

## 3. Coding Errors

Prof. Dr. Marc Rennhard  
Institut für angewandte Informationstechnologie InIT  
ZHAW School of Engineering  
rema@zhaw.ch

# Content

- An overview of typical security-relevant coding errors based on the taxonomy of the **7 (+1) Kingdoms**
- A more detailed look at **some particular security-relevant coding errors**
  - Buffer overflows
  - Dangerous functions
  - Race conditions
- Outlook: More on coding errors will follow later when discussing web application security

# Goals

- You know the 7 (+1) kingdoms and can provide some examples about typical errors in each kingdom
- You understand what buffer overflows are, why they happen, how they can be exploited, and how they can be prevented
- You know some dangerous C functions and better suited alternatives for them
- You understand what a race condition is, why they can happen, why they can be security-critical and know some countermeasures to prevent them

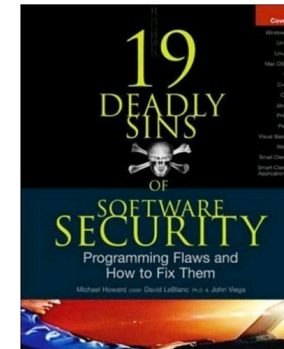
# Coding Errors (1)

- Experience shows that about 50% of all software vulnerabilities are due to coding errors (bugs)
  - This certainly justifies looking at them in more detail
- There's a wide spectrum of different types of coding errors, but they are usually based on a few fundamental problems
  - → Using some kind of classification scheme is reasonable
- Such a classification scheme helps...
  - To understand and structure the entire problem range better
  - To educate and inform software developers about common coding mistakes that have an impact on security in a structured way and to help them reducing them
  - To provide manuals and guidelines for software developers
  - To integrate the scheme into an automated source code analyzer and provide meaningful reports

## Coding Errors (2)

- There exist **several classification schemes** for security-related coding errors

- The 19 Deadly Sins of Software Security
- OWASP Top Ten
- SANS Top-20
- CWE/SANS Top 25 Most Dangerous Programming Errors



- We are using here **McGraws Taxonomy of Coding Errors**
  - Very general classification scheme, not just for one application domain (such as OWASP)
  - It is continuously maintained and extended
  - Uses a reasonable number of main error classes: **7 (+1) Kingdoms**

7 (+1) Kingdoms

## 7 (+1) Kingdoms of Coding Errors

The 7 (+1) kingdoms of coding errors, in order of importance to software security:

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Error Handling
6. Code Quality
7. Encapsulation
- \* Environment

# Kingdom 1: Input Validation and Representation (1)

- Primary problem: **user input is not or not correctly validated** before it is processed
- The solution is basically quite simple: **make sure all input is validated** before it is processed
  - Approaches based on **white lists** are preferable over those using black lists (e.g. define explicitly what is allowed)
  - Example: For a particular form field in a web application, allow at most 20 characters that consist of digits, letters, and space characters
- What makes input filtering difficult is that the same **data can be encoded (represented) in different ways**
  - It therefore may be possible for an attacker to circumvent a filtering mechanism by encoding the attack data such that only legitimate characters are used



# Kingdom 1: Input Validation and Representation (2)

Some examples associated with a lack of input validation:

- **Buffer overflows**
  - Writing data beyond an allocated buffer can allow an attacker to modify the program flow, crash the program, inject (malicious) own code etc.
- **Various injection attacks** (command injection, SQL injection, XML injection...)
  - Allows, e.g., execution of malicious commands on behalf of an attacker or execution of arbitrary SQL statements in the backend database
- **Cross-site scripting**
  - Allows an attacker to execute JavaScript code in the browser of another user to steal credentials, hijack a session etc.
- **Path traversal**
  - May allow a user to access arbitrary files on the target computer

## Kingdom 2: API Abuse

- API (application programming interface) abuse means a programmer...
  - ...does **not use** an API (e.g. a function or a method) **correctly**
  - ...makes **incorrect assumptions** about the offered functionality

