

Software Security 1

Administrative

Kevin Borgolte

kevin.borgolte@rub.de

Tentative Lecture Schedule / Deadlines

Lectures

Wednesday 10-12

1. Oct 9 – Lecture
2. Oct 16 – Flipped classroom
3. Oct 23 – Lecture
4. Oct 30 – Flipped classroom
5. Nov 6 – Lecture
6. Nov 13 – Flipped classroom
7. Nov 20 – Lecture
8. Nov 27 – Flipped classroom
9. Dec 4 – Lecture
10. Dec 11 – Flipped classroom
11. Dec 18 – Guest? Lecture
12. Jan 8 – Lecture
13. Jan 15 – Flipped classroom
14. Jan 22 – Lecture
15. Jan 29 – Flipped classroom

← First assignment due

← Second assignment due

← Third assignment due

← Fourth assignment due

← Fifth assignment due

← Sixth assignment due

Tentative
(will probably change)


Assignments

- Assignments
 - All online: <https://scoreboard.ws24.softsec.rub.de/>
 - Questions: Moodle or emails us (softsec+teaching@rub.de)
- Assignment 1:
 - 6 tasks (quiz, fizzbuzz, peek, peeky, seashells, deprecated)
 - Due: October 24th, 2024 0:00 (Bochum time)
- Assignment 2:
 - 6 tasks (or something like that)
 - Due: November 7th, 2024 0:00 (Bochum time)

Exercise Sessions

- Two sessions (you've been assigned a session on Moodle)
 - Thursday 10-12
 - Thursday 12-14
- Always in MC 5/222
 - North elevator, north side
 - Ring the bell if the door is closed
- TAs
 - Felipe Novais (main TA)
 - Tobias Holl (backup TA)

- Local CTF team (FluxFingers) has a beginners group
 - FluxRookies
- Every Thursday 16:00
 - Starts November 14th, 2024
- Meets in ID 03/445
- Check out their Discord
 - <https://fluxfingers.net/discord>

The logo for XUJ FINGERS, featuring the text 'XUJ' in blue and 'FINGERS' in green, both in a stylized, blocky font, enclosed in a blue and green border.

🇩🇪 Germany				
Worldwide position	Country position	Name	Points	Events
11	👑 1	FluxFingers	656.539	16
28	2	KITCTF	451.537	19
33	3	ENOFLAG	428.963	17
41	4	FAUST	395.409	19
68	5	Platypwnies	327.526	13
87	6	h4tum	296.194	21
90	7	CyberTaskForce Zero	288.543	15

Topics Today

- Basic Defenses
 - This will be turned on for future assignments
- Programs, Loading and Processes
- Format String Vulnerabilities
- Return-oriented Programming

Software Security 1

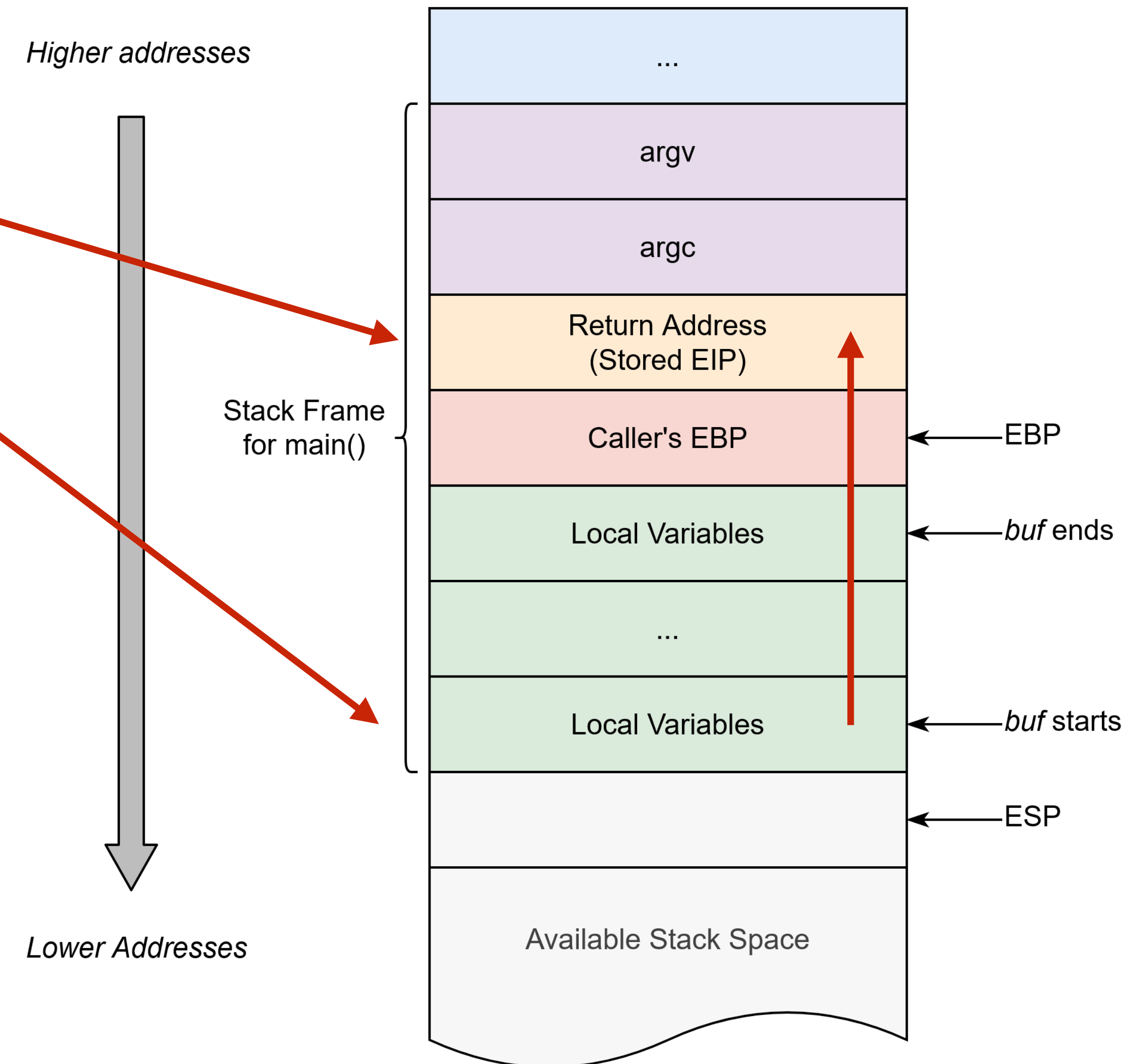
Basic Defenses

Kevin Borgolte

kevin.borgolte@rub.de

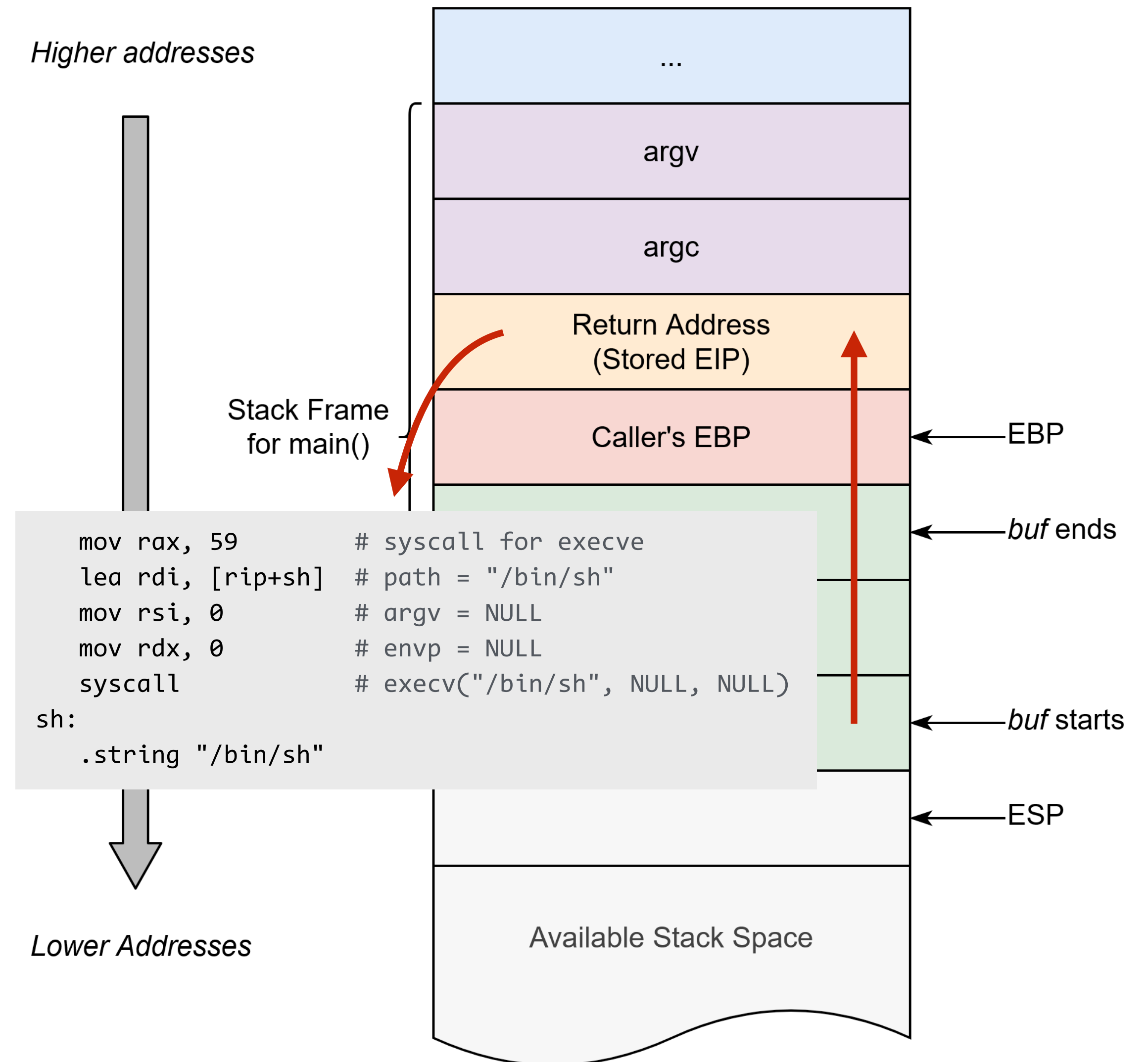
Recap: Stack-based Buffer Overflow

- Overwrite return address of a function in a program
 - Some buffer is on the stack
 - Stack grows down
 - Buffer is at a lower address than return address
 - Writing more means overwriting return address
 - Code execution



