

Advanced data Security

Part II:

Communication Security

Chapter8:

Software Security - Buffer Overflow

Prepared by:

Dr. Hamssa Hasrouny

Software Security

Lesson Introduction

- **Software vulnerabilities** and how attackers **exploit them**.
 - **Defenses against attacks** that try to exploit buffer overflows.
 - **Secure programming:** Code “defensively”, expecting it to be exploited. Do not trust the “inputs” that come from users of the software system.
-

Software Vulnerabilities & How They Get Exploited

- **Example: Buffer overflow** - a common and persistent vulnerability
- Stack buffer overflows
- **Stacks** are used...
 - in **function/procedure calls**
 - for **allocation of memory** for...
 - local variables
 - parameters
 - control information (return address)



A Vulnerable Password Checking Program



```
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
    char targetpwd[12] = "MyPwd123";
    gets(pwdstr);
    if (strncmp(pwdstr, targetpwd, 12) == 0)
        allow_login = 1;

    if (allow_login == 0)
        printf("Login request rejected");
    else
        printf("Login request allowed");
}
```



Stack Access Quiz

Check the lines of code, when executed, accesses addresses in the **stack frame** for **main()**:

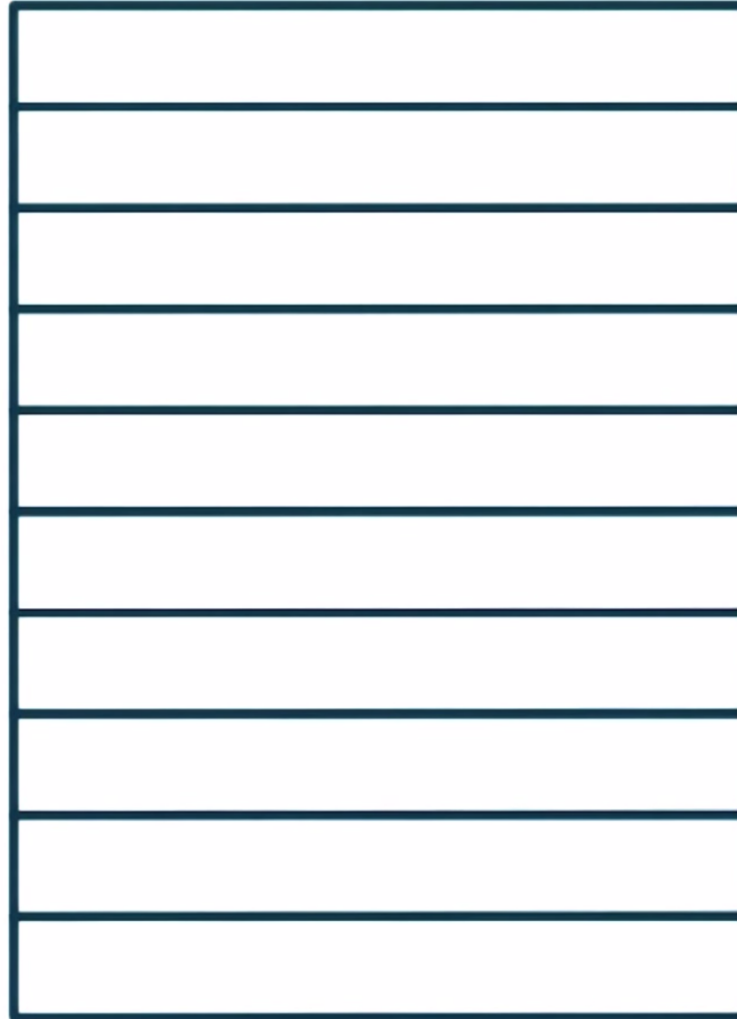
```
int main(int argc, char *argv[]) {  
    int allow_login = 0;  
    char pwdstr[12];  
☐ char targetpwd[12] = "MyPwd123";  
☐ gets(pwdstr);  
☐ if (strncmp(pwdstr, targetpwd, 12) == 0)  
        ☐ allow_login = 1;  
  
☐ if (allow_login == 0)  
        ☐ printf("Login request rejected");  
☐ else  
        ☐ printf("Login request allowed");  
}
```

