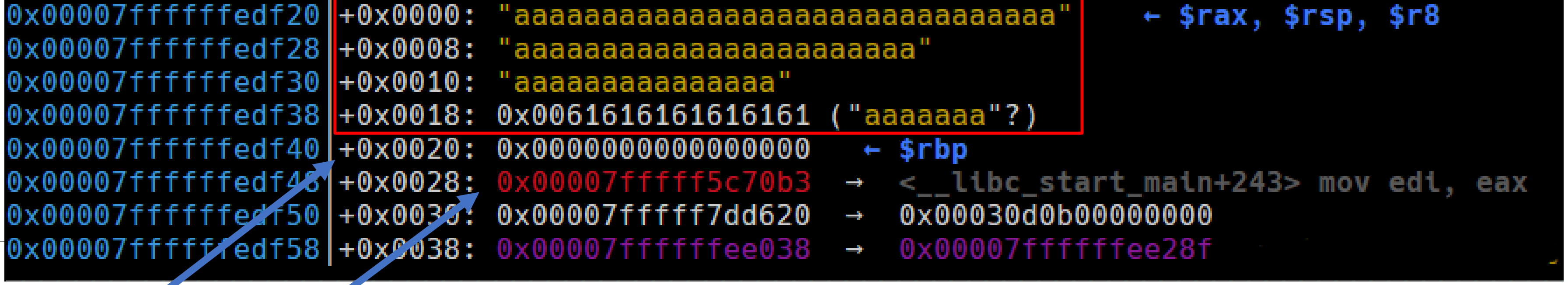
```
//classic/prog_1.c
//gcc -O0 -fno-stack-protector -o prog_1 prog_1.c

#include <stdio.h>

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

Exercise: classic/prog 1

- **Install gef: `pip3 install --user gdb-gef`**
- **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?

```
0x00007fffffffedf20|+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffedf28|+0x0008: 0x0000000000401090  →  <_start+0> endbr64
0x00007fffffffedf30|+0x0010: 0x00007fffffffee030  →  0x0000000000000001
0x00007fffffffedf38|+0x0018: 0x0000000000000000
0x00007fffffffedf40|+0x0020: 0x0000000000000000   ← $rbp
0x00007fffffffedf48|+0x0028: 0x00007ffff5c70b3  →  <__libc_start_main+243> mov edi, eax
0x00007fffffffedf50|+0x0030: 0x00007ffff7dd620  →  0x00030d0b00000000
0x00007fffffffedf58|+0x0038: 0x00007fffffffee038  →  0x00007fffffffee28f

     0x40117a <main+4>          push   rbp
     0x40117b <main+5>          mov    rbp, rsp
     0x40117e <main+8>          sub    rsp, 0x20
 →   0x401182 <main+12>         lea    rdi, [rip+0xe7b]        # 0x402004
     0x401189 <main+19>         call   0x401060 <puts@plt>
     0x40118e <main+24>         lea    rax, [rbp-0x20]
     0x401192 <main+28>         mov    rdi, rax
     0x401195 <main+31>         mov    eax, 0x0
     0x40119a <main+36>         call   0x401080 <gets@plt>

      1  #include <stdio.h>
      2
      3  int main() {
      4          char username[32];
          // username=0x00007fffffffedf20  →  0x0000000000000000
 →    5          puts("username:");
      6          gets(username);
      7          printf("Welcome %s!\n", username);
      8          return 0;
      9  }
     10

[#0] Id 1, Name: "prog_1", stopped 0x401182 in main (), reason: SINGLE STEP

[#0] 0x401182 → main()
```

Saved $BP
Saved $PC



What is the stack base address?

- `info frame:  0x7fffffffedf50`
- `p $rbp:  0x7fffffffedf40`

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?

- 32
- It could have been different due to empty space

What happens when an overflow is achieved?

- Saved **$BP** is overwritten and then Saved **$PC** is overwritten
- In this case, 31 ‘a’ were provided and an additional \0 was added at .. edf38

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

        puts("message:");
        gets(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

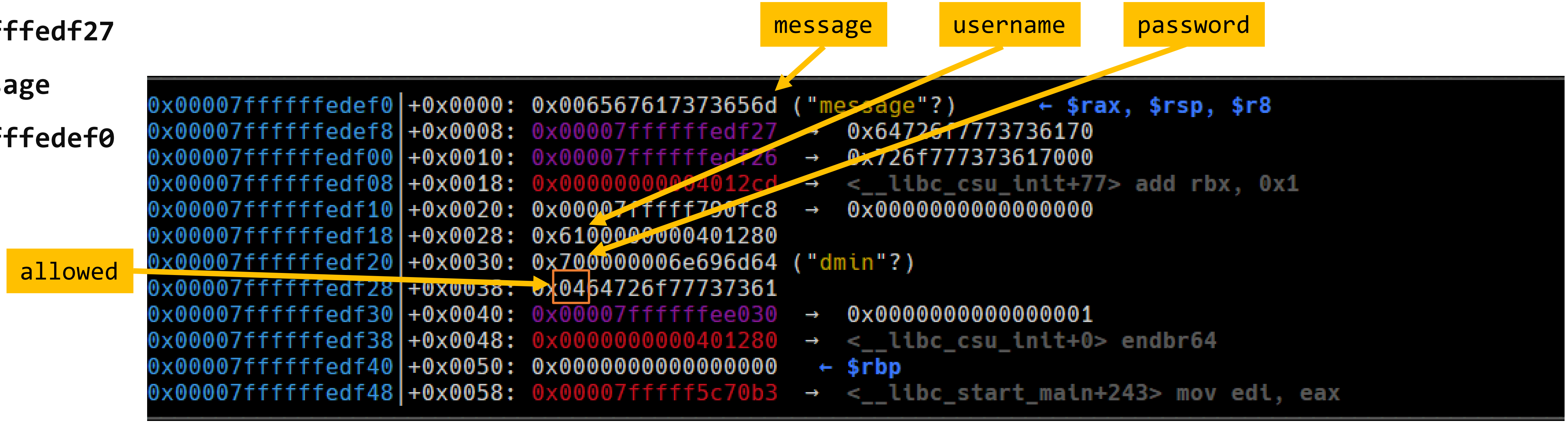
```
int main() {
        char allowed = 0;
        char password[8];
        char username[8];
        char message[32];

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

        puts("message:");
        gets(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
0x7fffffedf2f
p &username
0x7fffffedf1f
p &password
0x7fffffedf27
p &message
0x7fffffedef0
```

Memory grows from top to bottom

`message` can be used to overwrite everything!!!



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: ");
	gets(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

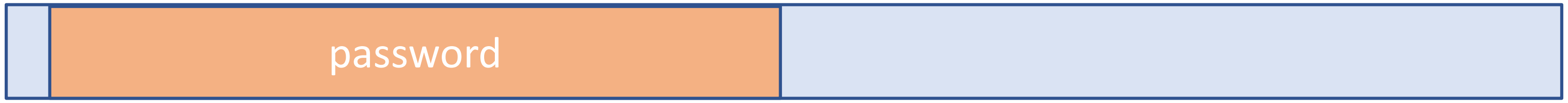
- **Vulnerability:**
  - In some situations, the password may overflow the buffer, and further memory operations erase the \0 character
  - Further `printf` of a message will include additional memory

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

	printf("Password: ");
	gets(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 erase end of string (`\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`**

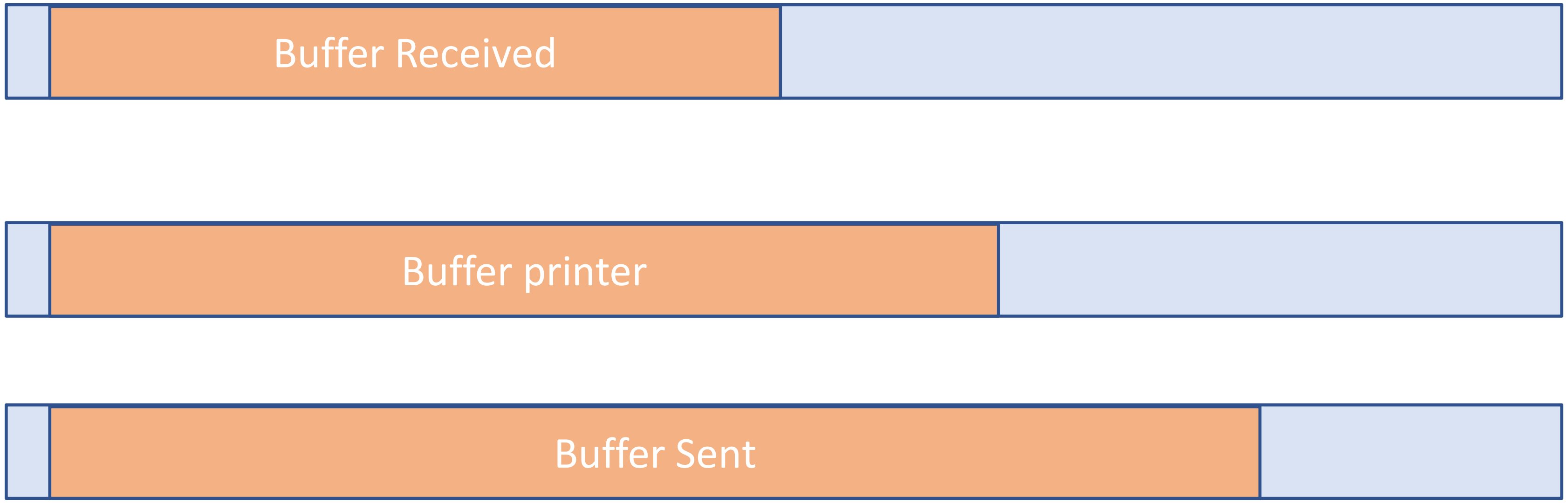
- **Impact: existing memory contents will be sent to clients**