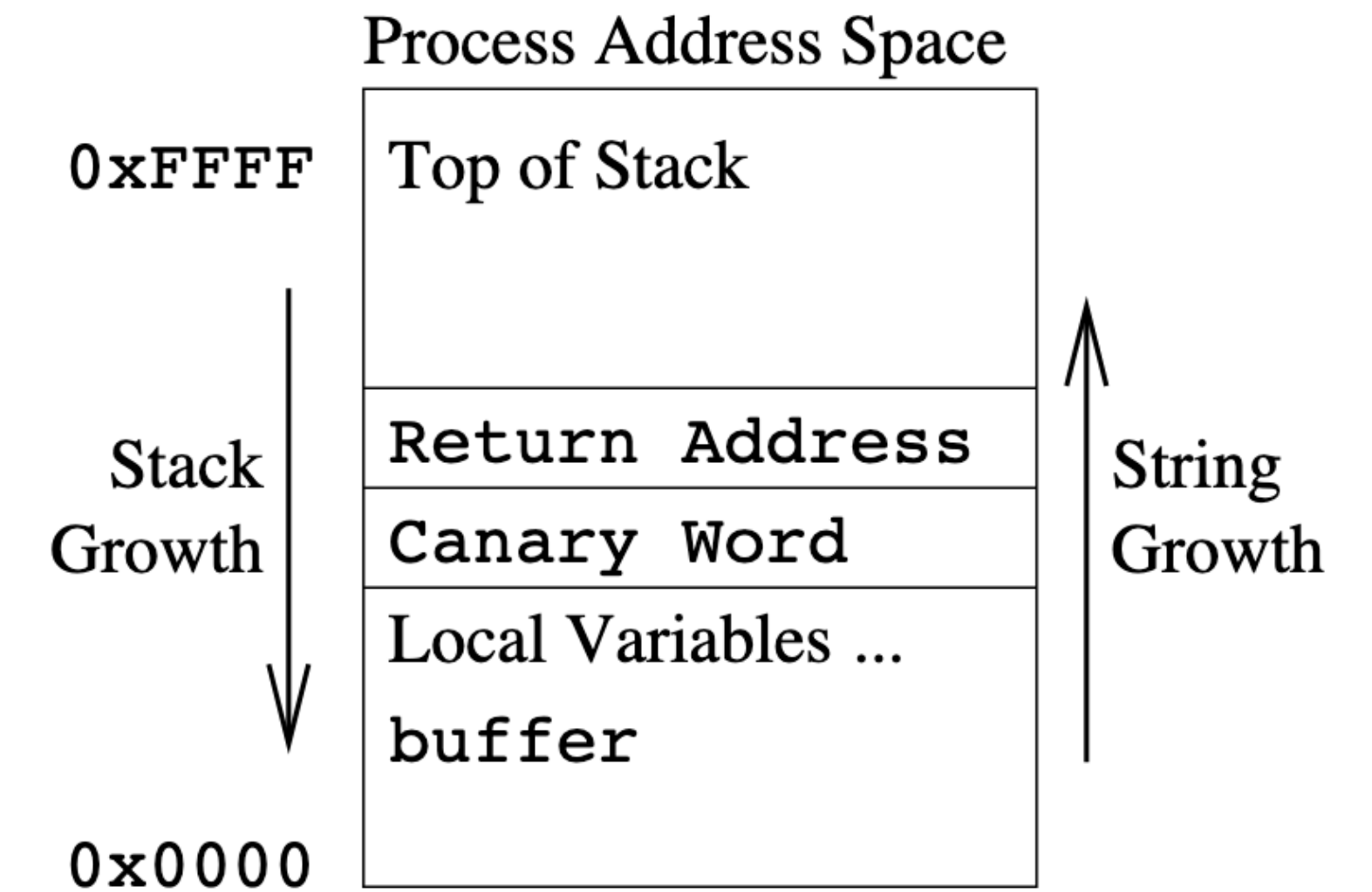
Recap: Stack-based Buffer Overflow

- Overwrite return address of a function in a program
 - Some buffer is on the stack
 - Stack grows down
 - Buffer is at a lower address than return address
 - Writing more means overwriting return address
 - Code execution



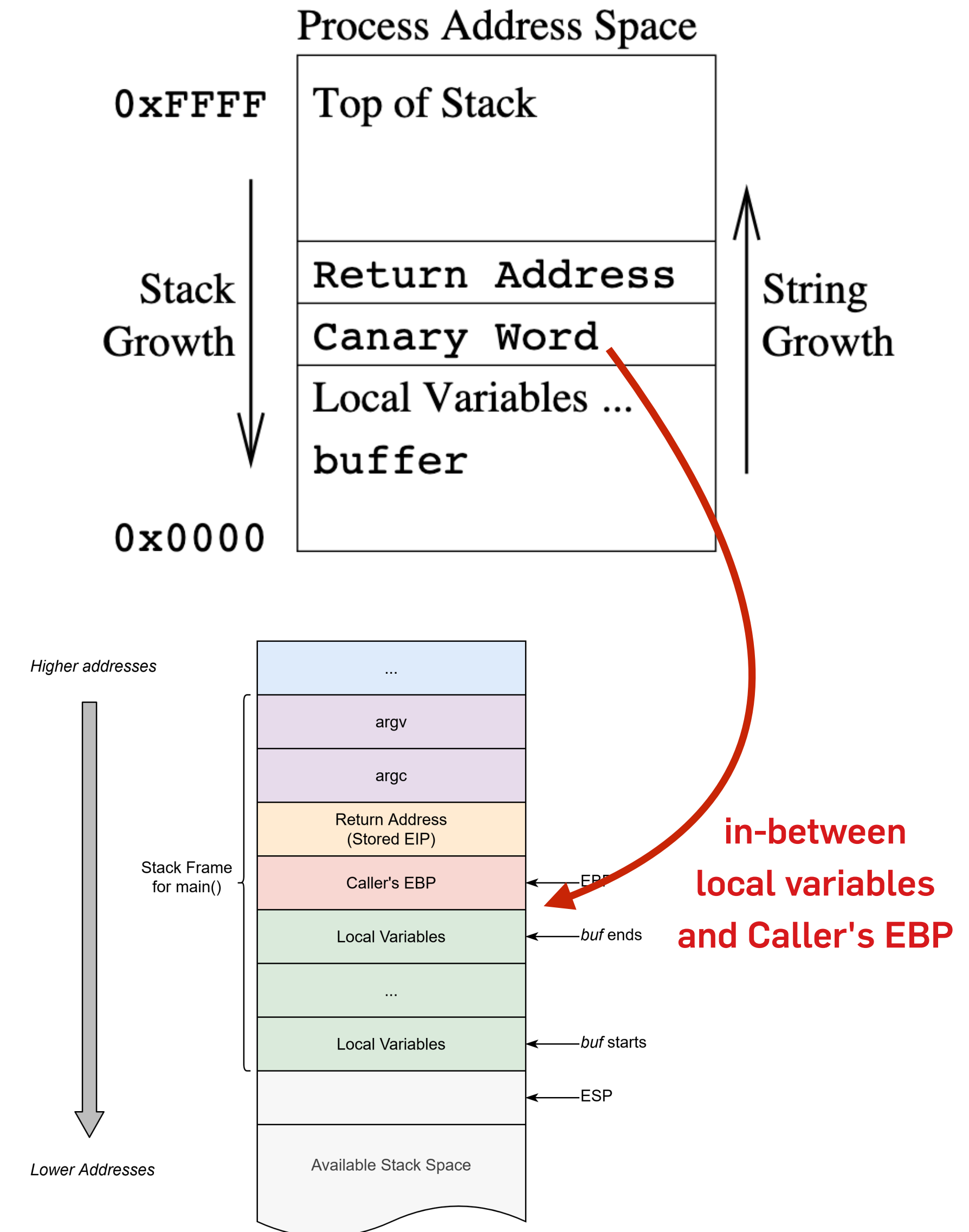
Defending against Stack-based Overflows

- Stack Canaries
 - Introduced in 1998 by Cowan et al.:
"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks"
 - Basic Idea: Put a guard value before the control information you want to protect
 - Check the value before returning
 - Jump to handler/exit if it does not match
 - Name comes from coal mine canaries (canaries need more oxygen than humans and if they died, it was an early warning sign for coal miners that oxygen supply is a problem)



Defending against Stack-based Overflows

- Stack Canaries
 - Introduced in 1998 by Cowan et al.:
"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks"
 - Basic Idea: Put a guard value before the control information you want to protect
 - Check the value before returning
 - Jump to handler/exit if it does not match
 - Name comes from coal mine canaries (canaries need more oxygen than humans and if they died, it was an early warning sign for coal miners that oxygen supply is a problem)



Defeating Stack Canaries

- Stack canaries are generally an effective defense

Defeating Stack Canaries

- Stack canaries are generally an effective defense
- Defeating via three main methods

Defeating Stack Canaries

- Stack canaries are generally an effective defense
- Defeating via three main methods
 - Leak the canary using another vulnerability

Defeating Stack Canaries

- Stack canaries are generally an effective defense
- Defeating via three main methods
 - Leak the canary using another vulnerability
 - Brute-force the canary for forking processes
 - Might require a lot of tries

```
/* brute force */
void main(void) {
    char buf[16];
    while(true) {
        if(fork()) {
            wait(0);
        } else {
            read(stdin, buf, 128);
            return;
        }
    }
}
```


Defeating Stack Canaries

- Stack canaries are generally an effective defense
- Defeating via three main methods
 - Leak the canary using another vulnerability
 - Brute-force the canary for forking processes
 - Might require a lot of tries
 - "Jump" over the canary and keep it intact
 - For example, overwrite read offset i

```
/* brute force */
void main(void) {
    char buf[16];
    while(true) {
        if(fork()) {
            wait(0);
        } else {
            read(stdin, buf, 128);
            return;
        }
    }
}
```

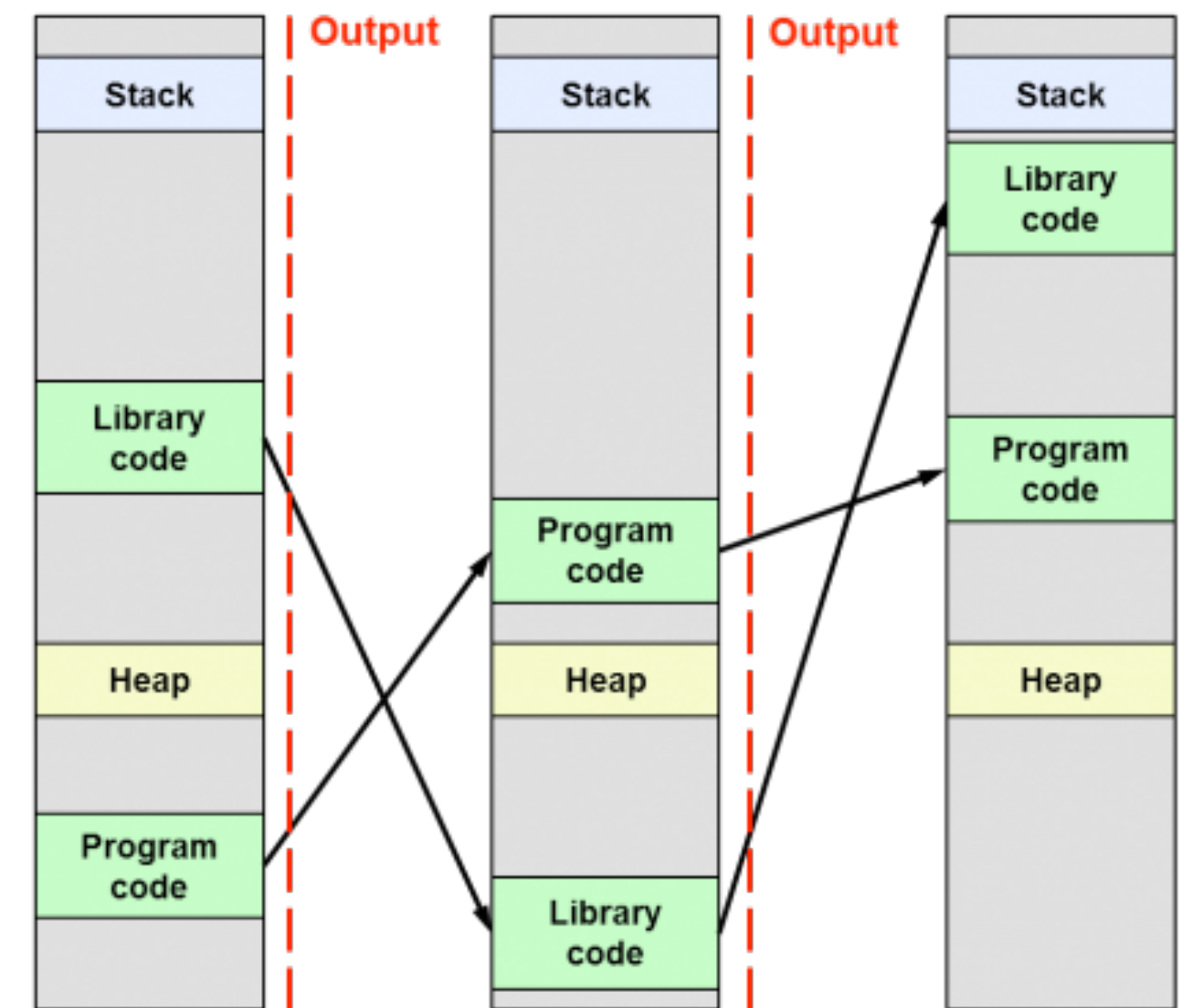
```
/* jump over */
void main(void) {
    char buf[16];
    int i;
    for(i=0; i < 128; ++i) {
        read(stdin, buf + i, 1);
    }
}
```


Address-Space Layout Randomization (ASLR)

- Memory corruptions aim to corrupt pointers to somewhere else and leverage them, for example:
 - Return address, pointer to the next instruction
 - Base pointer, pointer to where the caller's data is stored
- We can randomize these pointers in memory, so that the attacker cannot predict them from reverse engineering
 - This is the fundamental idea of ASLR
- Part of basically every OS nowadays
 - Linux since 2001 (PaX), Windows and MacOS since 2007
- Support by OS and the program are needed
 - Might not be used by all programs or on embedded systems, always worth checking for

Address-Space Layout Randomization (ASLR)

- We can randomize these pointers in memory, so that the attacker cannot predict them from reverse engineering
- But, you still need to compute offsets for code, data, etc.
 - Otherwise your code would not work
- Practically, you only randomize memory pages and not full pointers
 - Also consider relative jumps



Defeating ASLR

- ASLR is generally an effective techniques (like stack canaries)
 - Overall effectiveness (heavily) depends on entropy of randomness
- Defeating via three main methods
 - Leak an address to calculate offset using another vulnerability
 - Addresses still need to be in memory somewhere for the program to access data
 - Brute-force the offset for forking processes
 - Might require a lot of tries (depending on ASLR implementation, you might not re-randomize on fork())
 - Partial overwrite
 - Loaded program assets are page-aligned on Linux
 - Part of the address is predictable, we can just overwrite the page offset

Defeating ASLR: Partial Overwrite

- Memory pages are aligned at 0x1000 boundaries
 - It can be mapped at: 0xCAFE0000 or 0x00403000 or 0x12345000
 - But not at 0xCAFECAFE
- The last three nibbles (0x0-0xF) of an address are stable
- If we overwrite the two least significant bytes of a pointer, we only have to brute force one nibble (= 16 values) to successfully redirect the pointer to another location on the same memory page
 - This gives us limited control
- On little endian, these are the first two bytes that we overwrite!