Understanding the Stack

High Address



Low Address



Attacker Code Execution

We type a **correct password** (**MyPwd123**) of less than 12 characters:



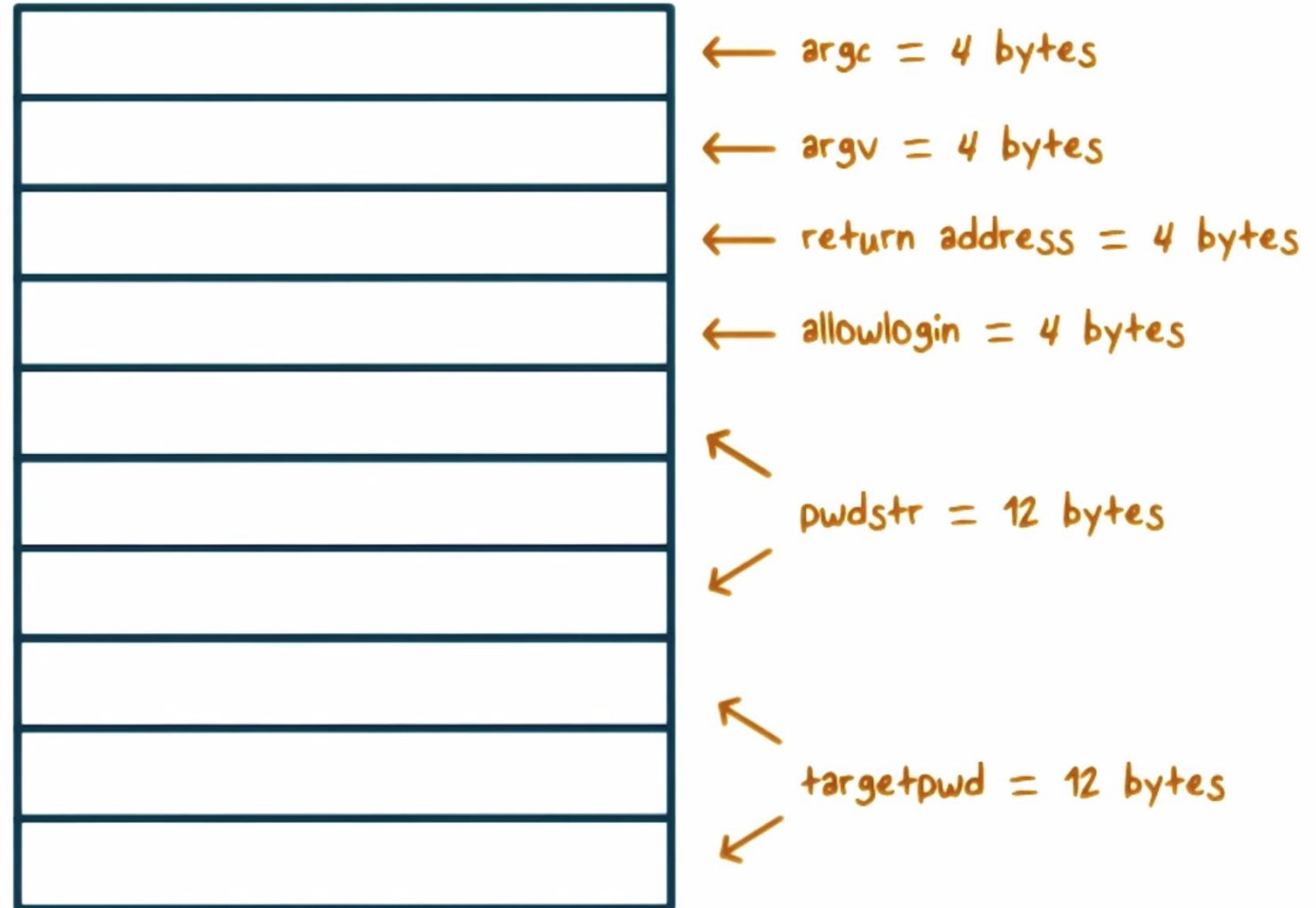
The login request is allowed.