```
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:

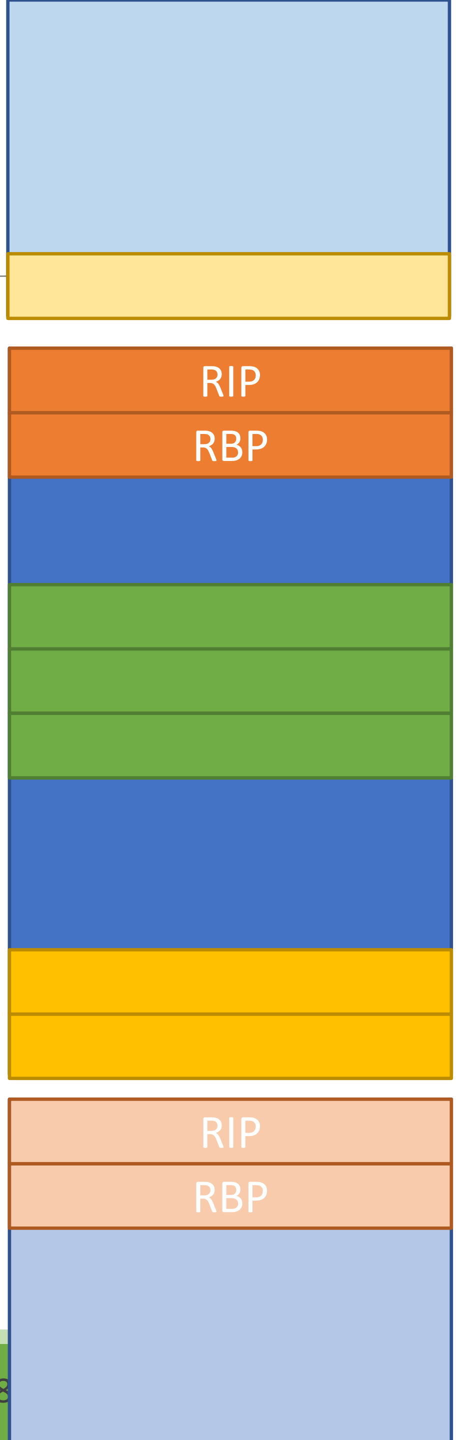


Stack Overflow

Stack Based Vulnerabilities

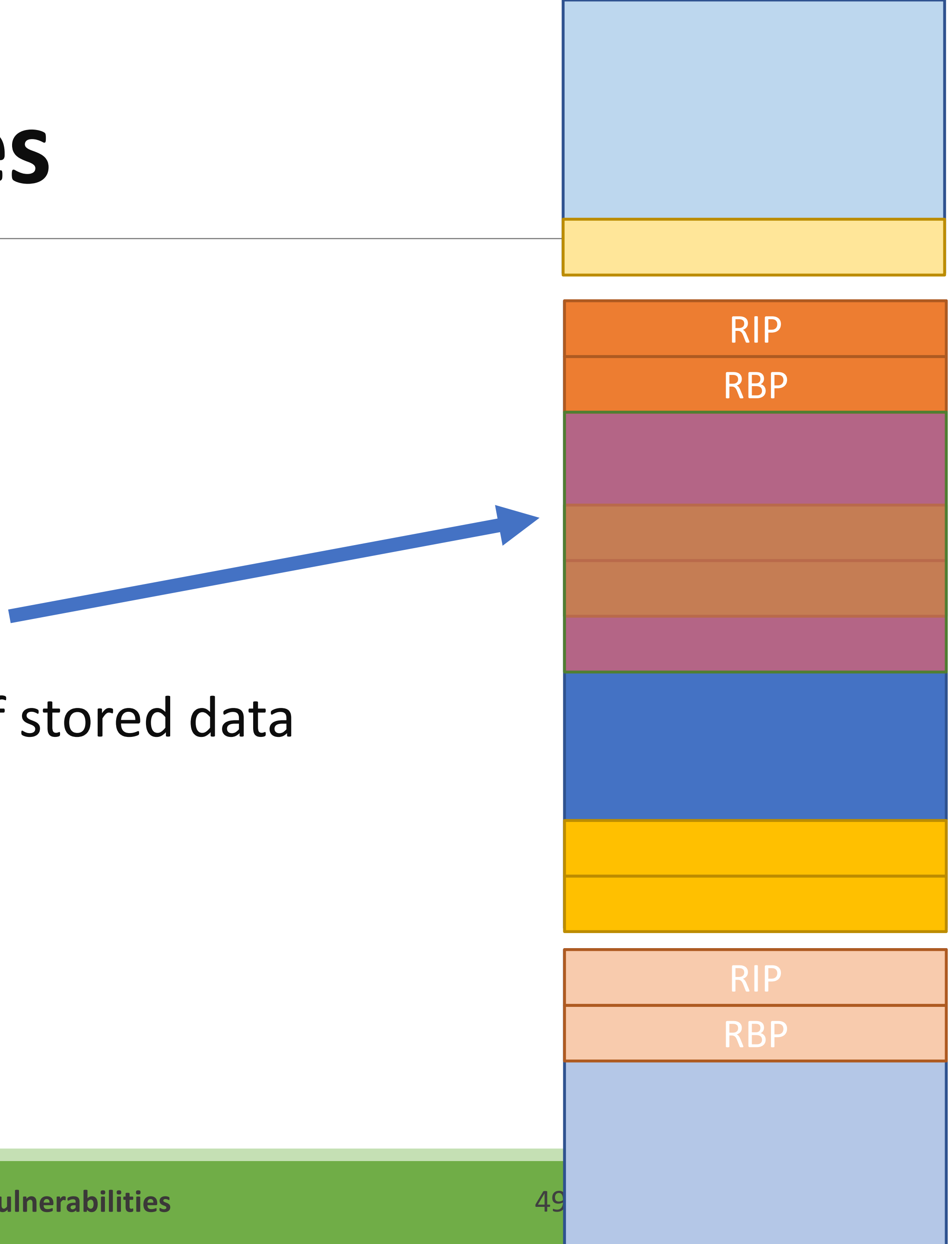
- **Stack can be subverted to conduct attacks**
  - it contains local variables (which store user injected data)
  - the program execution flow is kept in the stack

- **Mostly:**
  - Denial of Service: program crashes
  - Memory disclosure: attacker gains access to previous frames
  - Change program flow
  - Injection of malicious code