### Some examples of API abuse:

- **Dangerous functions**
  - Some functions or methods simply can't be used in a secure fashion and should therefore not be used at all (e.g. `gets` in C)
- **Unchecked return values**
  - Ignoring a function's return value can cause the program to overlook unexpected situations (e.g. a reference containing NULL instead of referencing a valid object because something went wrong)
- **Wrong security assumptions**
  - E.g. relying on `gethostbyaddr` in C or `getHostName` in Java to authenticate a remote host (it's relatively easy to modify DNS responses)

## Kingdom 3: Security Features

- When using **security functions** (or security features) in your program, a lot can still go wrong
  - If possible, **use existing approaches/components/products** that have demonstrated to work well in practice
  - Example: Don't try to invent your own cryptographic algorithm or secure communication protocol – you'll most likely fail

**Some examples** of poor usage of security features:

- **Insecure randomness**
  - Using poor pseudo random number generators (or seeding secure ones with predictable values) will result in weak key material
- **Incomplete access control**
  - A program that does not consistently perform access control will likely allow non-privileged users access to restricted functionality and/or data
- **Weak encryption**
  - Even communication protocols that are considered secure often support older algorithms due to backward compatibility (e.g. DES or MD5 in TLS)

## Kingdom 4: Time and State (1)

- Time and State-related issues may happen if multiple systems, processes or threads interact and share data
  - E.g. with distributed systems, multithreading or multiple processes in the same system
- One reason why these problems occur is because humans (developers) think about programs as if they were executing the job manually
  - E.g. the client sends data, it arrives at the server, the server application reads it all from the buffer, processes it...
  - Then the next client sends data, it arrives at the server...
  - In this world, execution of the program steps and handling of multiple tasks is sequential and well-defined and time and state-related issues rarely occur
- But computers work differently:
  - Tasks are not processed one after another, but (quasi-)parallel
  - As a result, problems may occur due to unforeseen interactions between tasks

## Kingdom 4: Time and State (2)

Some examples of time and state-related issues:

- **Deadlock**
  - Poor usage of locking mechanisms can lead to deadlock (and therefore availability problems)
  - A law in Kansas once stated: “When 2 trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
- **File access race condition: TOCTOU (time of check – time of use)**
  - The window of time between checking a file property and using the file may be exploited to launch a privilege escalation attack
- **Failure to begin a new session upon authentication**
  - Using the same session identifier across authentication boundaries (e.g. in a web application) may allow an attacker to hijack authenticated sessions

## Kingdom 5: Error Handling

- Error handling concepts of modern languages (e.g. exceptions) are powerful, but they are **difficult to implement correctly**
  - Introduces a second control flow, jumps between exception handlers...
  - As a result, errors are often not handled correctly or not at all

**Some examples** of error handling issues:

- **Empty catch block**
  - Ignoring exceptions may allow an attacker to provoke unexpected program behavior
- **Overly broad catch block** (e.g. catching `Exception` in Java)
  - If the program grows and new types of exceptions are thrown, the new exception types will likely not receive any attention
- **Leakage of internal information**
  - Error messages sometimes propagate internal information (system state, failed database queries...) to the attacker, which provides information for subsequent attacks

## Kingdom 6: Code Quality (1)

- Poor code quality increases the likelihood of erroneous code, which increases the probability that security vulnerabilities creep in
- Poor code quality is caused by the following
  - Unreadable code (very difficult to analyse and maintain)
    - Poor names for variables, methods etc.
    - Too long and too complex classes or methods
    - High coupling among classes
    - Forgetting to remove old code
    - ...
  - Because the developer is not careful enough during programming and does not think about various details, e.g.:
    - I'm allocating a resource – where is it freed again?
    - I'm using these three objects / variables – can I be sure they were in fact initialized before?