Now let us type “**BadPassWd**” when we are asked to provide the password:



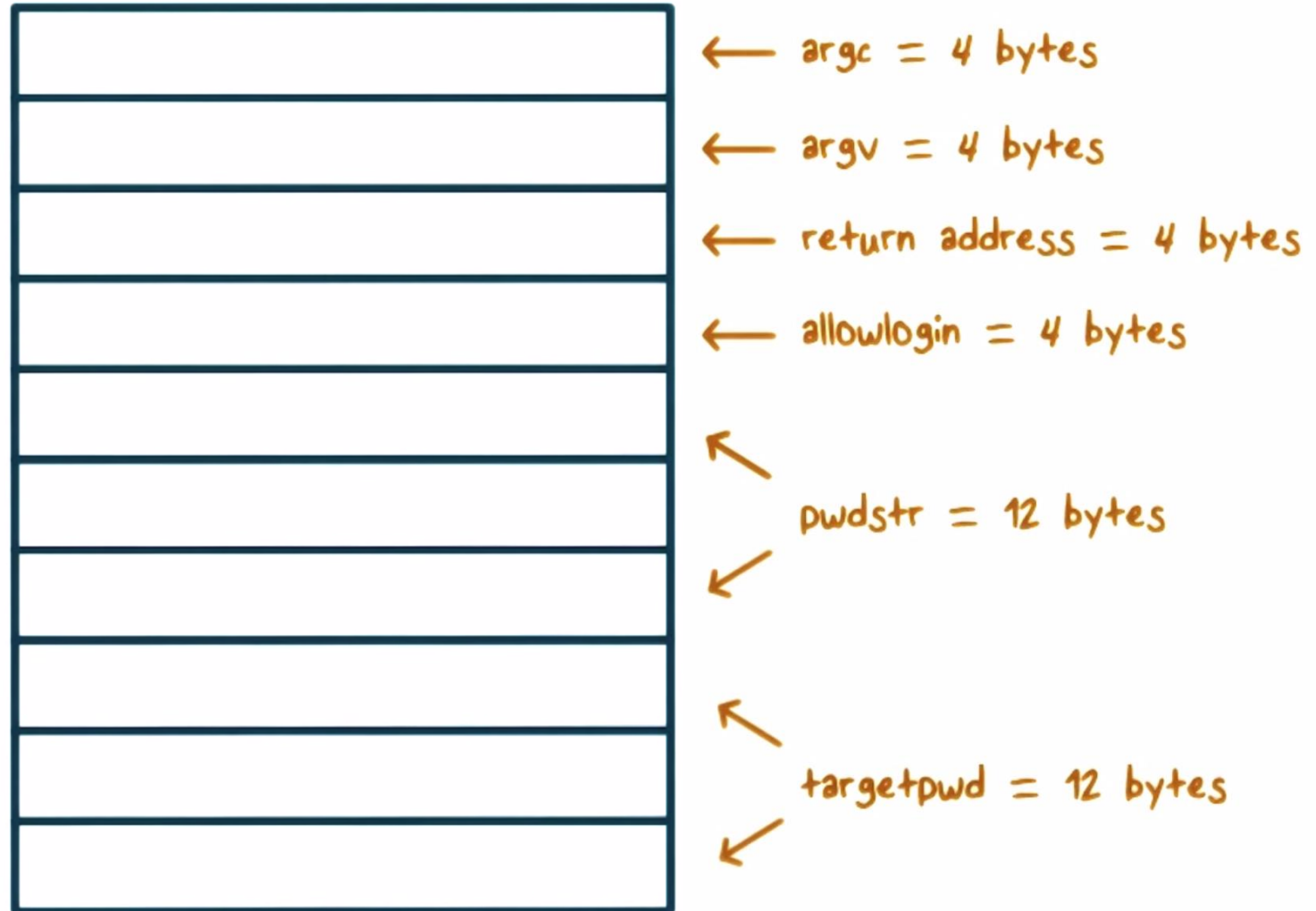
The login request is rejected.

