Buffer Overflows

Ramin Sadre

Buffer overflow

- Note that, for performance reasons, the length of a string is not stored nor checked *during runtime* in C. This is true for all data in C.
- What will happen here?

```
char *src="0123456789";
char dest[8];
int i,j;
strcpy(dest,src);
```

- The string `"0123456789"` is 11 bytes long
- The array `dest` is too short to hold the entire array
- `strcpy` does not care! It will overwrite whatever comes after `dest`

Using buffer overflows

- Buffer overflows are especially dangerous if the data comes from unchecked user input
- Imagine a web server that checks the username that a web browser has sent through HTTP:

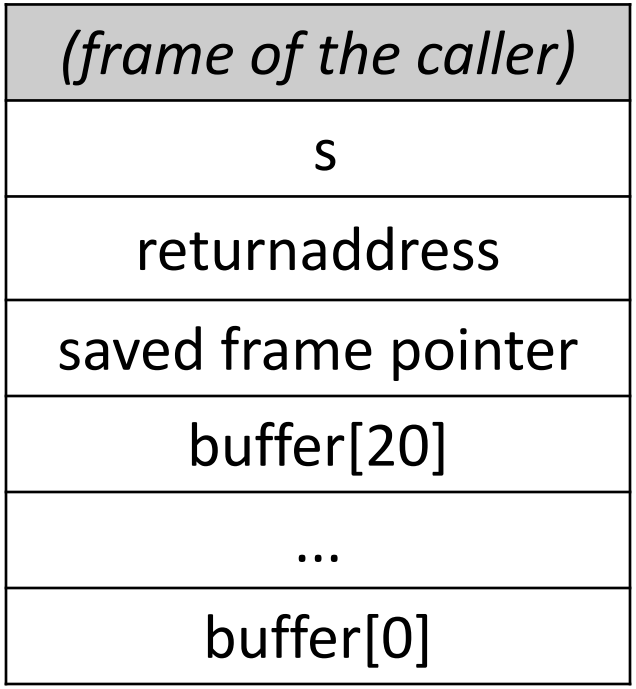
```
char currentUser[8];
int accessRight;

// "name" comes from a HTTP request
void initUser(char *name) {
    accessRight = 0;              // no access right
    strcpy(currentUser,name);
}
void deleteFile(char *filename) {
    if(accessRight==0) return;  // no access
    ...
```

- User can obtain more access rights than they are supposed to have

Buffer overflows in the stack frame

- Because C does not do any runtime checks, it is also possible to overwrite the stack frame with a buffer overflow

```
void f(char *s) {
   char buffer[21];
   strcpy(buffer,s);
}
```



- By providing a too long string s, you can overwrite the the saved frame pointer, the return address, and even the previous frame!

Code injection example

- Sometimes you will see an attacker trying to call a function with a string containing the bytes:

```
31 c0 50 68 2f 73 68 00 68 2f
62 69 6e 89 e3 50 89 e2 53 89
e1 b0 0b cd 80 fc ce ff ff
```

- This is the binary for the machine instructions (x86 CPUs) for the function call on Linux:

```
execve("/bin/sh")
```