## Kingdom 6: Code Quality (2)

Some examples of problems associated with code quality:

- **Memory leak / exhaustion**

- Memory is allocated but never freed, leading to memory exhaustion – and eventually to the termination of the program
- May happen explicitly (`malloc` and `free` in C) or implicitly (filling a `StringBuffer` in Java until all memory assigned to the JVM is used up)

- **Unreleased resource**

- A program that fails to release system resources (file handlers, sockets...) may exhaust all system resources and fail to function properly

- **Null dereference**

- A program dereferences a null pointer, which usually results in program termination
- Often a consecutive fault due to another mistake, e.g. because the programmer hasn't checked the return value of a method or function



## Kingdom 6: Code Quality (3)

- **Uninitialized variable**

- A program that uses a variable before it has been initialized may result in unpredictable behavior
- Today, compilers often warn about this, but warnings are easily ignored (which you never should do, warning nearly always make sense!)

- **Deprecated code**

- Many programming languages contain deprecated classes, methods or functions, which should no longer be used for various reasons
  - New naming conventions, poor interface design, but also due to security flaws!
- Using deprecated classes, methods or functions hints at **neglected code**
- Compilers usually warn about the usage of deprecated components
  - Always try to get rid of the deprecated component by replacing it – a newer variant is usually available

# Kingdom 7: Encapsulation

- Encapsulations is about having **strict boundaries** between users, programs, and data
  - E.g. make sure that one user of a web application cannot access the data of another current user

**Some examples** of problems associated with encapsulation include:

- **Wrong usage of hidden form fields**
  - Hidden fields are basically a useful feature to include data in a web form that should not be visible by the user
  - But don't use hidden fields in web forms to store important session information – it can easily be read and manipulated by an attacker
- **Cross-Site Request Forgery**
  - Allows an attacker to make arbitrary HTTP requests in another users authenticated web session unless this explicitly prevented (e.g. by using a user-specific secret/token)

# Kingdom \*: Environment

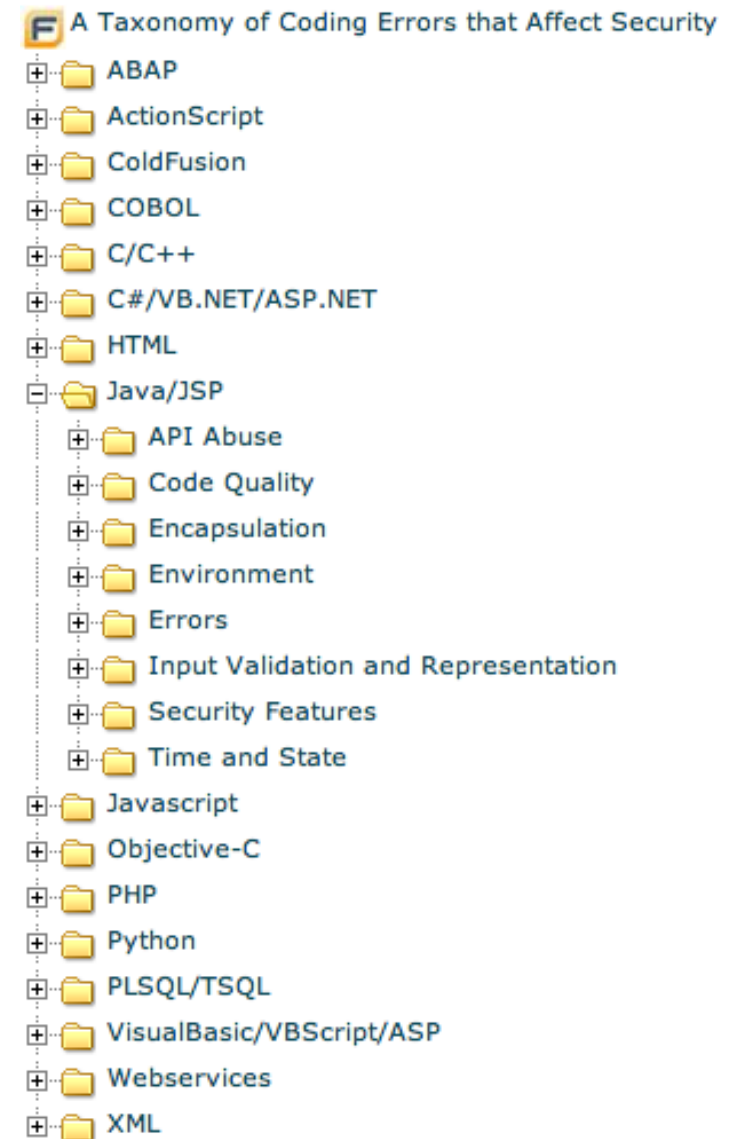
- The software you write does not run by itself, but usually **relies on other software**
  - Compilers, operating systems, execution environments (JVM, .NET...)
  - Libraries, other software on other computers (in a networked environment), network services (DNS...),...
- The environment includes all of the stuff that is **outside of your code but still critical to the security** of the software you create