Attacker Code Execution



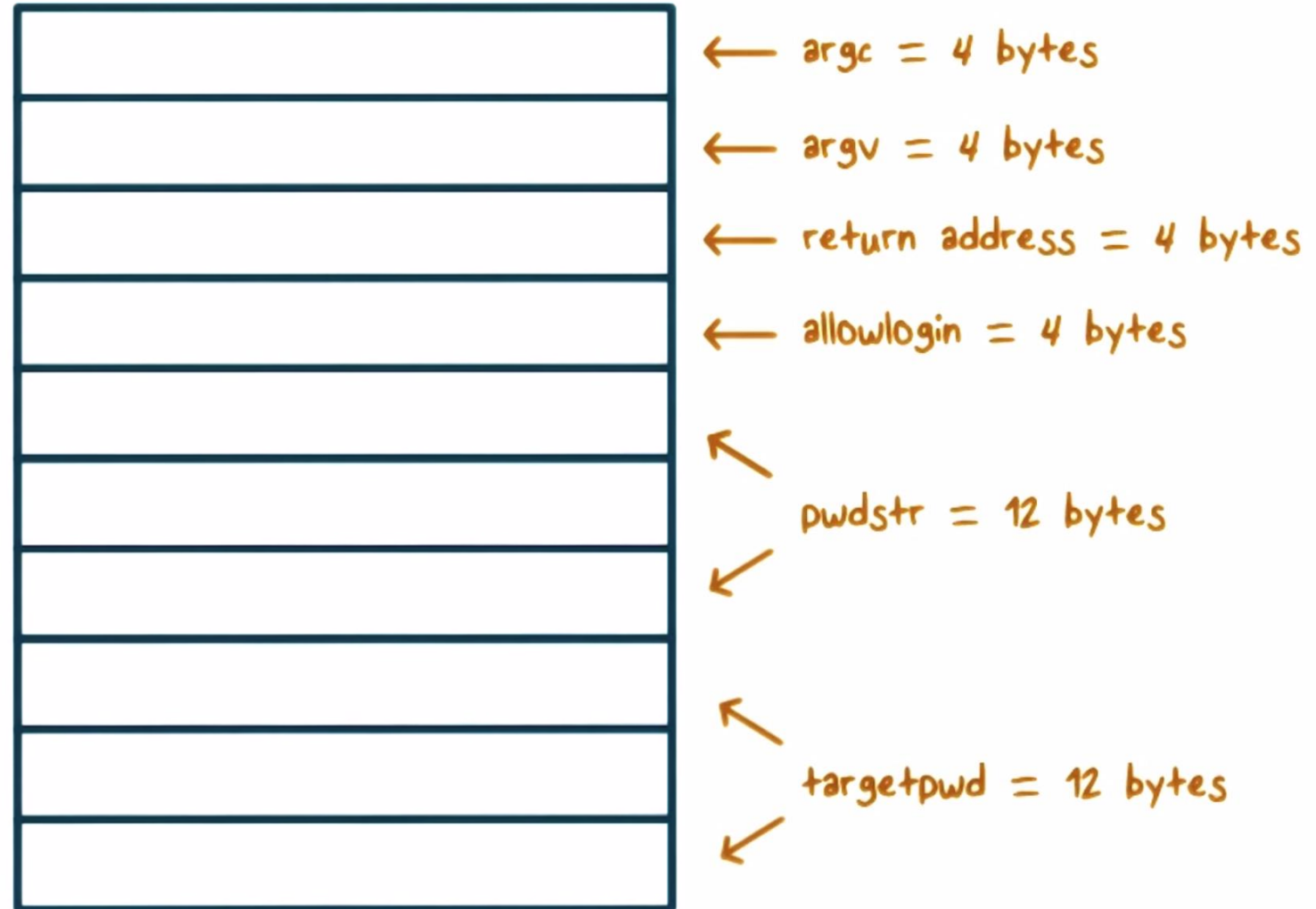
Attacker Code Execution

If we type a really long string, we will **overflow** into the return address space.



Attacker Code Execution

We can carefully **overflow the return address** so it contains the value of an address where we put some code we want executed.





Buffer Overflow Quiz

Which of these **vulnerabilities** applies to the code:

- ☐ The target password was too short, this made it easy to overflow the buffer.
- ☐ The code did not check the input and reject password strings longer than 12 bytes.
- ☐ The code did not add extra, unused variables. If this is done then when the user inputs a long password, it won't overflow into the return address.

ShellCode