Defeating Defenses: Leaking Information


- Three main causes why a program might leak information
 - Buffer overread and (lack of) null-termination

```
void main(void) {  
    char buf[16] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    write(stdout, buf, 128);  
}
```

Defeating Defenses: Leaking Information

- Three main causes why a program might leak information
 - Buffer overread and (lack of) null-termination

```
void main(void) {  
    char buf[16] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    write(stdout, buf, 128);  
}
```




write() will not stop at zero/null bytes, and continue writing 128 bytes (which will be what comes after buf on the stack)

Defeating Defenses: Leaking Information

- Three main causes why a program might leak information
 - Buffer overread and (lack of) null-termination

```
void main(void) {  
    char buf[16] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    write(stdout, buf, 128);  
}
```




write() will not stop at zero/null bytes, and continue writing 128 bytes (which will be what comes after buf on the stack)

```
void main(void) {  
    char name[10] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    printf("Name: ");  
    read(stdin, name, 10);  
    printf("Hello %s!\n", name);  
}
```


Defeating Defenses: Leaking Information

- Three main causes why a program might leak information
 - Buffer overread and (lack of) null-termination

```
void main(void) {  
    char buf[16] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    write(stdout, buf, 128);  
}
```



write() will not stop at zero/null bytes, and continue writing 128 bytes (which will be what comes after buf on the stack)

```
void main(void) {  
    char name[10] = {0};  
    char flag[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, flag, 64);  
    close(flag_fd);  
    printf("Name: ");  
    read(stdin, name, 10);  
    printf("Hello %s!\n", name);  
}
```



printf() assumes that name is null-terminated, but if name is set to 10 non-zero bytes, then printf() also outputs what comes after on the stack

Defeating Defenses: Leaking Information

- Third, remember, memory you might access might not be initialized

```
void main() {  
    foo();  
    bar();  
}  
  
void foo() {  
    char foo_buf[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, foo_buf, 64);  
    close(flag_fd);  
}  
  
void bar() {  
    char bar_buf[64];  
    write(stdout, bar_buf, 64);  
}
```

bar() has likely the same stack layout than foo(), and it is being called right after.

This means that bar_buf[64] is likely at the same place in memory than foo_buf[64]!

Defeating Defenses: Leaking Information

- When writing C, be careful about compiler optimizations

```
void main() {  
    foo();  
    bar();  
}  
  
void foo() {  
    char foo_buf[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, foo_buf, 64);  
    close(flag_fd);  
    bzero(foo_buf, 64);  
}  
  
void bar() {  
    char bar_buf[64];  
    write(stdout, bar_buf, 64);  
}
```

Defeating Defenses: Leaking Information

- When writing C, be careful about compiler optimizations

```
void main() {  
    foo();  
    bar();  
}  
  
void foo() {  
    char foo_buf[64];  
    int flag_fd = open("/flag", 0);  
    read(flag_fd, foo_buf, 64);  
    close(flag_fd);  
    bzero(foo_buf, 64);  
}  
  
void bar() {  
    char bar_buf[64];  
    write(stdout, bar_buf, 64);  
}
```

This looks safe, but if you turn on compiler optimizations, then the compiler is likely removing this call to `bzero()`.

The reason is that `foo_buf` is not being used after, so there is no reason to zero it out (this is perfectly legitimate for the compiler to do).

Side Note: Memory Allocation in iOS 16.1

- These kind of issues are still very common, such that Apple changed how their memory allocator worked in iOS 16.1 (released Oct 2022):
 - "The system memory allocator free() operation zeroes out all deallocated blocks in iOS 16.1 beta or later. Invalid accesses to free memory might result in new crashes or corruption, including:
 - Read-after-free bugs that previously observed the old contents of a block may now observe zeroes instead
 - Write-after-free bugs may now cause subsequent calls to calloc() to return non-zero memory"

Compiler Flags

- Defenses are typically enabled via compiler flags
 - ASLR: -fPIE / -fPIC
 - Stack Canary: -fstack-protector
 - Compile-time protection against static-size buffer overflows: -D_FORTIFY_SOURCE=1 to -D_FORTIFY_SOURCE=3
 - Many more options exist and they affect/prevent techniques we have not talked about yet, see: <http://wiki.debian.org/Hardening>
- Investigate how the executable you are trying to exploit was compiled, check build instructions etc.

Software Security 1

Programs, Loading, and Processes

Kevin Borgolte

kevin.borgolte@rub.de

What is /bin/cat?

```
$ file /bin/cat
```

```
/bin/cat: ELF 64-bit LSB pie executable, x86-64, version 1  
(SYSV), dynamically linked, interpreter /lib64/ld-linux-  
x86-64.so.2,  
BuildID[sha1]=44af8b317775373b1a7783fbd0d83c2fe7f21f6e, for  
GNU/Linux 3.2.0, stripped
```

What is /bin/cat?

```
$ file /bin/cat
```

```
/bin/cat: ELF 64-bit LSB pie executable, x86-64, version 1  
(SYSV), dynamically linked, interpreter /lib64/ld-linux-  
x86-64.so.2,  
BuildID[sha1]=44af8b317775373b1a7783fbd0d83c2fe7f21f6e, for  
GNU/Linux 3.2.0, stripped
```


What is /bin/cat?

```
$ file /bin/cat
```

```
/bin/cat: ELF 64-bit LSB pie executable, x86-64, version 1  
(SYSV), dynamically linked, interpreter /lib64/ld-linux-  
x86-64.so.2,  
BuildID[sha1]=44af8b317775373b1a7783fbd0d83c2fe7f21f6e, for  
GNU/Linux 3.2.0, stripped
```

ELF?

- ELF is the "Executable and Linkable Format"
- It is a binary file format and contains program and its data
- It describes how the program should be **loaded**
(via program and segment headers)
- It contains metadata describing program components
(via section headers)
- We will be working with ELF files in this course!
 - ELF is typical for Linux

ELF: Program Headers

- Program headers specify information needed to prepare the program for execution (program image)
- Most important are
 - INTERP, defines the interpreter/library to load this ELF file
 - LOAD, defines part of the file that should be loaded into memory
- They are the source of information when loading a file

ELF: Section Headers

- Section headers give you metadata for introspection, debugging, etc.
 - Section headers are NOT necessary for ELF files, only segments (via program header) are necessary for a program as an ELF file to load and execute
- Some common and important sections
 - .text, the executable part of your program
 - .plt (procedure linking table), used to resolve and dispatch library calls
 - .got (global offset table), also used to resolve and dispatch library calls
 - .data, used for initialized global writable data (e.g., global arrays with initial values)
 - .rodata, used for global read-only data (e.g., string constants)
 - .bss, used for uninitialized global writeable data (e.g., global arrays without initial values)
 - .rel, used when relocating an ELF file at loading/runtime (e.g., PIE with ASLR enabled)

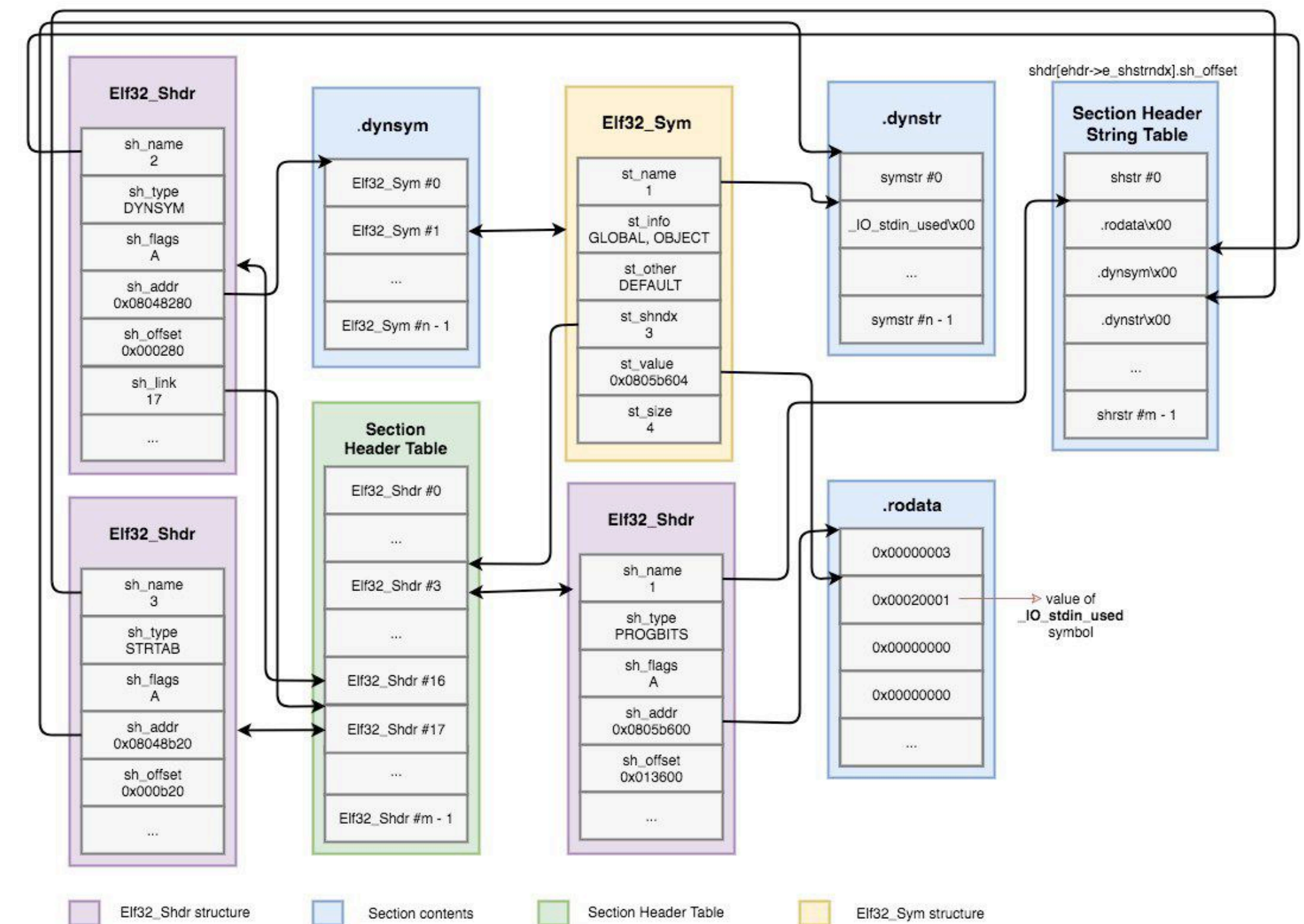
ELF: Section Headers

- Section headers give you metadata for introspection, debugging, etc.
 - Section headers are NOT necessary for ELF files, only segments (via program header) are necessary for a program as an ELF file to load and execute
- Some common and important sections
 - .text, the executable part of your program
 - .plt (procedure linkage table)
 - .got (global offset table)
 - .data, used for initialized global data (e.g., global arrays with initial values)
 - .rodata, used for read-only data (e.g., string literals)
 - .bss, used for uninitialized global writable data (e.g., global arrays without initial values)
 - .rel, used when relocating an ELF file at loading/runtime (e.g., PIE with ASLR enabled)

Important: Memory sections have permissions.
.text is typically read + execute
.plt and .got might be writable, this can allow you to overwrite where the programs thinks library calls are!
Be aware of your execution environment!