**Some examples** of problems associated with the environment:

- **Insecure compiler optimization**
  - Improperly removing (overwriting) sensitive data (e.g. a password or secret key) can compromise security
- **Issues with respect to web application frameworks**
  - Insufficient session-ID length or randomness (may allow session-ID guessing attacks)

## Further Information about Coding Errors

- The taxonomy presented here is available online
  - <http://www.hpenterprisesecurity.com/vulncat/en/vulncat/index.html>
  - It's organized according to programming languages and the kingdoms
  - It is continuously maintained and extended with
    - new types of coding errors as new attack variants are uncovered
    - new programming languages when they become popular
- It's a good idea to [have a look at this information](#) when you are developing an application



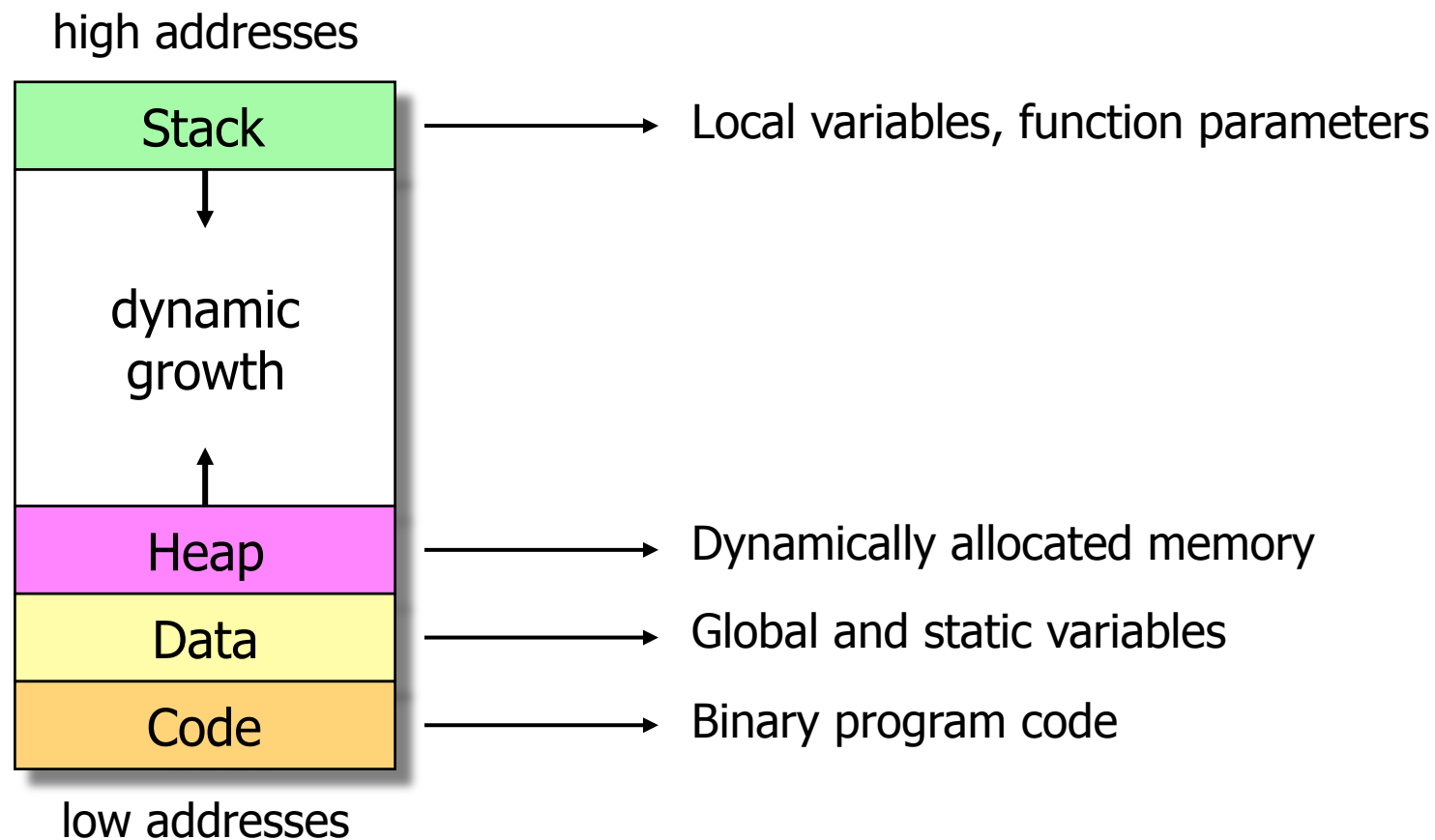
## Buffer Overflows

# Buffer Overflows

- Buffer overflow attacks **exploit programming bugs**
  - Fundamental problem: A program writes data submitted by a user/attacker **beyond the end of an allocated buffer** in memory
  - Allows an attacker to **modify** the program flow, **crash** the program, **inject** (malicious) own code etc.