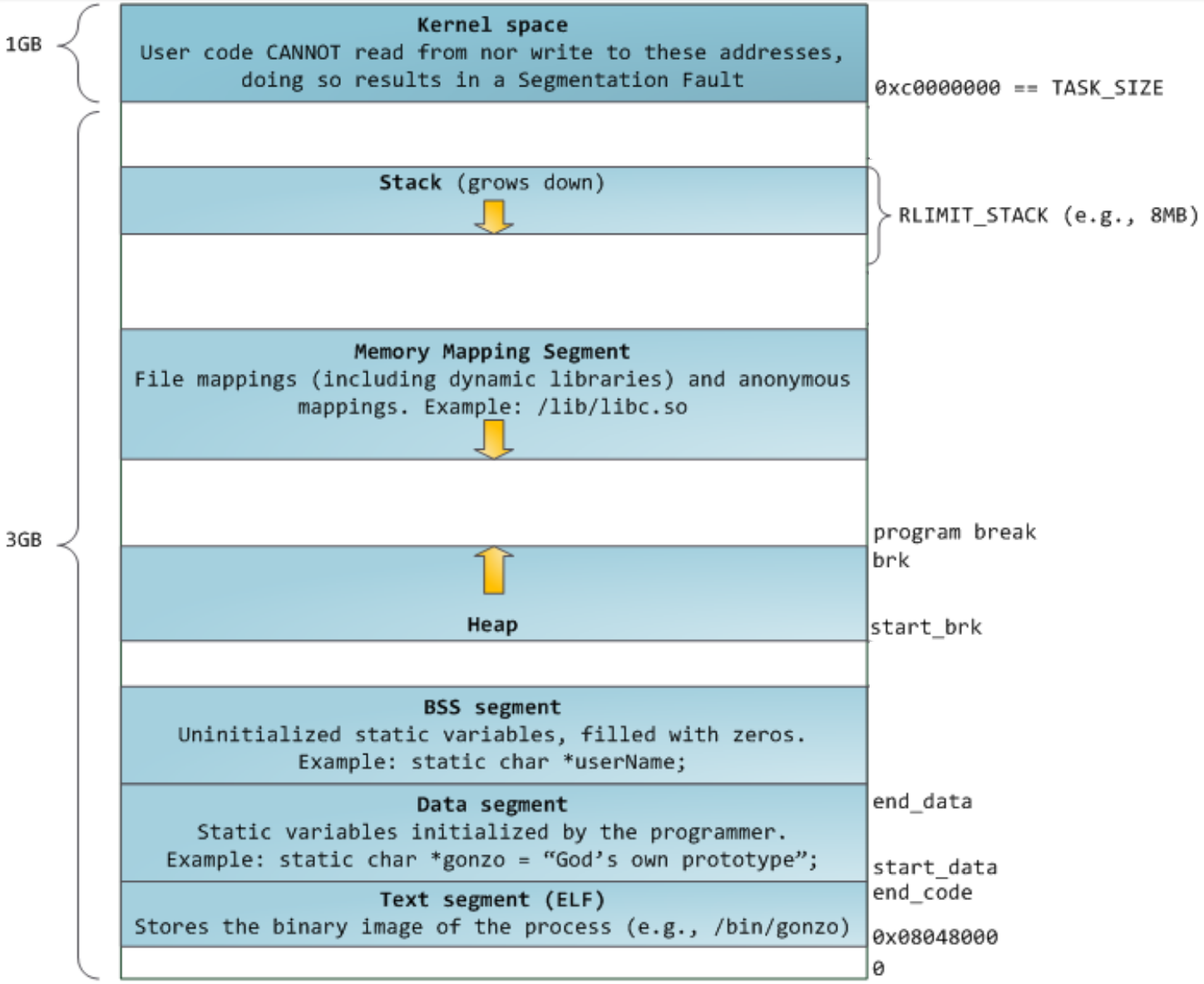
  followed by 4 bytes 0xFFFFCEFC

- In our example, this will write the code into the buffer (and the frame pointer) and then overwrite the return address with 0xFFFFCEFC which is the address of the array buffer
- When function f returns, the CPU will jump to the return address and execute the injected code!

Predicting addresses

- But... How does the attacker know that the buffer will be located at address 0xFFFFCEF?
- Isn't the stack located at a different address depending on how full your memory is when your program was started, how much memory your computer has, etc.?
  - No. Thanks to virtual memory, every process starts in a clean virtual address space with predictable addresses for the code, the stack, etc. set by the OS
- Of course, the address is program dependent. In our example, 0xFFFFCEF is only the correct address if the function f() is called from main().
- See next slide (adapted from http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/)

Virtual address space of a program running on 32-bit Linux



Using buffer overflows

- Typically, the attacker will use the buffer overflow to start a shell where they can
  - execute programs with the same rights as the attacked process.
    If the attacked process (e.g. webserver) was running with root rights, the attacker can do everything
  - start other attacks (buffer overflows,...) against the machine to get root rights

Protection

Avoiding buffer overflows

- First, the most important measure: Avoid them!
  - Languages like Java or C# do runtime checks on the array length