Symbols

- Binaries (and libraries) that use dynamically loaded libraries rely on symbols (names) to find libraries, resolve function calls into the libraries, etc.
- Resolving them manually is quite tedious (but tools help)



More about ELF Files

- Tools to play around with ELF files:
 - readelf, to parse the ELF header
 - objdump, to parse the ELF header and disassemble the source code
 - nm, to view symbols
 - patchelf, to change ELF properties
 - objcopy, to swap out entire ELF sections
 - strip, to remove helpful information (e.g., symbols)
 - katai struct (<https://ide.kaitai.io>), for "browsing" through it interactively
- Additional reading
 - <https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/>
 - <https://www.intezer.com/blog/elf/executable-linkable-format-101-part-2-symbols/>
 - <https://www.intezer.com/blog/elf/executable-and-linkable-format-101-part-3-relocations/>
 - <https://www.intezer.com/blog/elf/executable-linkable-format-101-part-4-dynamic-linking/>

Running `/bin/cat`

- Create a process
- Load cat into the process
- Initialize cat
- Launch cat
- cat reads its arguments and environment variables
- cat does what cat does
- cat exits

Running `/bin/cat`

- Create a process
- Load cat into the process
- Initialize cat
- Launch cat
- cat reads its arguments and environment variables
- cat does what cat does
- cat exits

Creating a Process

- Every Linux process has
 - state (running, waiting, stopped, zombie)
 - scheduling information (priority etc.)
 - parent, sibling, and children
 - shared resources (files, pipes, sockets, etc.)
 - virtual memory space
 - security context
 - effective user id and group id
 - saved user id and group id
 - capabilities

Creating a Process

- On Linux, processes start from another process
- **fork** and **clone** are system calls that create a nearly exact copy of the calling process (parent and child)
- The child process then usually uses the **execve** system call to replace itself with another process
 - You type `/bin/cat` into your shell (zsh/bash/etc.) to execute cat
 - Your shell forks itself into the old parent process and the child process
 - The child shell processes `execve /bin/cat`, becoming `/bin/cat`
 - The parent waits for the child

Loading an Executable

- Before we load, kernel checks for executable permissions of the file
 - If the file is not executable, `execve` will fail
- To determine what to do, kernel reads the beginning of the file (can be recursive):
 - If the file starts with a shebang (`#!`), kernel extracts the interpreter from the rest of the line and execute this interpreter with the original file as an argument
 - If the file matches a format in `/proc/sys/fs/binfmt_misc`, the kernel executes the interpreter specified for that format with original file as an argument
 - If the file is a dynamically-linked ELF file, the kernel reads the interpreter/loader defined in the ELF program header, loads the interpreter and the original file, and lets the interpreter do its thing
 - If the file is a statically-linked ELF file, the kernel will directly load it
 - Other legacy formats are also checked (sometimes relevant for more arcane exploitation)

Dynamically-linked ELF Files

- Process loading is done by the ELF interpreter specified in the binary.
- `$ readelf -a /bin/cat | grep "program interpreter"`
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
- Also called "the loader"
- Can be overwritten by specifying it directly:
`/lib64/ld-linux-x86-64.so.2 /bin/cat ...`
- Can also be changed permanently:
`patchelf --set-interpreter`

Dynamically-linked ELF Files

- Program and its interpreter are loaded by the kernel
- Interpreter locates the libraries
 - Checks LD_PRELOAD environment variable and /etc/ld.so.preload
 - Checks LD_LIBRARY_PATH environment variable
 - Checks DT_RUNPATH or DT_RPATH (part of the binary)
 - Checks system-wide configuration /etc/ld.so.conf
 - Checks /lib and /usr/lib
- Interpreter loads the libraries
 - Libraries can depend on other libraries, leading to more libraries being loaded
 - The relocation and information where to find library functions etc. are updated

Where Does This Information Go?

- Each Linux process has a virtual memory space, containing
 - Binary itself
 - Libraries
 - Stack (function local variables, return address etc.)
 - Heap (dynamically allocated memory)
 - Other memory regions explicitly mapped
 - Some helper regions
 - Sometimes some kernel code (usually legacy applications)
- Virtual memory is dedicated to the process
Physical memory is shared among the whole system
- You can check `/proc/self/maps` on Linux

Standard C Library: libc

- libc.so is linked by almost every process
- It provides functionality that you (likely) take for granted
 - printf, scanf
 - open, read, write, close
 - socket
 - malloc, free
- and other interesting stuff, which can be useful for vulnerability assessment:
 - <https://man7.org/linux/man-pages/man3/undocumented.3.html>
(not all functions here are bound to be in libc, but some are)
- These library functions need to be looked up! (typically via .plt or .got)

Initializing cat

- Every ELF file can specify constructors, which are functions that are run before the program is actually launched
- For example, libc can initialize memory regions for dynamic allocations (malloc/free) when the program launches
- You can also specify your own:

```
__attribute__((constructor)) void foo() {  
    puts("bar");  
}
```

Running `/bin/cat`

- Create a process
- Load cat into the process
- Initialize cat
- Launch cat
- cat reads its arguments and environment variables
- cat does what cat does
- cat exits

Launch cat and cat reading arguments

- Launch cat
 - Call `__libc_start_main()`, which calls `main()`
 - Success!
- Reading arguments and environment
 - Remember: `int main(int argc, void **argv, void **envp)`
 - At launch, entire input from outside the program is:
 - The loaded objects (binaries and libraries)
 - Command-line arguments via `argv`
 - Environment set-up via `envp`

cat does what cat does

- Using library functions
 - Binary's import symbols are resolved using libraries' export symbols
 - Nowadays set up at load-time
- Interacting with the outside world
 - System calls (program calls operating system)
 - Signals (operating system calls program)
 - Register a signal handler and receive signals (SIGKILL and SIGTERM are signals, which you can use to terminate a program)
 - Shared memory (establish via system calls, then use freely)

cat terminates

- Processes terminate for two reasons
 - Receiving an unhandled signal that is interpreted as default action kill
 - Calling the exit system call via `exit(status)`
- Processes that terminated need to be "reaped"
 - They remain a zombie until the parent `wait()`s on them
 - Their exit status code is then returned the parent, and the process freed
 - If the parents dies without `wait()`, then the processes are re-parented to PID 1 (init) and will remain there until cleaned up

Software Security 1

Format String Vulnerabilities

Kevin Borgolte

kevin.borgolte@rub.de

Format String Vulnerabilities

```
void main(int argc, char *argv[]) {  
    if(argc > 2) {  
        printf(argv[1], argv[2]);  
    }  
}
```



What does printf() do with its arguments?

Format String Vulnerabilities: printf

```
printf(const char *format, ...);
```

- printf() takes a variable number of arguments
 - printf("foobar");
 - printf("foo%s", bar);
 - printf("%d", 42);
 - printf("%d / %d = %f", 13, 37, 13.0/37.0);
- printf knows the number of arguments it should read based on the format string

printf arguments

```
printf(const char *format, ...);
```

- printf knows the number of arguments it should read based on the format string

printf arguments

```
printf(const char *format, ...);
```

- printf knows the number of arguments it should read based on the format string
 - What if there are too many arguments?

printf arguments

```
printf(const char *format, ...);
```

- printf knows the number of arguments it should read based on the format string
 - What if there are too many arguments?
 - What if there are not enough arguments?

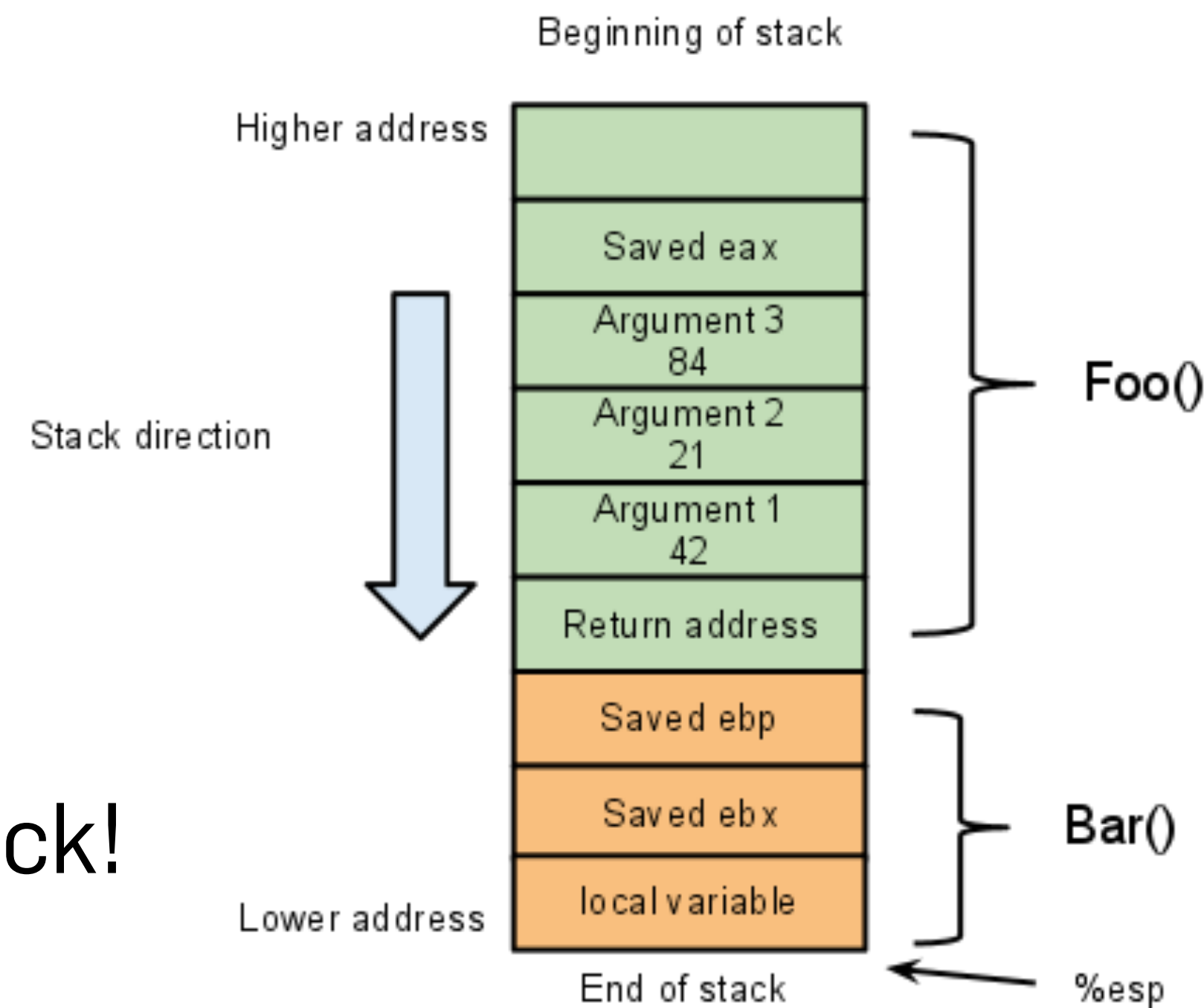
printf arguments

```
printf(const char *format, ...);
```

- printf knows the number of arguments it should read based on the format string
 - What if there are too many arguments?
 - What if there are not enough arguments?
 - What if you control the format string?
 - This is not actually uncommon, especially not as the result of another memory corruption that allows you to overwrite a string in memory

Calling Conventions

- Remember function calling conventions (from the first lecture)
 - cdecl and stdcall store function arguments on stack
 - fastcall stores them in registers
 - No information on how many arguments were passed to the function
 - printf() needs to parse the format string and then read that many values from the stack!



cdecl

Call:

```
push 3
push 2
push 1
call F
add esp, 12
cmp eax, 0
jz error
```

Subroutine F:

```
push ebp
mov ebp, esp
...
pop ebp
ret
```

Capabilities

- If you control the format string, you can
 - Leak information on the stack, based on the format string specifier
 - %c to read a single character
 - %s to read a string
 - %p to read a pointer
 - %d, %i to read 4 byte integers (signed)
 - %o, %u, %x, %X to read 4 byte integers (unsigned)
 - They can be modified by length modifiers
 - hh to convert to char, h to short, l to long, ll to long long, and others!