Shell Code: creates a shell which allows it to execute any code the attacker wants.

Whose **privileges** are used when attacker code is executed?

- The host program's
- System service or OS root privileges

LEAST Privilege is IMPORTANT



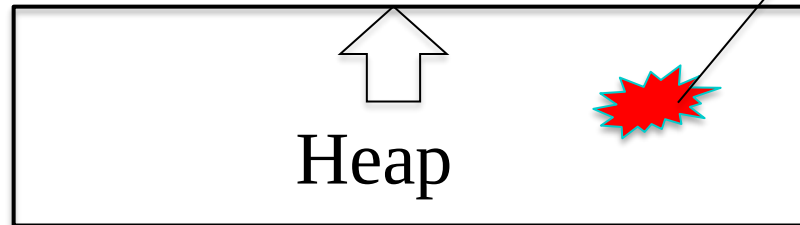
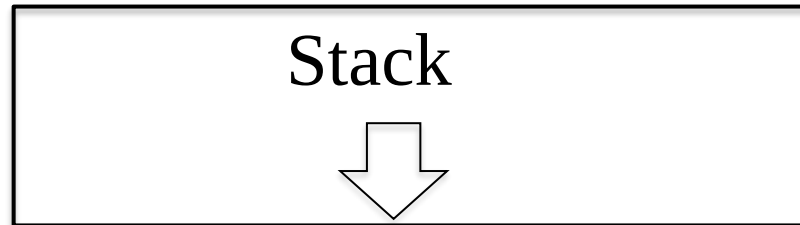
Variations of Buffer Overflow

- **Return-to-libc**: the return address is overwritten to point to a standard library function.
- **Heap Overflows**: data stored in the heap is overwritten. Data can be tables of function pointers.
- **OpenSSL Heartbleed Vulnerability**: read much more of the buffer than just the data, which may include sensitive data.

●Heap Overflow

- Buffer overflows that occur in the heap data area.
 - Typical heap manipulation functions: malloc()/free()