- In C, you should never use `strcpy` (and similar functions like `gets` and `sprintf`)

- Instead, use `strncpy`, `snprintf`,…

```
void f(char *s) {
    char buffer[8];
    strncpy(buffer,s,sizeof(buffer));
}
```

Avoiding buffer overflows (2)

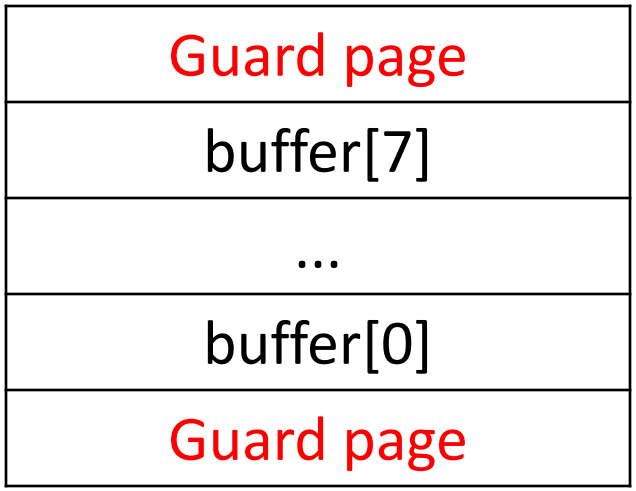
- **Second: Always check and sanitize data coming from outside**

```
int picture[100][100];

void writePixel(int x, int y, int color) {
   picture[y][x]=color;
}
```

- What happens if x or y are negative?
- Easy to see in this example, but difficult in more complex C programs doing pointer arithmetics
- And even if *your* code is save, there are still a lot of old programs and libraries with vulnerabilities...

Adding bound checking to C

- There are tools to add (limited) bound checks to C
  - Cannot catch all cases because of C's flexibility (pointer arithmetics,...)
- "Electric Fences": Put guard pages (virtual memory pages) around the array and tell the OS to raise an exception if somebody tries to access them.
  - Very resource consuming!



Tools to detect vulnerabilities

- Use tools that do a static analysis of the source code to find vulnerabilities
  - Find usages of strcpy,...
  - Find uninitialized variables,...
  - Find suspicious code by symbolic execution: Such tools analyze the source code to discover mathematical properties of variables.

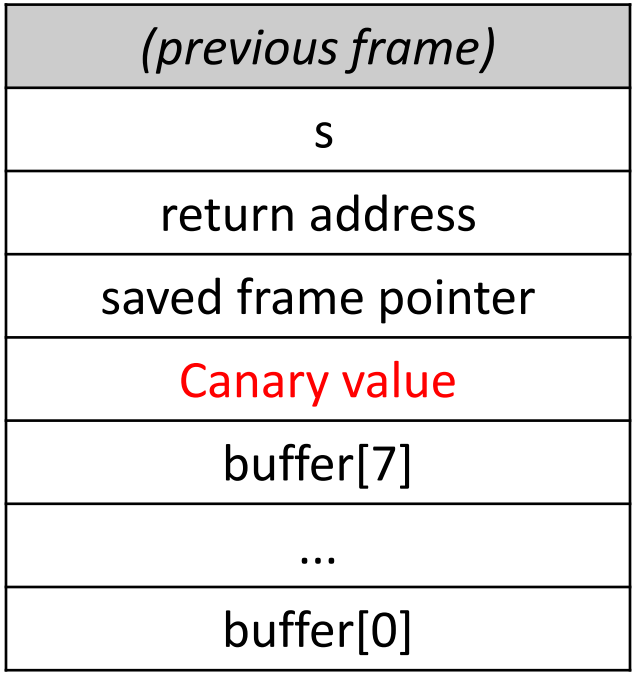
```
int buffer[10];
if(x>20) {
    buffer[x]=1;
}
```