Leaking Information

- manpages are trove of information, these functions have some interesting behavior:
 - One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m\$" instead of '%' and "*m\$" instead of '*', where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,


```
printf("%*d", width, num);
```


and


```
printf("%2$*1$d", width, num);
```


are equivalent.
 - This kind of behavior is useful for a lot of applications, so it has a purpose and cannot readily be removed (independent of backward compatibility), but it is still very useful for us
- This means you have direct control over what you leak from the stack!

Real-world Vulnerabilities

- Non-trivial format string vulnerabilities exist

Four format string injection vulnerabilities exist in the XCMD testWifiAP functionality of Abode Systems, Inc. iota All-In-One Security Kit 6.9X and 6.9Z. **Specially-crafted configuration** values can lead to **memory corruption, information disclosure and denial of service**. An attacker can modify a configuration value and then execute an XCMD to trigger these vulnerabilities.

Supplying a **config->ssid_hex** or **config->ssid** value of **%x.%x.%x.%x.%x.%x...** results in the following log message being generated:

```
[DBG ][WEB ]driver/wpa_cli -i wlan0 set_network 0 ssid  
""7148d871.7148d950.333a4143.252e51.0.33.76c46000.3a224e50.252e78""
```

- Not limited to printf():

- 8 other printf variants
- 6 scanf variants
- all equally exploitable

```
/* Examples:  
    Log(6, 13, "Initialized SSL"); -> [DBG!][NET ]Initialized SSL  
    Log(3, 13, "SSL init error: %d", error); -> [ERR!][NET ]SSL init error: {error}  
*/  
void log(unsigned int severity, unsigned int task, const char *format, ...)  
{  
    [...]  
  
    // Populate remainder of message with format and var_args  
    vsnprintf(&log_buffer[12], 499u, format, var_args);  
  
    // Put crafted message on to UART  
    puts(log_buffer);  
}
```

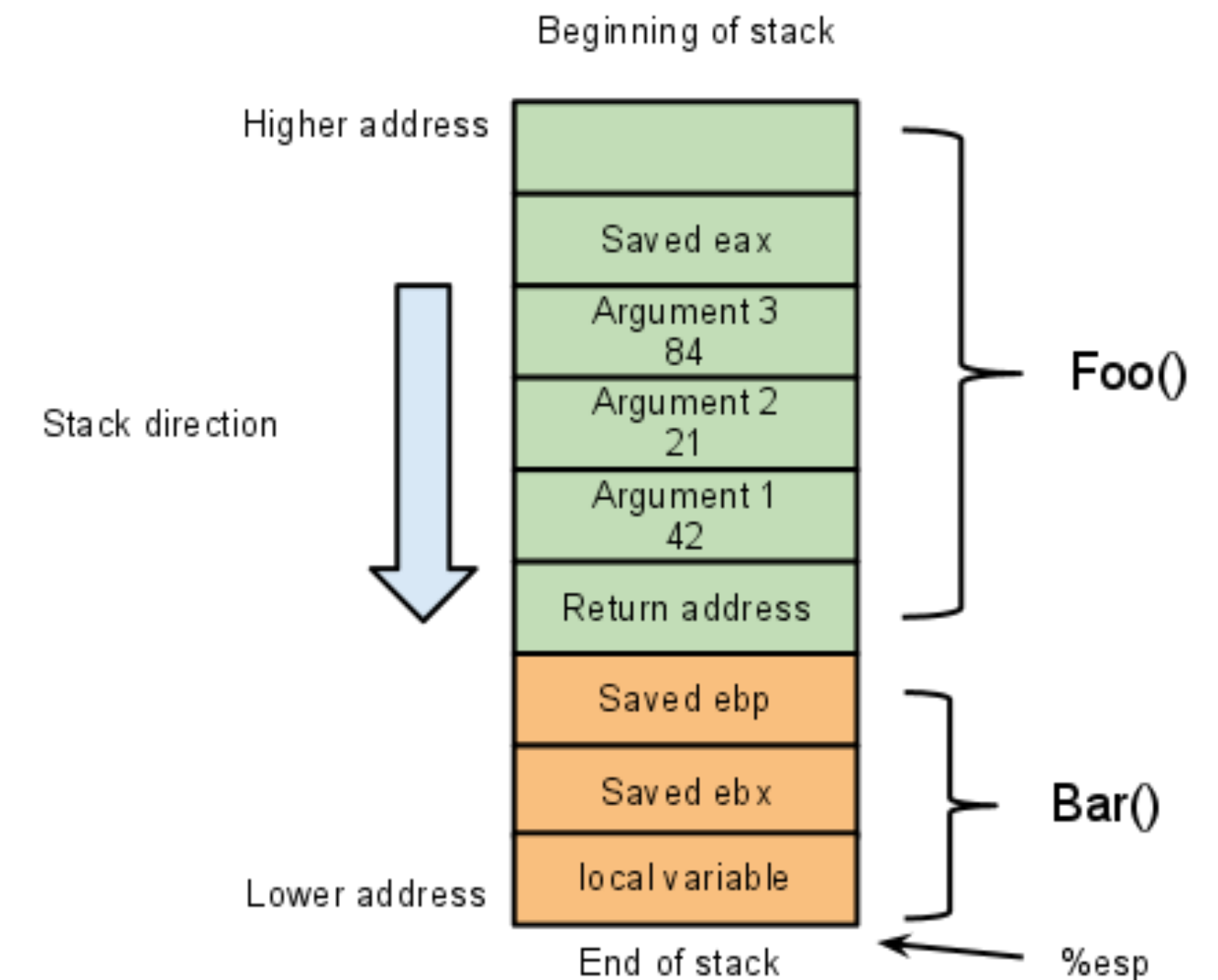

Writing Memory?

- Leaking information is only part of what we can do:
 - There is another conversion specifier: %n
 - "The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. The behavior is undefined if the conversion specification includes any flags, a field width, or a precision."

```
void main(int argc, char *argv[]) {  
    if(argc > 1) {  
        int len;  
        printf("%s%n\n", argv[1], &len);  
        printf("len(argv[1]) = %d\n", len);  
    }  
}
```

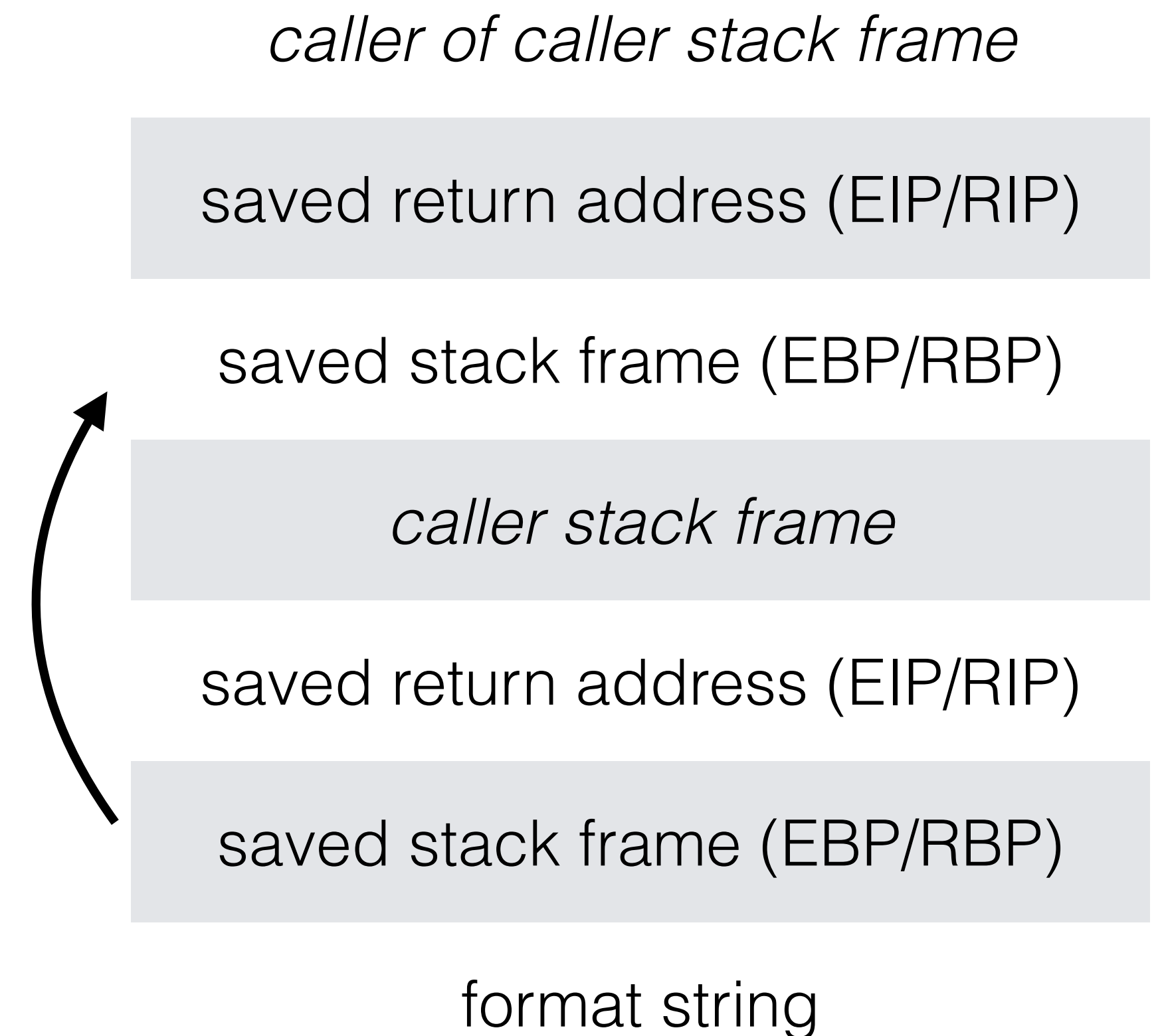
Writing Memory

- `%n` requires a pointer to where we want to write
- Our buffer might be on stack
- We can use other pointers on the stack (`%3$n`)
- Frame/base pointers can be on the stack and point to each other
 - We can overwrite the frame pointer



Writing Memory

- `%n` requires a pointer to where we want to write
- **Our buffer might be on stack**
 - If the format string is on the stack



What is below?