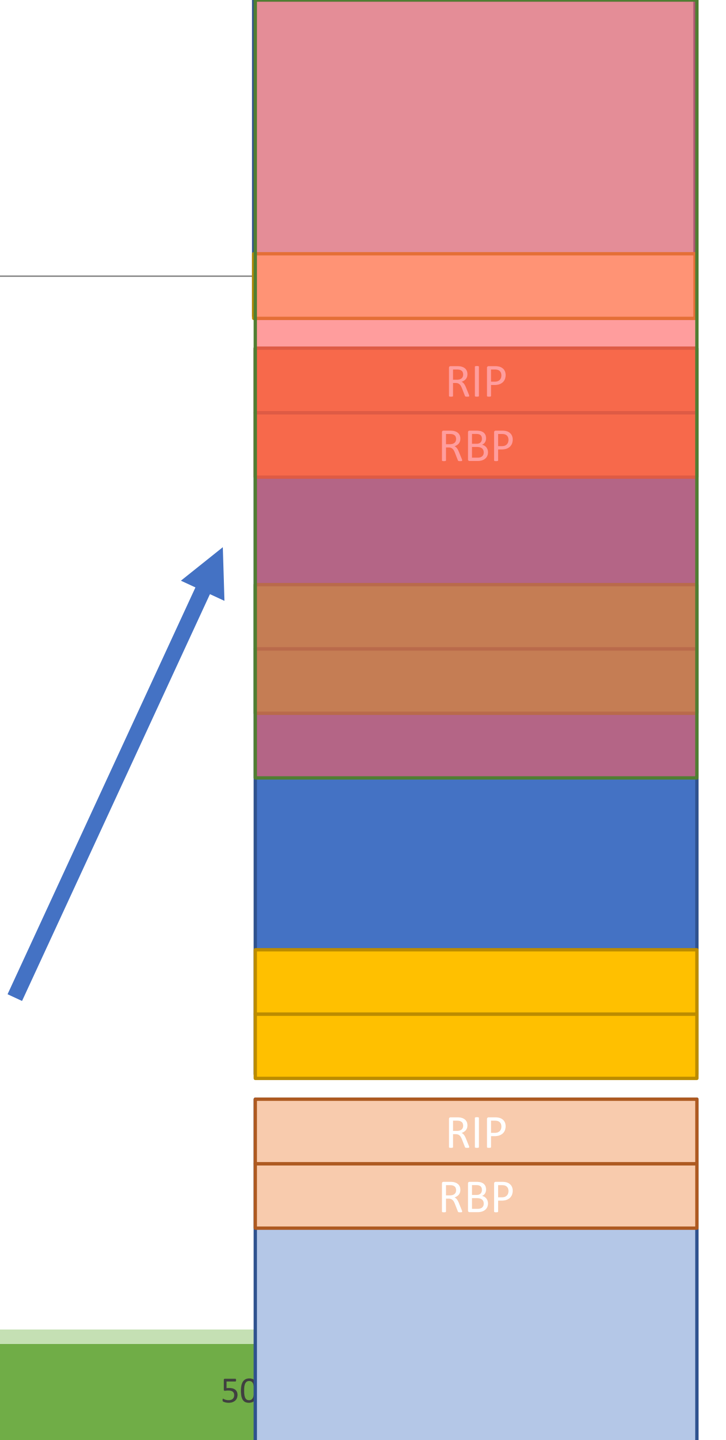
Stack Based Vulnerabilities

- **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**

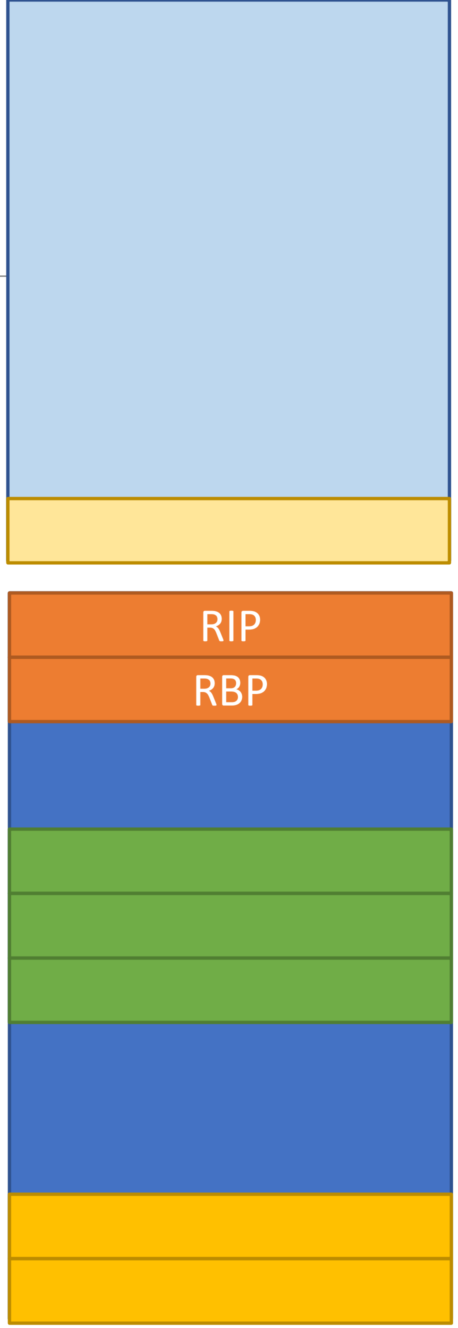


Stack Smashing

- **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 `SIGSEV`**



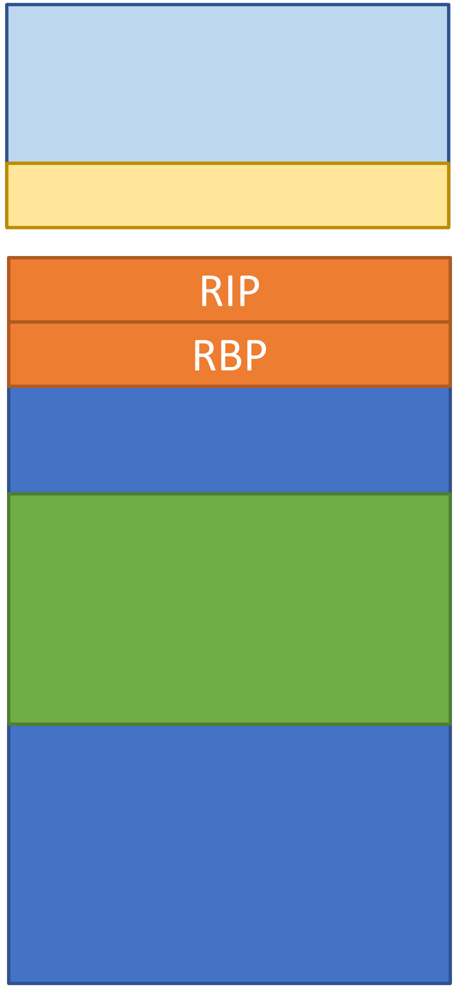
Stack: program_flow.c

- **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**

```
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

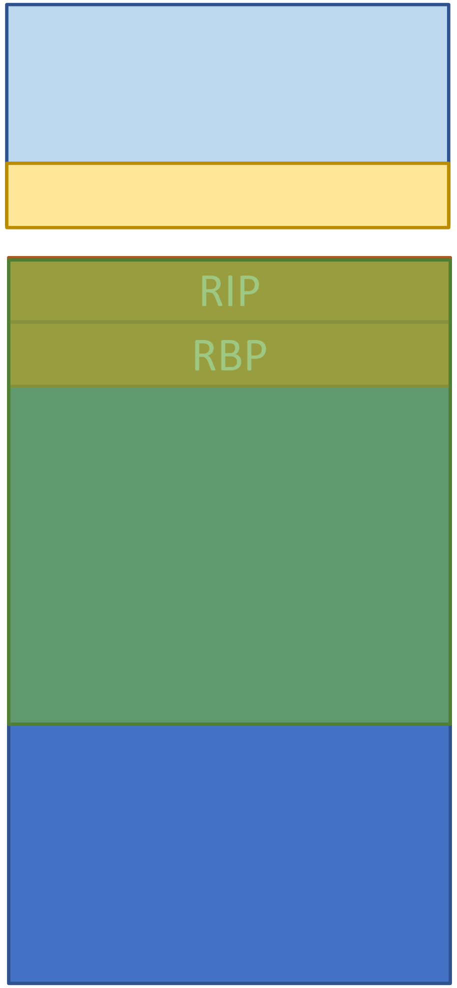
- **Main stack**
- **Foo stack**
  - Stored program flow
  - **buffer[8]**
- **Secret has no stack!**