← x is definitely too large here

- Fuzzers: Tools that test a program with random input

Mitigation by Canaries

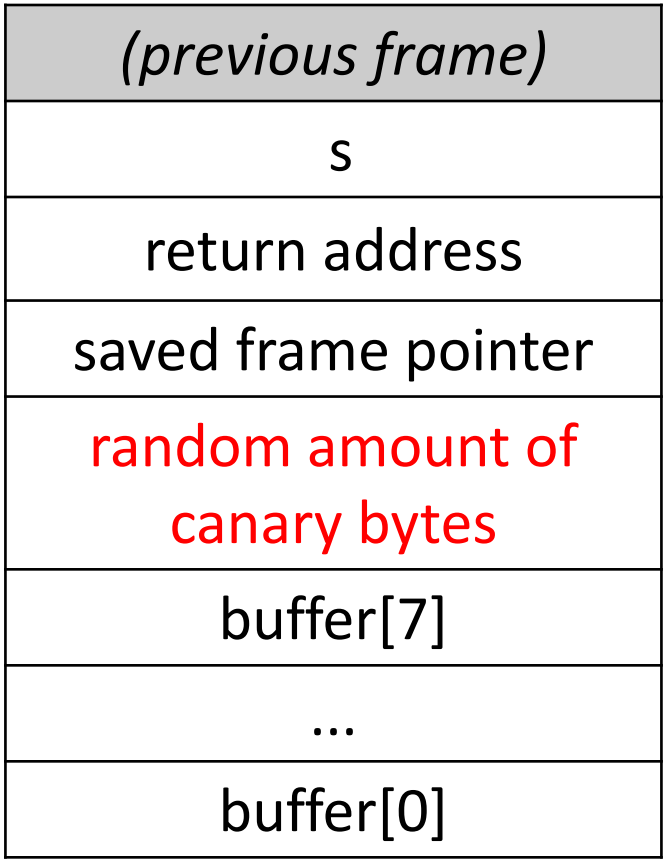
- Compilers can add code to push a canary value onto the stack
- When the function returns, the code checks whether the canary has been overwritten



- Makes your program slightly slower
- Risk: Attacker knows the canary value

Mitigation by Random Canaries

- Compiler can insert random amount of data into the stack frame to make it harder for the attacker to guess the frame layout



Mitigation by Canaries (2)

- Attacker might still be successfull: Guess correctly the size of the random canary
- And canaries do not protect the local variables!
  - Example: Attacker can overwrite a pointer variable

```
int globalVariable;
void f(char *s) {
  int *ptr=&globalVariable;
  char buffer[8];
  strcpy(buffer,s);
  *ptr = 5;
}
```

Mitigation by Canaries (3)