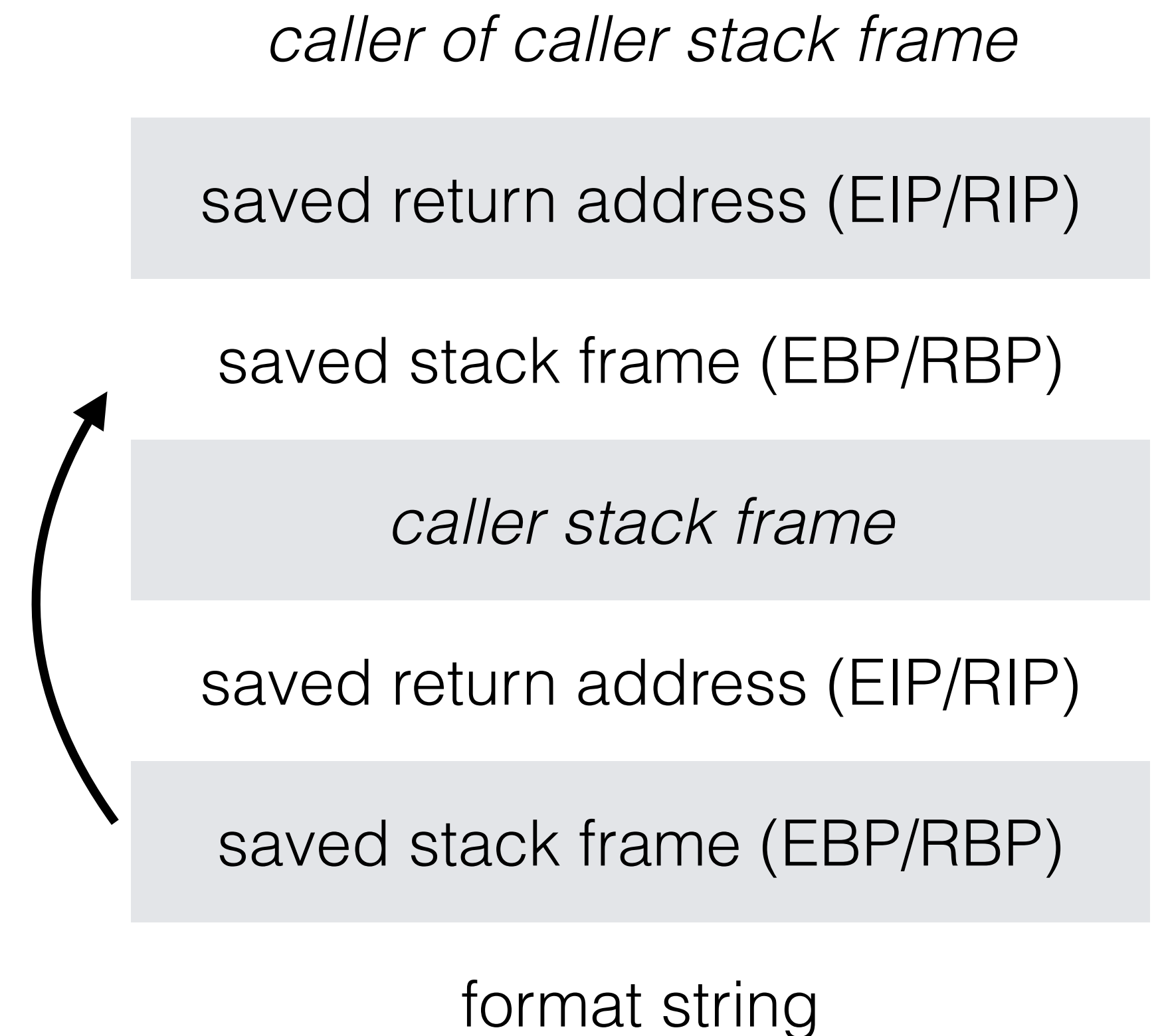
Writing Memory

- `%n` requires a pointer to where we want to write

- **Our buffer might be on stack**

- If the format string is on the stack
- Below the stack frame of the function calling `printf()` is the stack frame of `printf()`.

We can reference our own format string using `_%$n!`



What is below?

Writing Memory: Length Modifiers

- `%n` writes 4 bytes: "integer pointed to by the argument"
- We can write more or less with length modifiers
 - `%ln` to write as long
 - `%hn` to write a short
 - `%hhn` to write a char
- Controlling what to write is another challenge, ideas?

Writing Memory: What to Write

- We can leverage another feature of format strings: field width
 - An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given).

```
void main(void) {  
    char buf[4];  
    printf("%43981x%1$n", buf);  
}
```

What does this write?

Is there any problem with this?

Writing Memory: What to Write

- We can do better combining it with length modifiers

```
void main(void) {  
    char buf[4];  
    printf("%43981x%1$n", buf);  
}
```

Slow!

```
void main(void) {  
    char buf[4];  
    printf("%65x%1$hhn%x%2$hhn%c%3$hhn%c%4$hhn",  
          buf, buf + 1, buf + 2, buf + 3);  
}
```

Faster!

Writing Memory: What to Write

- We can do better combining it with length modifiers

```
void main(void) {  
    char buf[4];  
    printf("%43981x%1$n", buf);  
}
```

Slow!

```
void main(void) {  
    char buf[4];  
    printf("%65x%1$hhn%x%2$hhn%c%3$hhn%c%4$hhn",  
          buf, buf + 1, buf + 2, buf + 3);  
}
```

Faster!

There is still an issue here, any idea what it might be?

Writing Memory: Precision

- Problem: "The number of characters written so far"
- The number is always increasing, we cannot write 0xDCBA
- Luckily, there is another feature that we can misuse: precision
 - An optional precision, in the form of a period ('.') followed by an optional decimal digit string. Instead of a decimal digit string one may write "*" or "*m\$" (for some decimal integer m) to specify that the precision is given in the next argument, or in the m-th argument, respectively, which must be of type int.

Writing Memory: Precision

- Example: `/*7$x%8$n`
 - Take the 7th parameter
 - Use it as the padding size of a single character
 - Print that many bytes
 - Write the number of bytes we just printed (or read for scanf) to the memory location that the 8th parameter points to
- This copies the 7th parameter to the 8th parameter
- Remember: the arguments do not actually need to be argument, but can be arbitrary stack values
- Same problem as before: Too slow if values are too big

Format String Vulnerabilities

- We (mis)use how printf and variants work:
 - Stack frame and calling convention to access other memory
 - Conversion parameters:
 - Leaking information via "normal" conversions
 - Writing memory via %n
 - Specify the data type/width with we want to access via length modifiers
 - Use field width to write arbitrary values to memory
 - Use precision to copy values in memory

Side Note: Weird Machines

- Generic printf() is Turing complete
 - printbf -- Brainfuck interpreter in printf
<https://github.com/HexHive/printbf>
 - Control-Flow Bending: On the Effectiveness of Control-Flow Integrity
Carlini et. al, <https://nebelwelt.net/publications/files/15SEC.pdf>
- Abusing printf is "programming a weird machine"
 - The analysis of weird machines allow to scrutinize exploits from a more theoretical side
 - We'll talk more about this November 6th!

Software Security 1

Return-Oriented Programming

Kevin Borgolte

kevin.borgolte@rub.de

Shellcode Mitigations: NX-bit

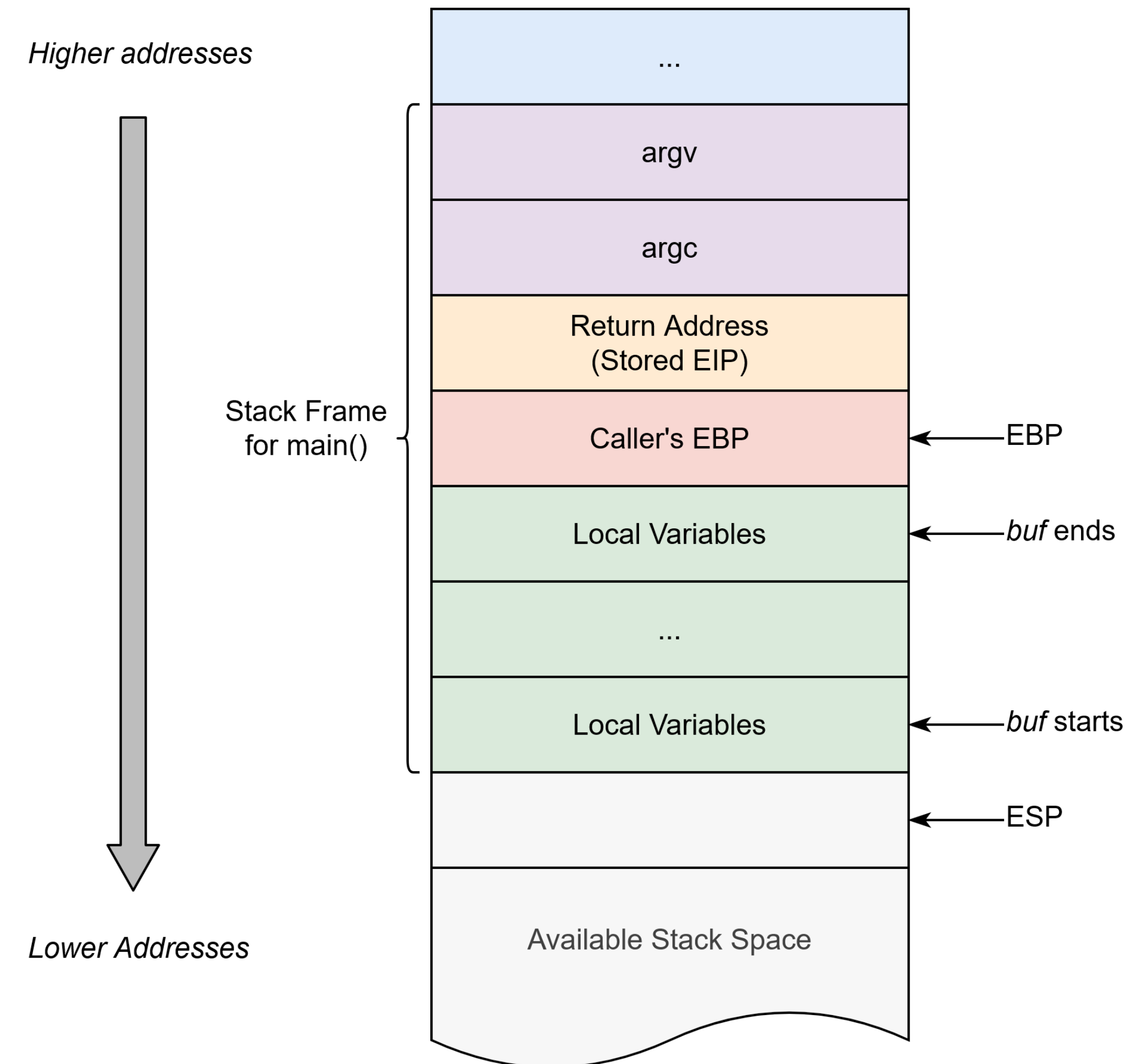
- Nowadays, the stack is basically never executable (NX)
- Memory has three primary permissions
 - PROT_READ, allow the memory to be read
 - PROT_WRITE, allow the memory to be written
 - PROT_EXEC, allow the memory to be executed
- Usually, all code is in the .text segment of the ELF file
 - No need to load execute code on the stack or heap
- Memory is typically W^X (either writable or executable, but not both)
- This also extends to the heap

Code Reuse

- ~~Code injection~~
 - ~~Basic idea: inject some new code into the application~~
 - Doesn't work anymore 😞
- Instead: reuse code that is already there
 - return-to-libc
 - more generally, return oriented programming (ROP)
 - ROP chains

return-to-libc

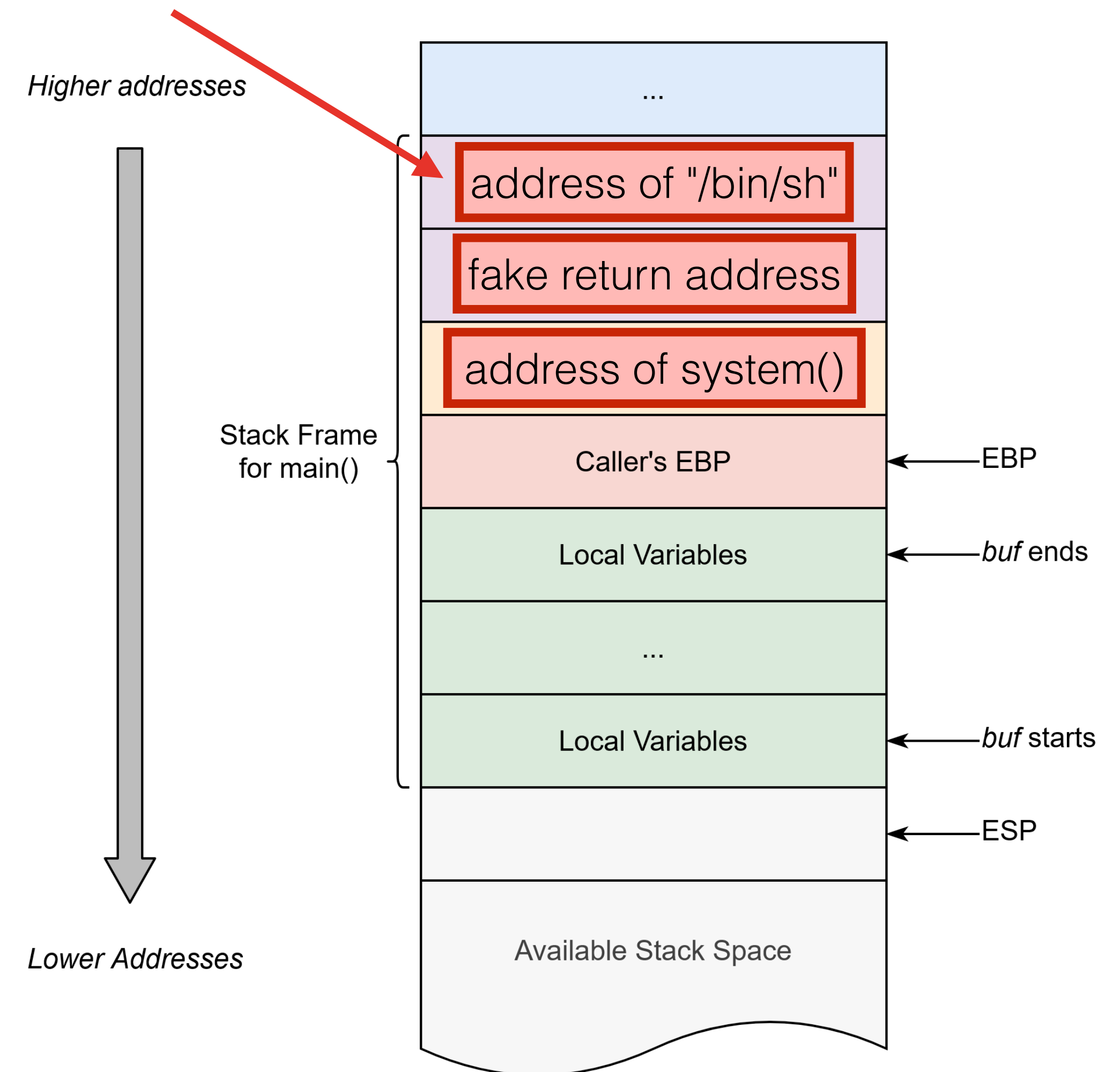
- If the arguments to a function are on the stack, then we can:
 - Overwrite the return address
 - CALL is nothing but push EIP and jump to new address
 - RET is nothing but pop EIP and jump to the new address
 - Overwrite the arguments (since they are on the stack)
- This means we can just make it do a CALL without having to inject code