```
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




```
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

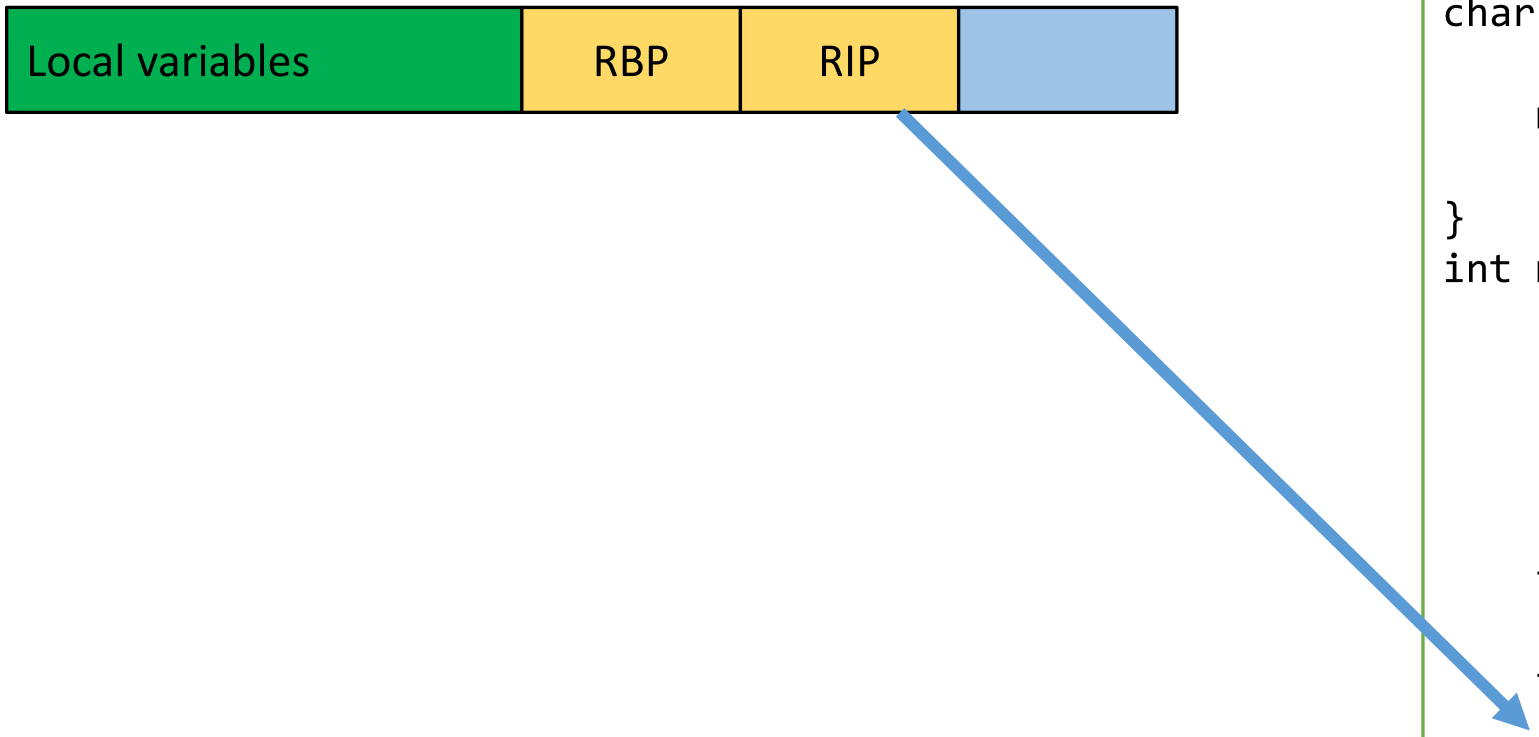
```
$ ./program_flow payload
0x8001209

$ 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!

```
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

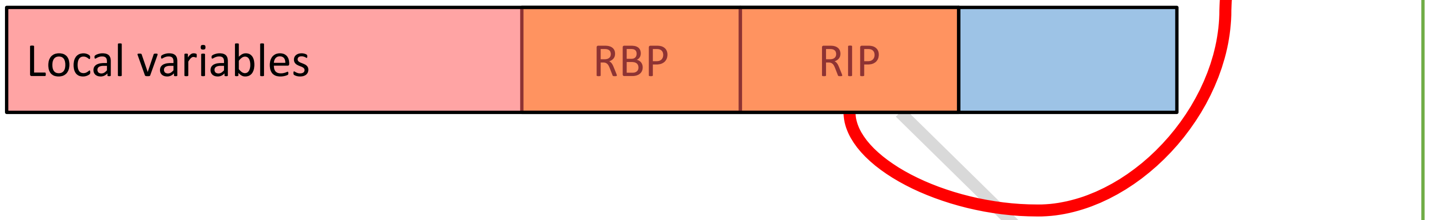
- **Typical flow**




```
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()**




```
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
```

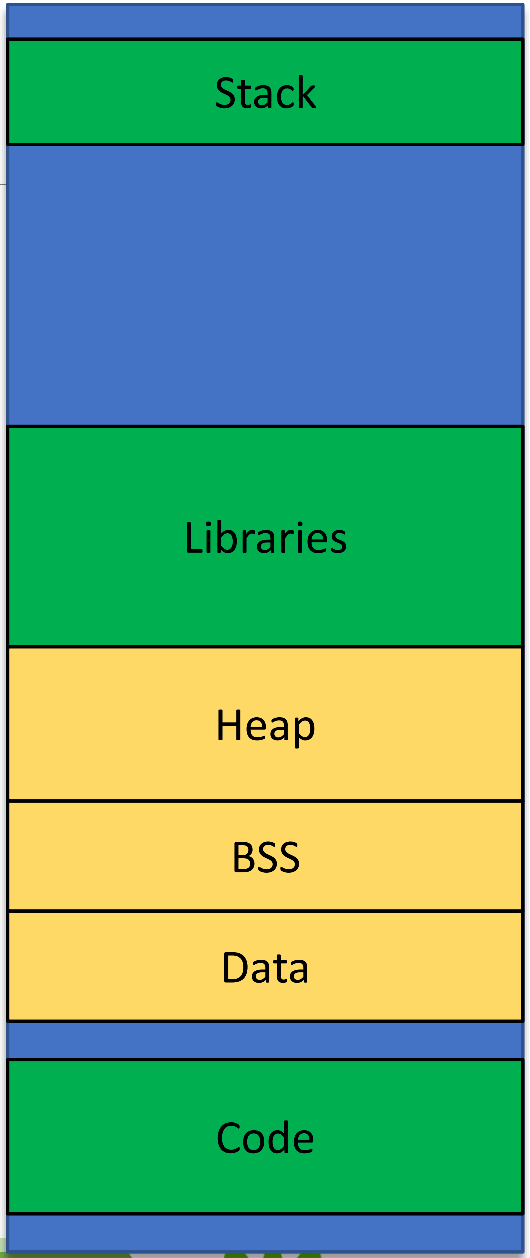
With the correct payload, secret() is called

```
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;
```

Q: What payload?