Higher Address

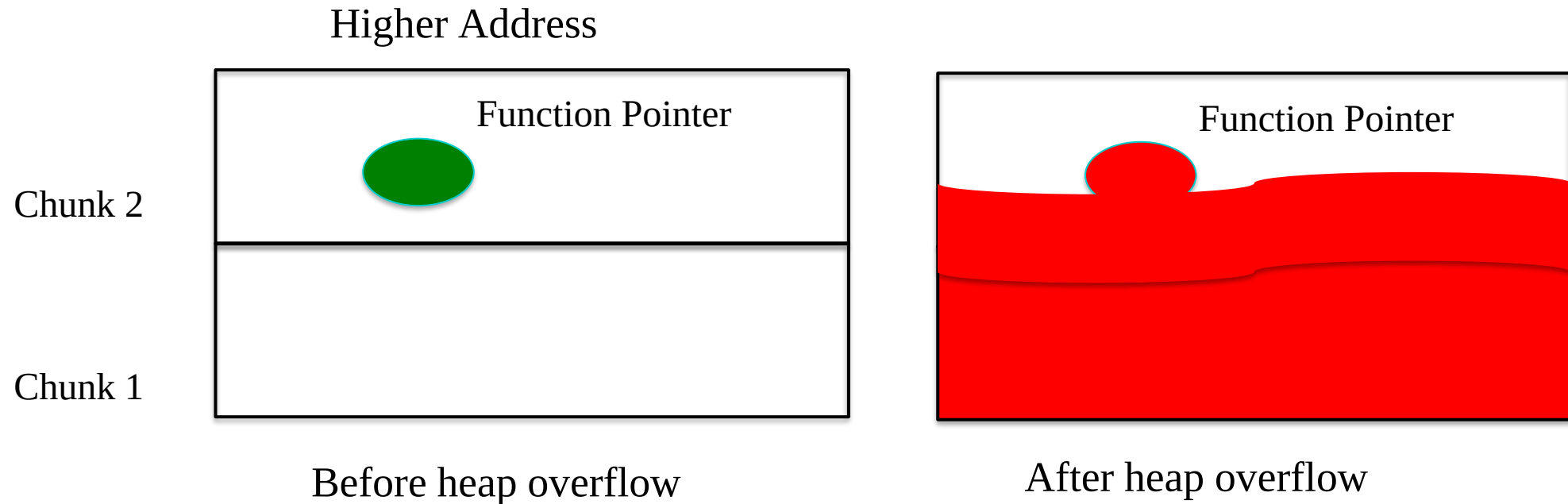


Lower Address

```
char* p = malloc (256);  
memset (p, 'A', 1024);
```

●Heap Overflow – Example

- Overwrite the function pointer in the adjacent buffer



Defense Against Buffer Overflow Attacks

Programming language choice is crucial.

The language...

- Should be **strongly typed**
- Should do **automatic bounds checks**
- Should do **automatic memory management**

Examples of **Safe languages**: Java, C++, Python



Defense Against Buffer Overflow Attacks

Why are some languages safe?

- Buffer overflow becomes impossible due to runtime system checks



The drawback of secure languages

- Possible performance degradation

Defense Against Buffer Overflow Attacks

When Using Unsafe Languages:

- Check input (**ALL input is EVIL**)
- Use safer functions that do **bounds checking**
- Use **automatic tools** to analyze code for potential unsafe functions.



Defense Against Buffer Overflow Attacks



Analysis Tools...

- Can **flag** potentially unsafe functions/constructs
- Can **help mitigate security lapses**, but it is really hard to eliminate all buffer overflows.

Examples of analysis tools can be found at:

https://www.owasp.org/index.php/Source_Code_Analysis_Tools

Thwarting Buffer Overflow Attacks

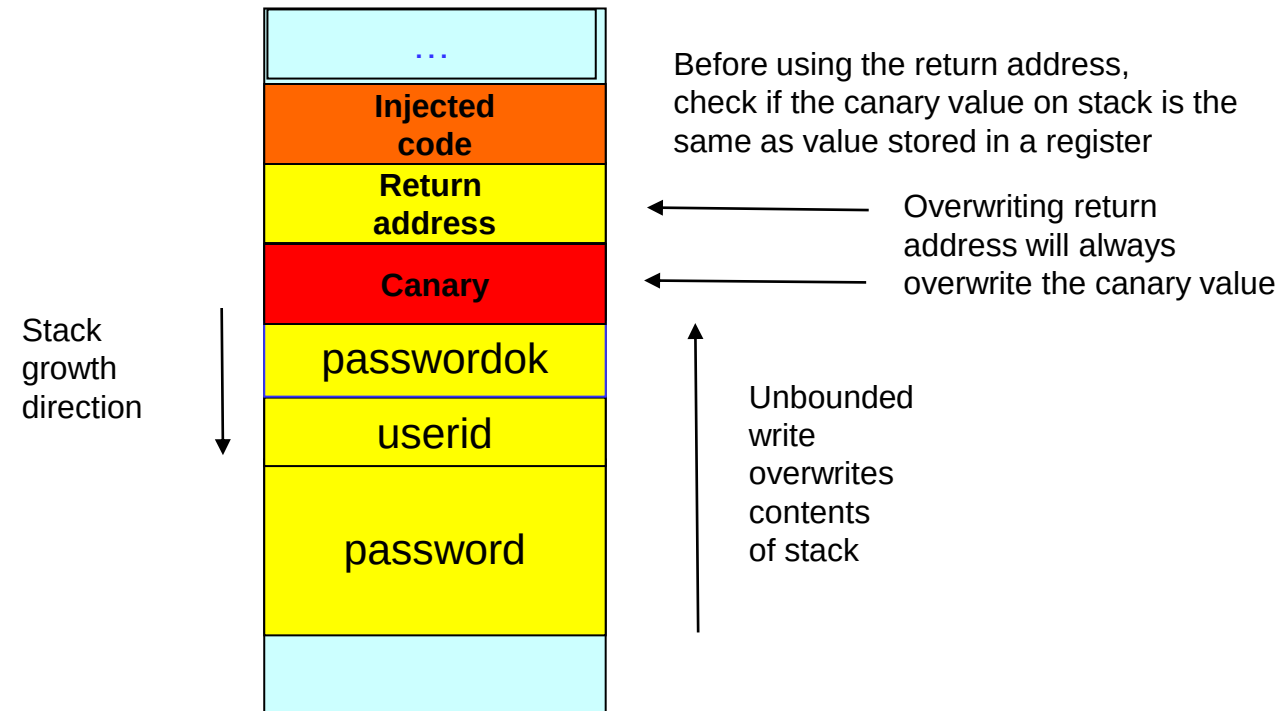
Stack Canaries:

- When a return address is stored in a stack frame, a random **canary value** is written just before it. Any attempt to rewrite the address using buffer overflow will result in the canary being rewritten and an overflow will be detected.



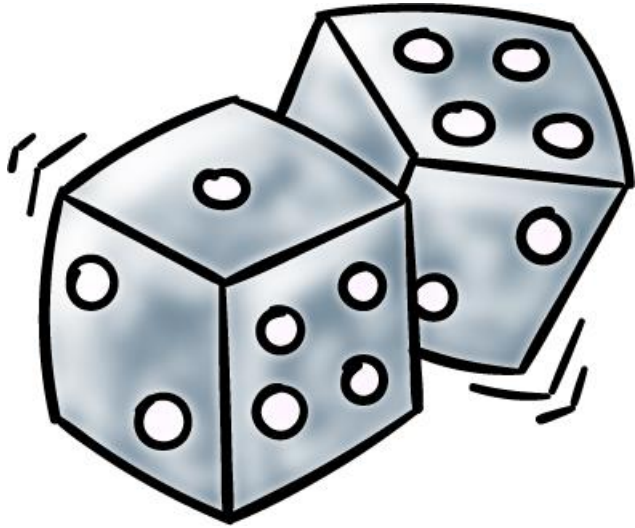
●Countermeasure – Stack Protection

n Canary for tamper detection



n No code execution on stack

Thwarting Buffer Overflow Attacks



- **Address Space Layout Randomization (ASLR)** randomizes stack, heap, libc, etc. This makes it harder for the attacker to find important locations (e.g., libc function address).
- Use a **non-executable stack** coupled with ASLR. This solution uses OS/hardware support.