return-to-libc

- If we know the address of a function, say `system()`, we can misuse the vulnerable program to call it and get a shell
- Overwrite return address with the address of `system()`
- Add a fake return address to the stack from which we (supposedly) called `system()` from
- Add the arguments to `system()` to the stack

"/bin/sh" can be on stack!



return-to-libc?

- Arguments might not be on stack
 - Recall fastcall calling convention
- We can still call
 - Functions that take no arguments

```
void flag() {  
    int fd = open("/flag", 0);  
    sendfile(stdout, fd, 0, 1024);  
    close(fd);  
}
```
 - Functions for which arguments matter only partially for what we want to achieve

fastcall

Call:

```
mov ecx, 3  
mov edx, 2  
mov eax, 1  
call F
```

```
cmp    eax, 0  
jz     error
```

Subroutine F:

```
push ebp  
mov    ebp, esp  
...  
pop    ebp  
ret
```

Ignore Function Arguments?


- Functions for which arguments matter only partially
 - Part of the function will still be executed, this might be sufficient
 - We control the return address to where the function will return to

```
bool read_check(char *value) {  
    int fd = open("/flag", 0);  
    char flag[64];  
    read(fd, flag, 64);  
    close(fd);  
    if(0 != strncmp(value, flag, 64)) {  
        return false;  
    }  
    [...]  
}
```

Ignore Function Arguments?

- Functions for which arguments matter only partially
 - Part of the function will still be executed, this might be sufficient
 - We control the return address to where the function will return to

We cannot control **value*

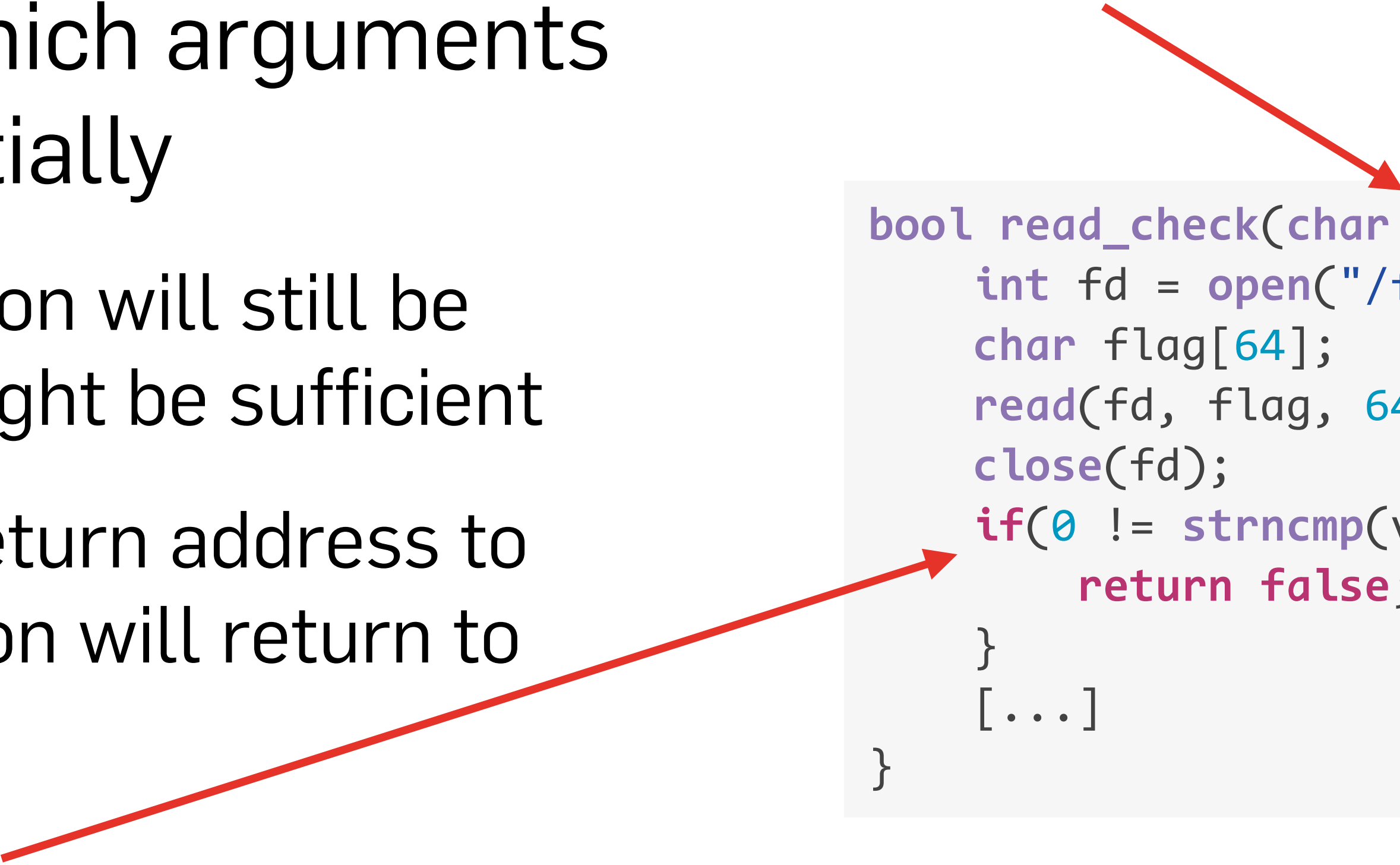


```
bool read_check(char *value) {  
    int fd = open("/flag", 0);  
    char flag[64];  
    read(fd, flag, 64);  
    close(fd);  
    if(0 != strncmp(value, flag, 64)) {  
        return false;  
    }  
    [...]  
}
```

Ignore Function Arguments?

- Functions for which arguments matter only partially
 - Part of the function will still be executed, this might be sufficient
 - We control the return address to where the function will return to

We cannot control `*value`



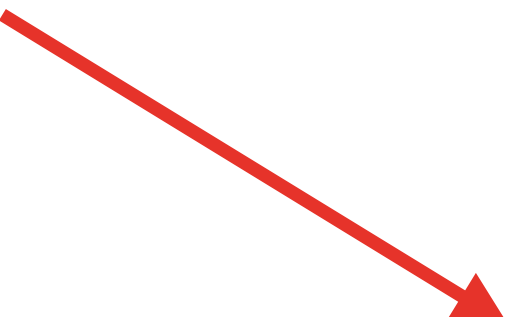
```
bool read_check(char *value) {  
    int fd = open("/flag", 0);  
    char flag[64];  
    read(fd, flag, 64);  
    close(fd);  
    if(0 != strncmp(value, flag, 64)) {  
        return false;  
    }  
    [...]  
}
```

This check will fail, but we can control where `read_check()` returns to, and `flag` is now (probably) on the stack (recall memory initialization) and the address of `flag` might still be in a register (which might be a function argument), we only need some way to read now! (caveat: `strncmp` will attempt to access whatever `value` points to, so `value` must point to a memory region that we can read, or the program will throw a segmentation fault and crash)


Ignore Existing Function Boundaries

- We do not even need to respect existing function boundaries, we can treat any instruction as the start of a function

We cannot control **value*



```
bool read_check(char *value) {  
    int fd = open("/flag", 0);  
    char flag[64];  
    read(fd, flag, 64);  
    close(fd);  
    if(0 != strcmp(value, flag, 64)) {  
        return false;  
    }  
    [...]  
}
```



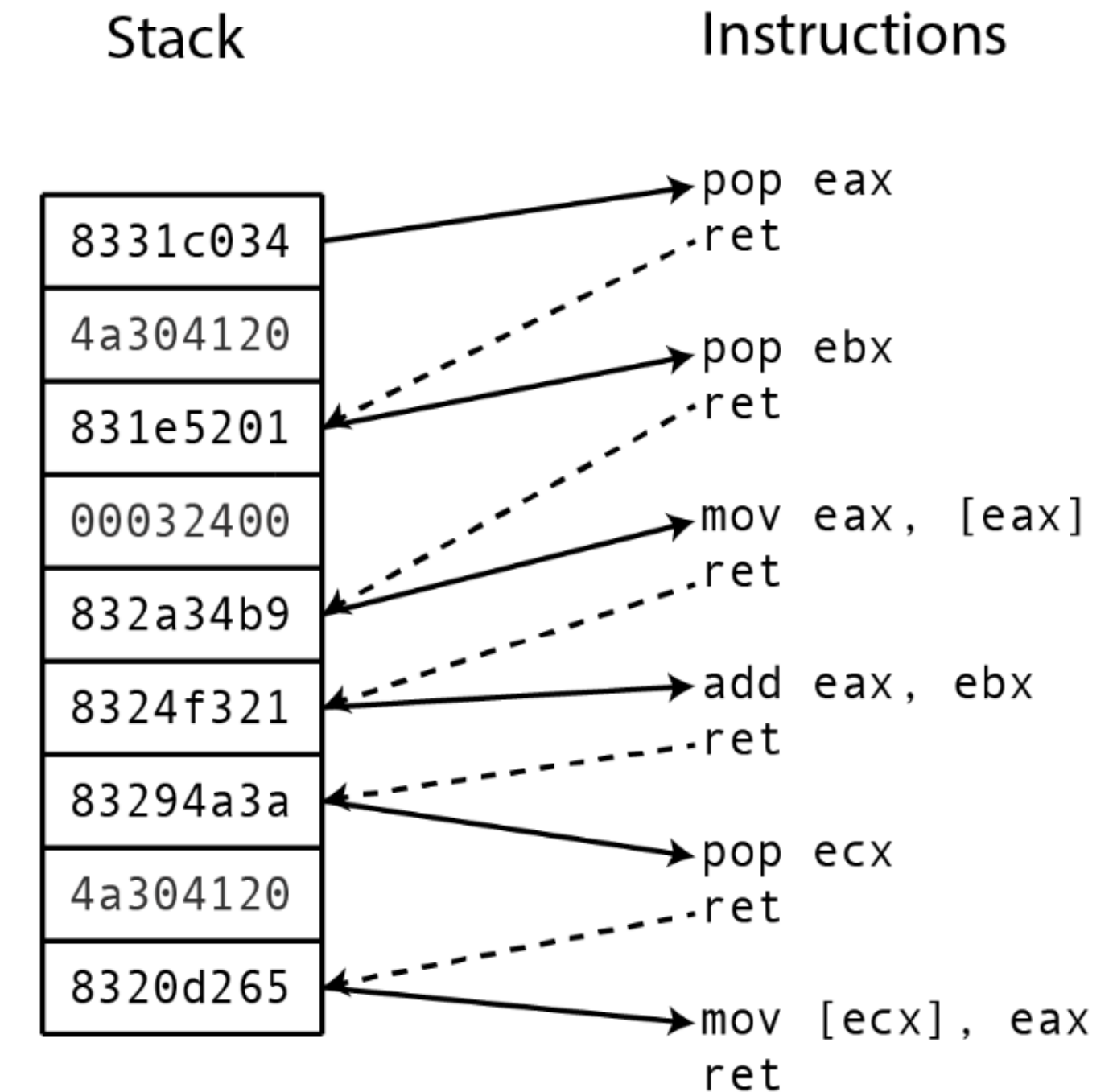
We can call the "function" that starts here, completely skipping the string comparison. But there is a (small) caveat: we no longer have a function prologue, and the stack might become mangled