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**
  - In particular: system()
  - Is mostly executable
- **Stack can be executable**
  - but it isn’t on recent systems



Stack: return_to_libc.c

Typical Flow

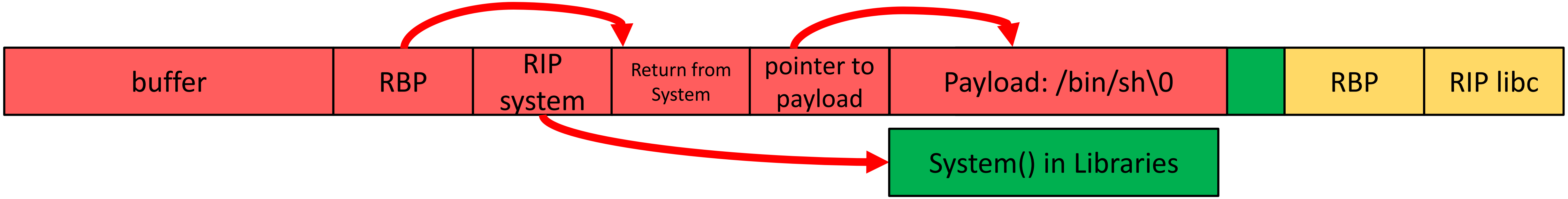
Local variables	RBP	RIP main	Function args	Local variables	RBP	RIP libc

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 payload
    - 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)
    - code cannot jump to it
    - Return to lib-c attack 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**
  - 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 what is in the stack but we cannot execute code from it
  - 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**
  - **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

ROP

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

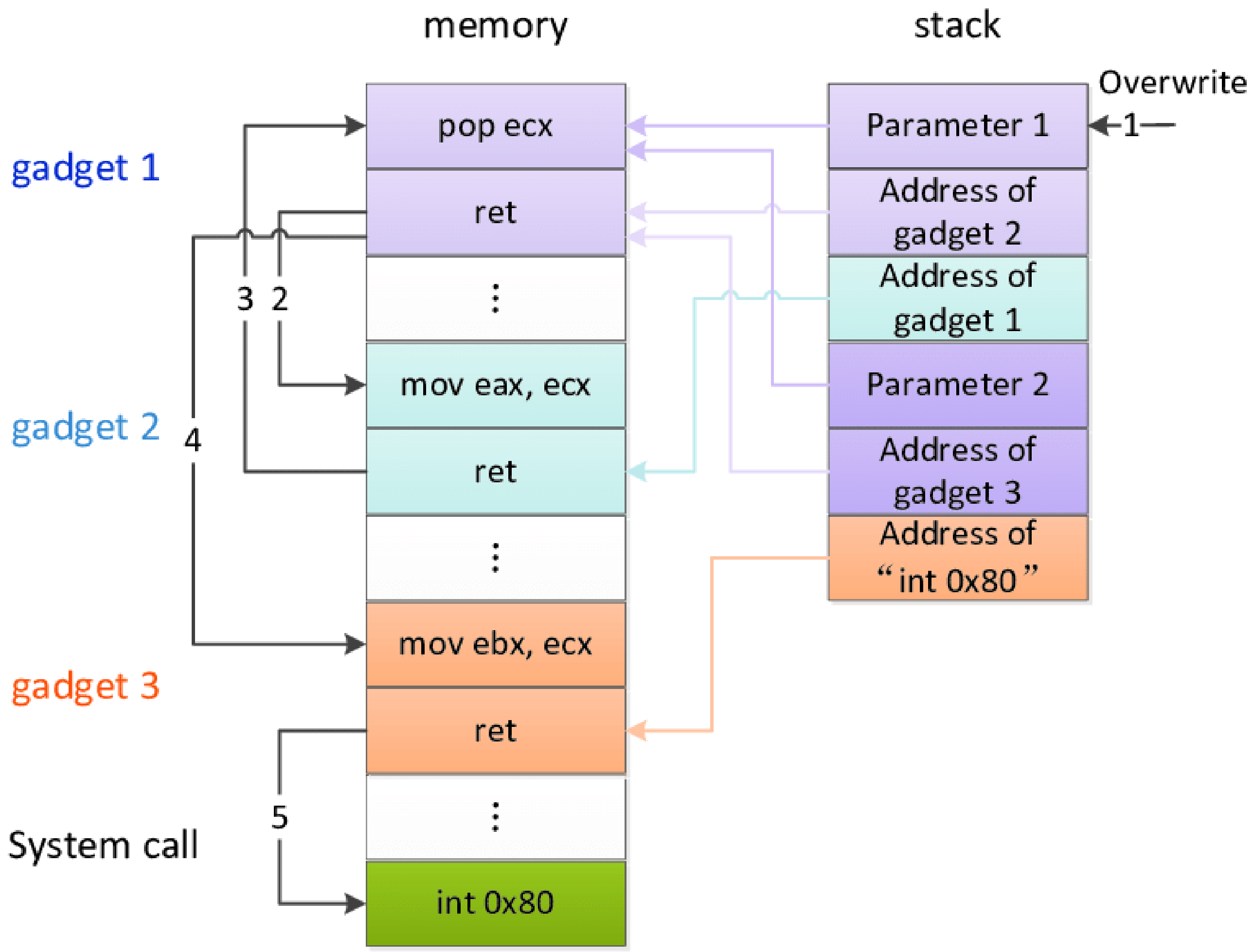
```
f7c7070000000f9545c3 → test edi, 0x7 ; setnz byte ptr [rbp-0x3d] ;
  c7070000000f9545c3 → mov dword ptr [rdi], 0xf000000 ; xchg ebp, eax ; ret
```

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

```
0x000000000040124c: mov rsi, rcx; mov rdi, rax; call 0x10e0; movzx eax, byte ptr [rbp - 8]; leave; ret;
0x0000000000401306: mov rsi, rdx; mov rdi, rax; call 0x1214; mov eax, 0; leave; ret;
0x0000000000401257: movzx eax, byte ptr [rbp - 8]; leave; ret;
0x0000000000401388: nop dword ptr [rax + rax]; endbr64; ret;
0x0000000000401387: nop dword ptr cs:[rax + rax]; endbr64; ret;
0x0000000000401386: nop word ptr cs:[rax + rax]; endbr64; ret;
0x0000000000401007: or byte ptr [rax - 0x75], cl; add eax, 0x2fe9; test rax, rax; je 0x1016; call rax;
0x0000000000401166: or dword ptr [rdi + 0x404060], edi; jmp rax;
0x000000000040137c: pop r12; pop r13; pop r14; pop r15; ret;
0x000000000040137e: pop r13; pop r14; pop r15; ret;
0x0000000000401380: pop r14; pop r15; ret;
0x0000000000401382: pop r15; ret;
0x000000000040137b: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x000000000040137f: pop rbp; pop r14; pop r15; ret;
0x00000000004011dd: pop rbp; ret;
0x0000000000401383: pop rdi; ret;
0x0000000000401381: pop rsi; pop r15; ret;
0x000000000040137d: pop rsp; pop r13; pop r14; pop r15; ret;
0x00000000004011cd: push rbp; mov rbp, rsp; call 0x1150; mov byte ptr [rip + 0x2e83], 1; pop rbp; ret;
0x00000000004012ea: ret 0xfffd;
0x0000000000401011: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x00000000004011d8: sub dword ptr [rsi], 0; add byte ptr [rcx], al; pop rbp; ret;
0x000000000040139d: sub esp, 8; add rsp, 8; ret;
0x0000000000401005: sub esp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x000000000040139c: sub rsp, 8; add rsp, 8; ret;
0x0000000000401004: sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x000000000040138a: test byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; endbr64; ret;
0x0000000000401010: test eax, eax; je 0x1016; call rax;
0x0000000000401010: test eax, eax; je 0x1016; call rax; add rsp, 8; ret;
0x0000000000401163: test eax, eax; je 0x1170; mov edi, 0x404060; jmp rax;
0x00000000004011a5: test eax, eax; je 0x11b0; mov edi, 0x404060; jmp rax;
0x000000000040100f: test rax, rax; je 0x1016; call rax;
0x000000000040100f: test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
0x0000000000401162: test rax, rax; je 0x1170; mov edi, 0x404060; jmp rax;
0x00000000004011a4: test rax, rax; je 0x11b0; mov edi, 0x404060; jmp rax;
```

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



```
0x0000000000401011: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x00000000004011d8: sub dword ptr [rsi], 0; add byte ptr [rcx], al; pop rbp; ret;
0x000000000040139d: sub esp, 8; add rsp, 8; ret;
0x0000000000401005: sub esp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x000000000040139c: sub rsp, 8; add rsp, 8; ret;
0x0000000000401004: sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x000000000040138a: test byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; endbr64; ret;
0x0000000000401010: test eax, eax; je 0x1016; call rax;
0x0000000000401010: test eax, eax; je 0x1016; call rax; add rsp, 8; ret;
0x0000000000401163: test eax, eax; je 0x1170; mov edi, 0x404060; jmp rax;
0x00000000004011a5: test eax, eax; je 0x11b0; mov edi, 0x404060; jmp rax;
0x000000000040100f: test rax, rax; je 0x1016; call rax;
0x000000000040100f: test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
0x0000000000401162: test rax, rax; je 0x1170; mov edi, 0x404060; jmp rax;
0x00000000004011a4: test rax, rax; je 0x11b0; mov edi, 0x404060; jmp rax;
0x000000000040125a: clc; leave; ret;
0x000000000040139b: cli; sub rsp, 8; add rsp, 8; ret;
0x0000000000401003: cli; sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x0000000000401143: cli; ret;
0x0000000000401398: endbr64; sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: endbr64; sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x0000000000401140: endbr64; ret;
0x000000000040113e: hlt; nop; endbr64; ret;
0x000000000040125b: leave; ret;
0x000000000040113f: nop; endbr64; ret;
0x000000000040116f: nop; ret;
0x000000000040101a: ret;

111 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


0x0000000000401383: pop rdi; ret;
```