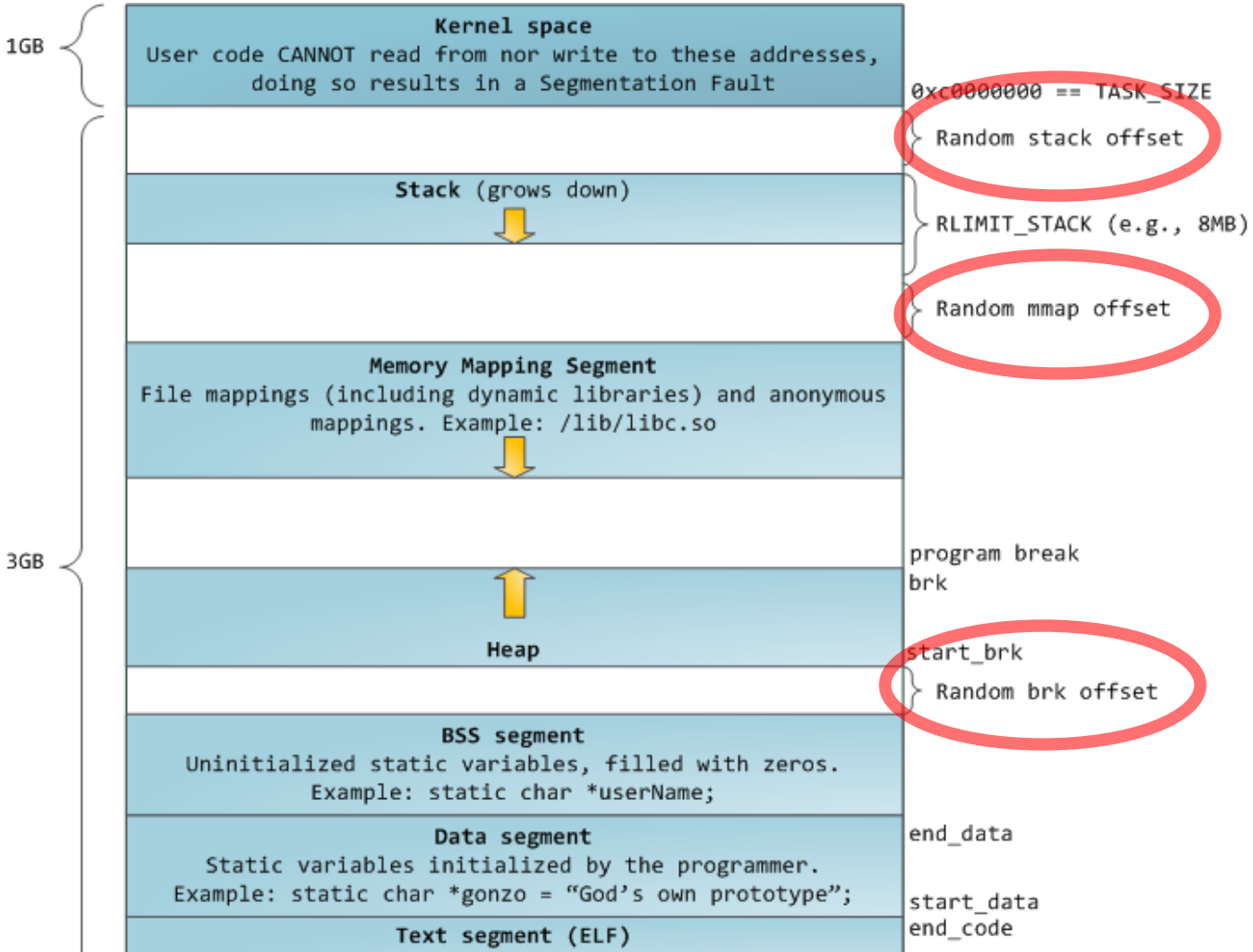
- Attacker can overwrite a function pointer variable to let it point to own code

```
void f(char *s) {
   void (*funcptr)()=...; // some function pointer
   char buffer[8];
   strcpy(buffer,s);
   funcptr();
}
```

- More tricks with overwriting pointers here: http://www.win.tue.nl/~aeb/linux/hh/hh-11.html

Address Space Layout Randomization (ASLR)

- Modern OSs make addresses of stack etc. less predictable for an attacker



ASLR (2)

- ASLR might not work well on CPUs with a small 16-bit or 32-bit address space
  - Random gap cannot be very large
  - And addresses typically start on 2/4/8-byte or even 4096-byte boundaries (pages), further reducing the number of possible values for the gap size

- If the randomization is not large enough, the attacker can succeed by preparing large areas of memory on the heap (several MBytes), hoping that program execution will arrive there ("heap spraying")

Data Execution Prevention (DEP)

- There is no reason why a normal program should be able to execute code on the stack
- Most OS nowadays mark the memory pages where the stack is located as "not executable"

  → Exception triggered when CPU tries to run with IP pointing to that area

- Programs that dynamicaly generate code (JIT of JVM, etc.) have to be adapted
  - First write the code as data into memory
  - Then make the memory location executable

DEP (2)

- How can we do a successful attack if DEP is activated?
  - If DEP does not allow us to write our own attack code, maybe we can call code that already exists in the system?
- Candidate: the library function `system`. This function can be used to start a shell