Return Oriented Programming

- We can also jump in the middle of existing instructions
 - bytes: 49 bc 31 c0 b0 3c 0f 05 90
mov r15, 0x9090050f3cb0c031
 - bytes: 31 c0 b0 3c 0f 05 90
xor eax, eax ; 31 c0
mov al, 60 ; b0 3c
syscall ; 0f 05
nop ; 90
nop ; 90
- This generalization of return-to-libc to treating any address as the start of a "function" is return oriented programming

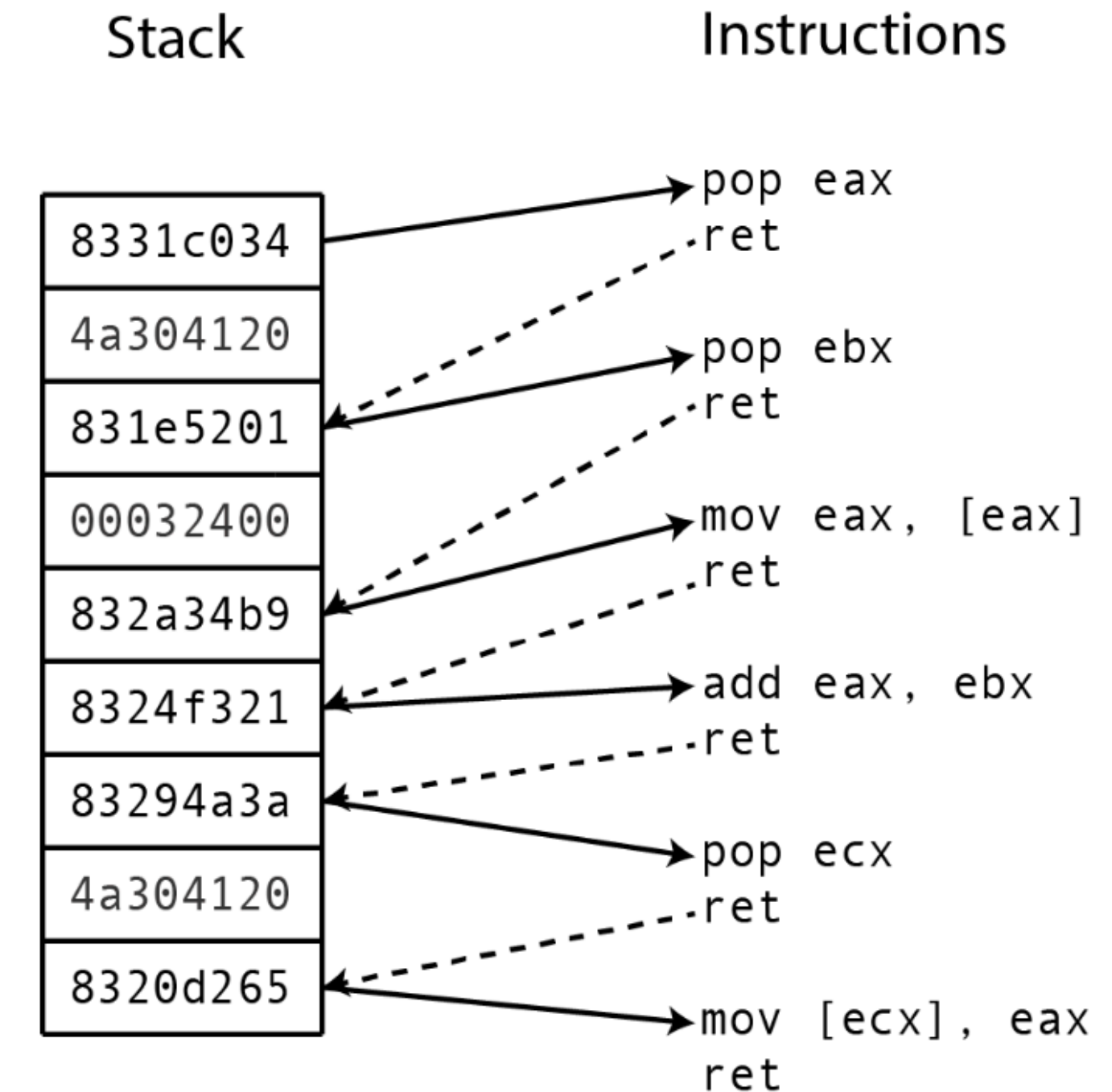
ROP Chains

- ROP steps
 1. Write to the stack
 2. Overwrite the first return address
 3. Overwrite the second return address
 4. ...
 5. Overwrite the n-th return address



ROP Chains

- ROP steps
 1. Write to the stack
 2. Overwrite the first return address
 3. Overwrite the second return address
 4. ...
 5. Overwrite the n-th return address
- Not much different from writing shellcode
 - Different "instruction" set (ROP gadgets)
 - "Instructions" are just longer
 - "Instructions" have more (side-)effects
 - = Programming a weird machine, that is often accidentally Turing complete (fun read: https://beza1e1.tuxen.de/articles/accidentally_turing_complete.html)



ROP: Gadgets

- Setting registers:
pop rax; ret → Set rax and returns to the next gadget
 - Necessary for syscalls, and useful for setting up function calls
- Not all gadgets are equally common, misaligning them helps
 - Common: ret and leave; ret, or pop register; ret, or mov rax, register; ret
 - Some instructions have prefixes for their width that you can jump over (for example: add rsp, immediate vs. add esp, immediate)

ROP: Resources on the Stack

- We might to store some data on the stack, but need to prevent it being interpreted as a return address and jumped to
- Some of our gadgets will need to deal with this, and fix-up our stack
 - `pop rax; pop rbx; ret`
 - `add rsp, X; ret`

Gadget 6 (syscall)

Gadget 5

`"/bin/cat"`

Gadget 4

Gadget 3

`"/flag\0xx"`

Gadget 2

Address of `"/flag\0xx"`

Gadget 1

ROP: Storing Addresses in Registers

- Gadgets to load the effective address (lea instruction) are rare, they are mostly used in the beginning/middle of functions, not near the end; alternatives are often needed
 - `push rsp; pop rax; ret --` basically `mov rax, rsp`
 - `add rax, rsp; ret --` useful if you can somehow set rax before
 - `xchg rax, rsp; ret --` dangerous: you changed the stack pointer
- A ROP gadget to get the stack address can be very helpful
 - You can leak it (think ASLR)
 - You can calculate offsets in your ROP chain, without having to leak it

ROP: Stack Pivoting

- **"dangerous: you changed the stack pointer"**
 - Changing the stack to a new memory location is called **stack pivoting**
 - Frees you from limitations that might be imposed on your stack
 - Many instruction sequences possible
 - `xchg rax, rsp; ret`
 - `pop rsp; ...; ret`
 - `mov rsp, ...; ret`
 - If you can keep the old stack pointer around, you can also pivot back

ROP: Setting Values

- Usually, you need to set data to some value somehow
 - In shellcode, you can simply set values
 - For ROP, you can use common gadgets like
`add byte [rcx], al ; pop rbp ; ret`
but this requires you to have other gadgets to set rcx and set al

ROP: Setting Values

- Usually, you need to set data to some value somehow
 - In shellcode, you can simply set values
 - For ROP, you can use common gadgets like
`add byte [rcx], al ; pop rbp ; ret`
but this requires you to have other gadgets to set rcx and set al
- Consider `rax = 0` and a gadget `"add rax, 1; ret"`. How long does a ROP chain would have to be to set a memory location to `"/flag\0"` (`0x2f666c616700`)?
 - If you can set rcx arbitrarily, it will be 108 returns for setting al
 - If you can only increment rcx (write left to right), it will be 358 returns for setting al (you will also set rax to values larger than what al can hold, but you only access al)

ROP: Notes

- For real-world programs, the syscall instruction is quite rare (even misaligned). Instead, consider calling a ROP chain to set up a library call to do what you want (libc or other).
- The environment your ROP chain runs in is not clean (same as for your shellcode), understand what is stored where
 - Addresses to code, stack, or heap can be in registers and in the stack
- ROP gadgets that you will find will likely have a lot of side-effects, such as clobbering registers you just used!

ROP Gadget Finder

- Finding ROP gadgets is simple, but tedious, tools exist
 - <https://github.com/Gallopsled/pwntools/>
 - <https://github.com/Overcl0k/rp>
 - <https://github.com/sashs/Ropper>
- Ropper also supports constraints on your gadget:
 - Set `eax` to 1 and do not clobber `ebx`:
`eax==1 !ebx`

```
(ls/ELF/x86_64)> semantic |rax==0x_mplus_a2p.git/info/refs
[INFO] Searching for gadgets: rax==0x_mplus_a2p.git/objects/
/mnt/backup/gogs/sash/2017.00016_mplus_a2p.git/objects/info/
[INFO] File: /bin/lsash/2017.00016_mplus_a2p.git/objects/info/packs
0x00000000000000c880: xor eax, eax; ret; s_a2p.git/objects/pack/
Clobbered Register = rsp, rip, rax; StackPointer-Offset = 8; /pack-e050e3b499d1b4d23
/mnt/backup/gogs/sash/2017.00016_mplus_a2p.git/objects/pack/pack-e050e3b499d1b4d23
0x00000000000000d1c4: xor eax, eax; add rsp, 8; ret; ort 22: Broken pipe
Clobbered Register = rsp, rax, rip; StackPointer-Offset = 16
$ ssh trenzalore
0x000000000000013f3e: xor eax, eax; pop rbx; ret;
Clobbered Register = rsp, rip, rax, rbx; StackPointer-Offset = 16
[sash@trenzalore ~]$ ls /mnt/backup/
0x0000000000000156d0: xor edx, edx; mov rax, rdx; ret;
Clobbered Register = rsp, rdx, rip, rax; StackPointer-Offset = 8
archives/ gogs/
0x0000000000000f3c9: xor eax, eax; pop rbp; pop r12; ret;
Clobbered Register = rsp, r12, rax, rip, rbp; StackPointer-Offset = 24
[sash@trenzalore ~]$ ls /mnt/backup/
```

ROP: Practical Issues

- Stack control limited, one gadget might all we can trigger
 - Length, byte values etc.
 - This directly affects the length of your ROP chain
 - Approach: Try to jump half-way through `system()` before `execve()`
- ASLR
 - Partial overwrite: Try to jump back to `main()`, restart the program, and not have it crash, to possible store additional/more data/code somewhere (slowly build your ROP chain)
 - Otherwise information disclosure needed
- Stack canaries
 - Leak or bypass, careful about aligned gadgets, they all might compare the stack canary

Side Note: Blind ROP

- "it is possible to write remote stack buffer overflow exploits without [...] a copy of the target binary or source code, against services that restart after a crash. [...] Traditional techniques [...] where the hacker knows the location of useful gadgets for ROP. Our Blind ROP attack instead remotely finds enough ROP gadgets to perform a `write` system call and transfers the vulnerable binary over the network [...]. This is accomplished by leaking a **single bit of information** based on whether a process crashed or not when given a particular input string."

Side Note: Blind ROP

- Blind ROP works because randomization is at start time
- Three phases
 - Read the stack: Defeat canaries, ASLR, and starting point for gadgets
 - Find one specific BRROP gadget and two entries in the procedure linking table (ELF) to then leak enough of the binary from memory
 - Build a normal ROP exploit
- Leverages the single bit of information that the program either crashes (invalid return address) or does something else (valid address)

ROP: Defenses

- Removing ROP gadgets (too onerous, slow, inefficient)
 - "G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries" (<https://dl.acm.org/doi/abs/10.1145/1920261.1920269>)
- Detecting ROP attacks in progress (bypassable):
 - "kBouncer: Efficient and Transparent ROP Mitigation" (<https://people.csail.mit.edu/hes/ROP/Readings/kbouncer.pdf>)
 - "ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks" (https://www.ndss-symposium.org/wp-content/uploads/2017/09/02_1_1.pdf)
- Control-Flow Integrity