Stack: return_to_libc.c (x86_64)

buffer	RBP	Gadget address	Command address	System address	Command to be executed (optional)		RBP	RIP libc

- **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`

```
0x000000000040100f: test rax, rax; je 0x1016; call rax;
0x000000000040100f: test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
0x0000000000401162: test rax, rax; je 0x1170; mov edi, 0x404060; jmp rax;
0x00000000004011a4: test rax, rax; je 0x11b0; mov edi, 0x404060; jmp rax;
0x000000000040125a: clc; leave; ret;
0x000000000040139b: cli; sub rsp, 8; add rsp, 8; ret;
0x0000000000401003: cli; sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x0000000000401143: cli; ret;
0x0000000000401398: endbr64; sub rsp, 8; add rsp, 8; ret;
0x0000000000401000: endbr64; sub rsp, 8; mov rax, qword ptr [rip + 0x2fe9]; test rax, rax; je 0x1016; call rax;
0x0000000000401140: endbr64; ret;
0x000000000040113e: hlt; nop; endbr64; ret;
0x000000000040125b: leave; ret;
0x000000000040113f: nop; endbr64; ret;
0x000000000040116f: nop; ret;
0x000000000040101a: ret;

111 gadgets found
gef➤  print system
$14 = {<text variable, no debug info>} 0x7fffff5f5410 <system>
gef➤  grep "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '[heap]'(0x405000-0x426000), permission=rw-
  0x4058b8 - 0x4058bf  →   "/bin/sh"
[+] In '/usr/lib/x86_64-linux-gnu/libc-2.31.so'(0x7fffff73d000-0x7fffff787000), permission=r--
  0x7fffff7575aa - 0x7fffff7575b1  →   "/bin/sh"
[+] In '(0x7fffff7df000-0x7fffff7e2000), permission=rw-
  0x7fffff7e1db0 - 0x7fffff7e1db7  →   "/bin/sh"
[+] In '[stack]'(0x7fffff7ef000-0x7ffffffef000), permission=rw-
  0x7ffffffedf30 - 0x7ffffffedf37  →   "/bin/sh"
  0x7ffffffedf58 - 0x7ffffffedf5f  →   "/bin/sh[...]"
gef➤
```

Stack: return_to_libc.c (x86_64)

buffer	RBP	Gadget address	Command address	System address	Command to be executed (optional)		RBP	RIP libc

- **Payload strategy:**
  - All addresses are 8 bytes
  - Buffer: padding with 16 bytes (buffer + RBP)
  - Gadget address: `0x00401383`
    - pop RDI: load command address into RDI
    - ret: load system address into RIP
  - Command address: `0x7ffff7575aa`
    - Approaches: **Find a string already in RAM (better);** add the payload after the system address (if required)
  - System address: `0x7ffff5f5410`

Stack: return_to_libc.c (x86_64)

buffer	RBP	Gadget1 address	Gadget2 address	Command address	System address	Command to be executed (optional)	RBP	RIP libc

- **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: `0x00401384 ; ret`
  - Gadget 2: `0x00401383 ; pop rdi;ret`

Stack: return_to_libc.c (x86_64)

- **Exercise: build a ROP chain and get a shell in the program**
  - 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

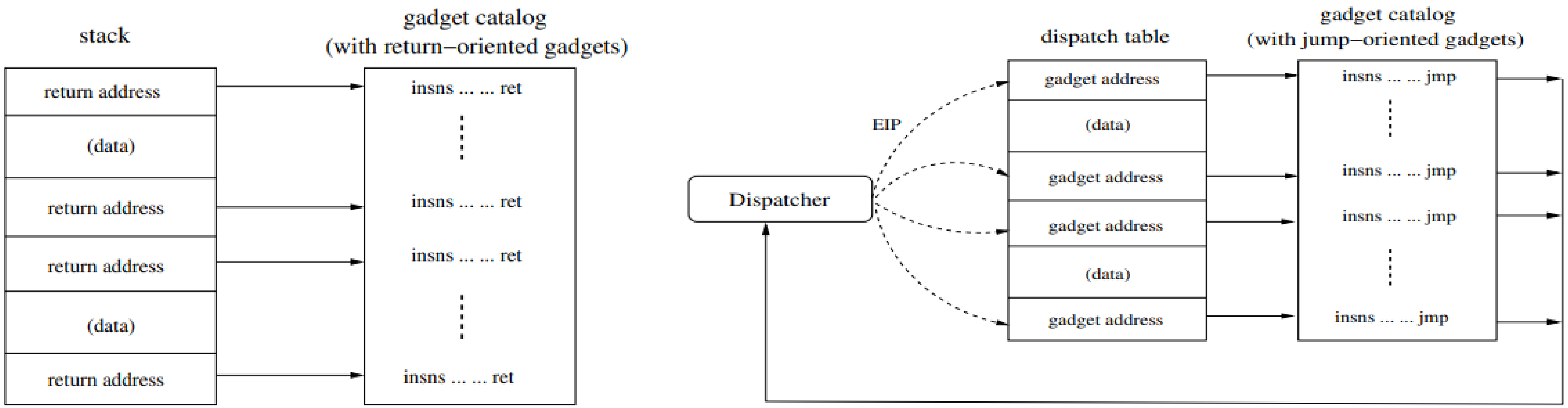
- **Exercise: build a ROP chain to start a remote shell**
  - Document the payload and the differences from the previous

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

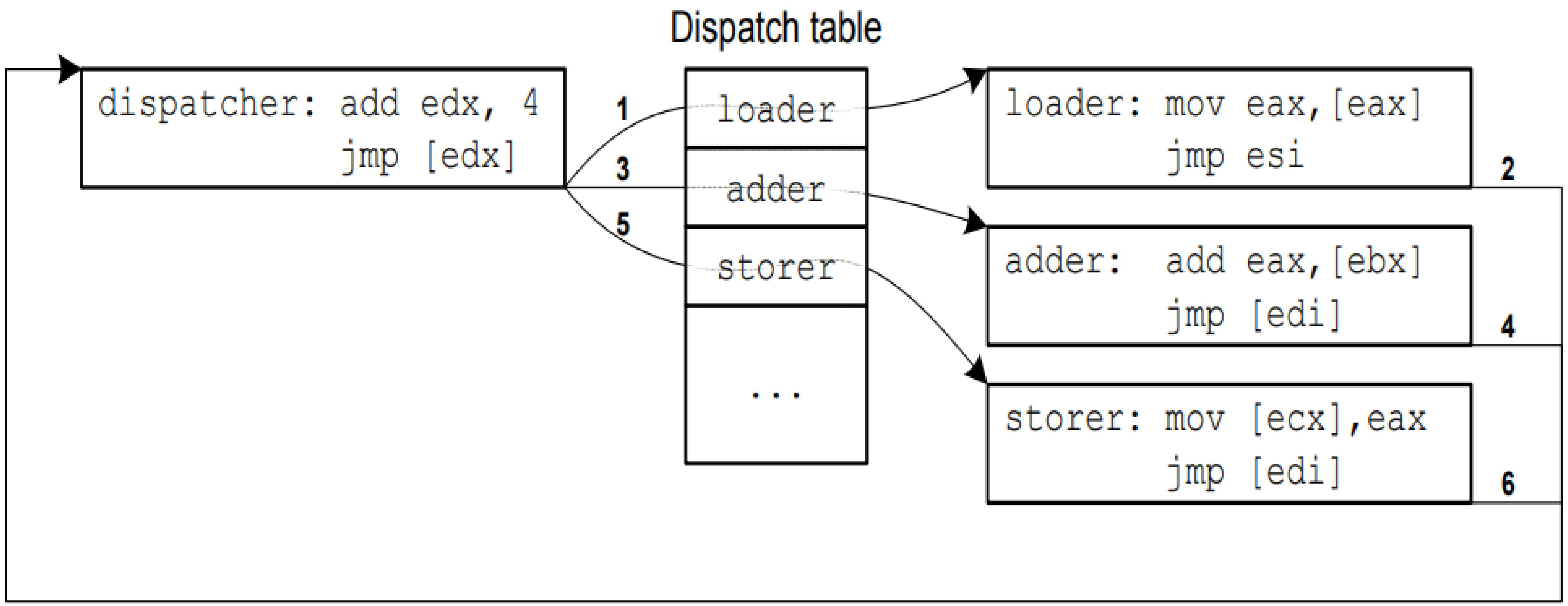
Jump Oriented Programming

- **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 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.