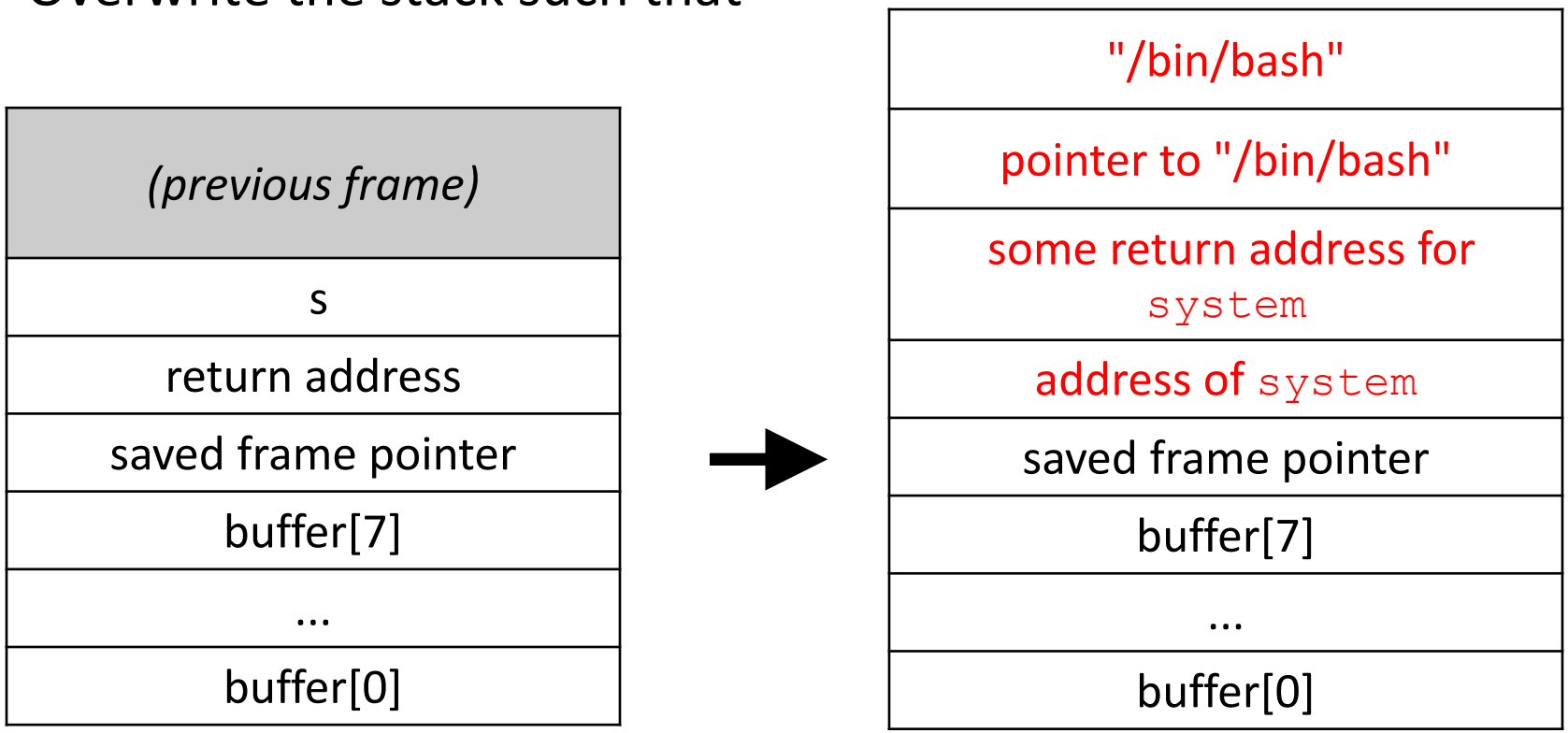
  `system`("/bin/bash")
- We only have to find a way to call the function with parameter "/bin/bash" without writing our own code...

Return-Oriented Programming

- If ASLR is not used, library functions like `system` are located at (version-specific) predictable addresses in the virtual address space of the process
- Since we control the stack contents, it's pretty easy to call the function. We just have to "fake" a function call:
  1. Prepare the stack such that there is "/bin/bash" as parameter
  2. Write the address of the `system` function into `return address`
- As soon as the current function returns, the CPU will jump to `system`
  - It looks like a normal function call for `system`

Return-Oriented Programming (2)

- Overwrite the stack such that



- We can even chain function calls

Conclusions

- When writing new programs:
  - Check input data coming from outside
  - Consider using a modern programming language...

- But: you never know what vulnerabilities the OS and the libraries contain
  - Isolate your server from the rest of the system, to minimize the damage
  - Don't run your web server as root