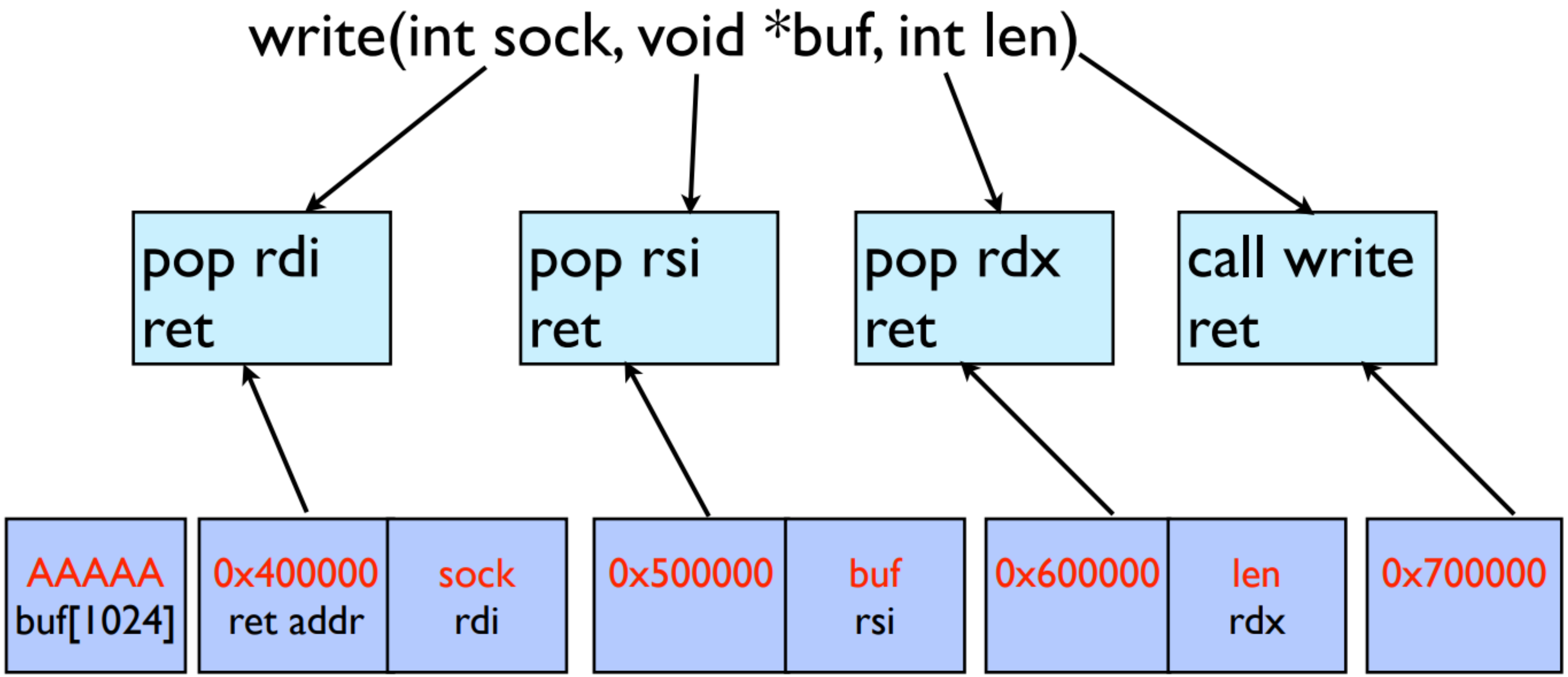
Blind Return Oriented Programming

- **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.

Blind Return Oriented Programming

- Looks for specific ROP Gadgets until a specific combination is found



Blind Return Oriented Programming

➢ **The BROP attack has the following phases:**

1. **Stack reading: read the stack to leak canaries and a return address to defeat ASLR.**
   Method: overflows varying the last byte. Byte found if app doesn't crash
   512-640 requests required

2. **Blind ROP: find enough gadgets to invoke write and control its arguments.**
   Method: find a Gadget1 that stops the service. Then brute force other gadgets together with this.
   Implement a clever method to identify different gadgets

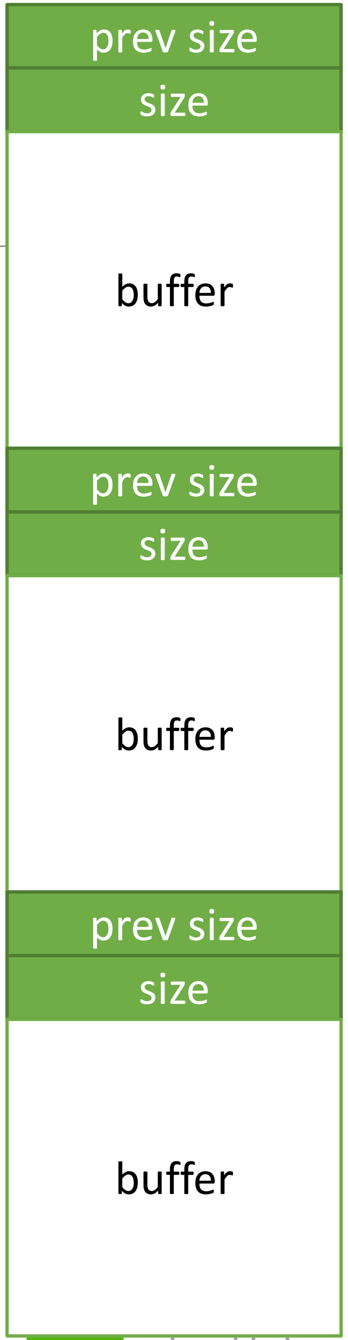
3. **Build the exploit: dump enough of the binary to find enough gadgets to build a shellcode, and launch the final exploit.**
   Obtain access to the write call so that the binary can be dumped

Heap Overflow

Heap Overflow

- **Heap is used to store dynamically allocated variables**
  - Allocation: malloc, calloc and new (C++), release: free or delete (C++)

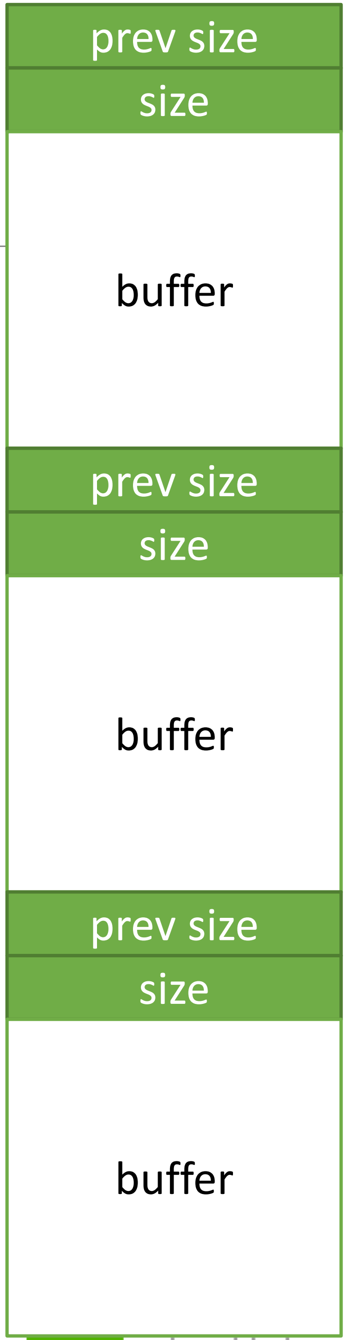
- **Call reserves a chunk and returns a pointer to the buffer**
  - buffer: (8 + (n / 8)*8 bytes)
    - If chunk is free data will have
      - Forward Pointer (4 bytes), pointer to next free chunk
      - Backwards Pointer (4 bytes), pointer to previous free chunk
  - Headers used for housekeeping
    - Previous Chunk Size (previous chunk is free), 4 bytes
    - Chunk Size + flags, 4 bytes
      - Flags
        - 0x01 PREV_INUSE – set when previous chunk is in use
        - 0x02 IS_MMAPPED – set if chunk was obtained with mmap()
        - 0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena



Heap Overflow: overflow.c

- **Overflow/underflow will write/read over control structures and then data**
  - Control structures are implementation specific
  - As well as reuse and actual buffer location

```
int main(int argc, char **argv) {
        char *buf1 = (char *) malloc(BUFSIZE);
        char *buf2 = (char *) malloc(BUFSIZE);
        memset(buf1, 0, BUFSIZE); //Clear data
        memset(buf2, 0, BUFSIZE);

        printf("Buf2: %s\n", buf2); //Should print "Buf2: "
        strcpy(buf1, argv[1]);
        printf("Buf2: %s\n", buf2); //Should print "Buf2: "
}
```



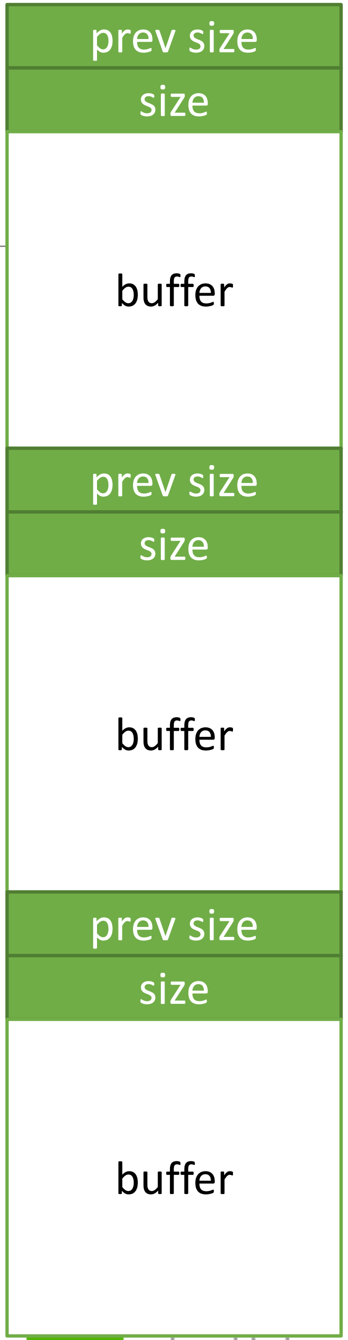
Heap Overflow: dangling.c

- **Dangling references can give access to memory**
  - Both for read and write purposes

```
char *buf1 = (char *) malloc(BUFSIZE*100); //Allocate buffer
memset(buf1, 'U', BUFSIZE);  //Fill it with 0x55
free(buf1); //Free the memory

char *buf2 = (char *) malloc(BUFSIZE); //Allocate new buffer
memset(buf2, 'A', BUFSIZE);  //Fill it with 0x41

printf("%s\n", buf1); //buf1 was freed
```

- **Access to buf1 should be denied: it isn't**

- **Access to buf1 should not give access to other ranges: it gives to buf2**



Heap Overflow: fastbin.c

- **Glibc has lists of recently freed blocks**
  - Each list (bin) stores chunks with a specific size
  - Blocks are reused in future allocations if size is compatible
    - Great for performance as the memory is already reserved
    - Horrible for security as dangling pointers will give a view to memory areas

- **Bins are also used to detect double free**
  - We cannot free a chunk that rests at the top of the bin
  - Which is great for security as a double free could corrupt the linked list

Heap Overflow: fastbin.c

- **Fast Bin attack explores Bins to get a pointer to an already allocated area**
  - Result is program will have **two pointers to the same memory**
    - Especially useful if memory stores dynamic objects with function, as function pointers can be overwritten
  - The first pointer is legitimate
  - The second is a shadow pointer

- **Attack strategy**
  - Allocate at least three buffers (a, b, c) with the same size
    - To use same bin
  - free(a), then free(b), then free(a) again
    - Double freeing a will ensure that the fast bin will have duplicated entries (a)
    - Bin will have three pointers ready to use: a b a
  - Allocate three buffers again with the same size.
    - Result is a legitimate pointer, another legitimate pointer, and a shadow pointer

Heap Overflow: fastbin.c

- **Impact: attacker can gain access to memory region**
  - If victim has chunk a with data and leaks
  - Attacker can fill free list and allocate again

```
// Allocating 3 buffers
int *a = calloc(1, 8);
int *b = calloc(1, 8);
int *c = calloc(1, 8);

free(a);
free(b);
free(a); //AGAIN!

//Free list now has: a b a

int *d = calloc(1, 8);
int *e = calloc(1, 8);
int *f = calloc(1, 8);

// d will be equal to f
```

Heap Overflow: overflow.c

- **Exercise: Observe and document the behavior in both programs**
  - dangling.c and overflow.c
  - Use GDB to analyse the addresses
  - What is the impact of writing to a freed pointer?

Countermeasures: ASLR

- **Address Space Layout Randomization (ASLR)**
  - Address are dynamic across process execution
    - Different architectures and configurations apply randomization to different segments
    - Only Stack is randomized, all segments are randomized
  - Not trivial to predict the address to issue a jump or change memory
- **echo $n > /proc/sys/kernel/randomize_va_space**
  - 0 = No randomization
  - 1 = Conservative Randomization: Stack, Heap, Shared Libs
  - 2 = Full Randomization: 1 + memory managed via brk())

Effects of ASLR (WSL1 on Windows 10)

- randomize_va_space =2

```
main: 0x7f80def82189, argc: 0x7fffbfce569c, local: 0x7fffbfce56ac, heap: 0x7fffb8c4b2a0, libc: 0x7f80ded85410
main: 0x7fb811d47189, argc: 0x7fffdbd2928c, local: 0x7fffdbd2929c, heap: 0x7fffd47952a0, libc: 0x7fb811b55410
main: 0x7f95178f0189, argc: 0x7fffee962b7c, local: 0x7fffee962b8c, heap: 0x7fffe67082a0, libc: 0x7f95176f5410
```

- randomize_va_space =1

```
main: 0x7f1672f77189, argc: 0x7fffe5835f0c, local: 0x7fffe5835f1c, heap: 0x7f1672f7b2a0, libc: 0x7f1672d85410
main: 0x7f6f0aed0189, argc: 0x7fffd8eb4e9c, local: 0x7fffd8eb4eac, heap: 0x7f6f0aed42a0, libc: 0x7f6f0acd5410
main: 0x7f8106545189, argc: 0x7ffff8601bdc, local: 0x7ffff8601bec, heap: 0x7f81065492a0, libc: 0x7f8106355410
```

- randomize_va_space=0

```
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
main: 0x8001189, argc: 0x7ffffffee0ec, local: 0x7ffffffee0fc, heap: 0x80052a0, libc: 0x7fffff5f5410
```

Coutermeasures: PIE

- **Position Independent Executables**
  - Executables compiled such that their base address does not matter, 'position independent code'
- **PIE fully enables ASLR as code can be placed dynamically**
  - Must be enabled at compile time!!
    - gcc –pie –fPIE
- **Breaking ASLR and PIE: Find a reference to some known function**
  - Because while addresses change, the change keeps relative distance
  - e.g.: if we know printf is at 0xbf00332, we will know where is system.

ASLR and relative offsets

```
main: 0x7f80def82189, argc: 0x7fffbfce569c
main: 0x7fb811d47189, argc: 0x7fffdbd2928c
main: 0x7f95178f0189, argc: 0x7fffee962b7c

local: 0x7fffbfce56ac, heap: 0x7fffb8c4b2a0
local: 0x7fffdbd2929c, heap: 0x7fffd47952a0
local: 0x7fffee962b8c, heap: 0x7fffe67082a0

libc: 0x7f80ded85410
libc: 0x7fb811b55410
libc: 0x7f95176f5410
```