  - Part of the “Input Validation and Representation” kingdom
- Despite advances with respect to countermeasures, buffer overflow attacks are **still frequently used** and many prominent malware incidents (past & present) exploited buffer overflow vulnerabilities:
  - 1988: **Morris worm**, 2001: **Code Red**, 2003: **SQLSlammer**...
  - ...2010: **Stuxnet**, 2012 **Flame**
- To understand buffer overflow attacks, one must understand the **memory layout** of a process and how the **stack** is used

# Typical Memory Organisation

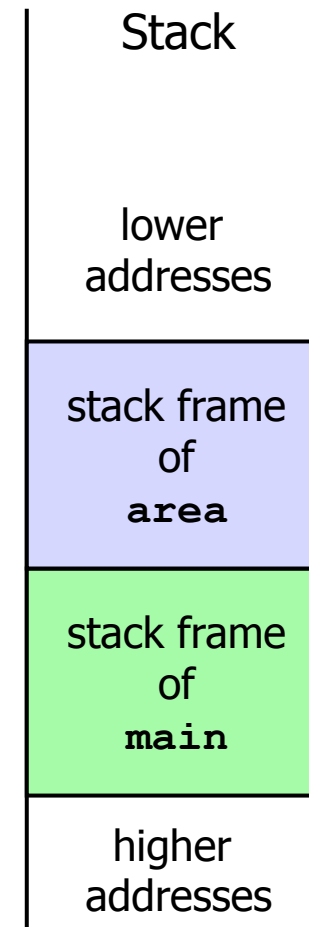
- Every process runs in its **own virtual address space**
- The address space of a process can be separated into **segments**



# The Stack

- The “most prominent” place to carry out buffer overflow attacks
- Simple example in C:

```
int area(int length, int width) {  
    int scale;  
  
    scale = 3;  
    return (scale * length * width);  
}  
  
int main() {  
    int a, b, res;  
  
    a = 5;  
    b = 2;  
    res = area(a, b);  
    return 0;  
}
```





# The Stack – Example – main Function (1)

Source code:

```
int main(

    int a, b, res;

    a = 5;
    b = 2;

    res = area(a, b);

    return 0;

}
```

Assembler code:

```
main:
push ebp
mov  ebp,esp

sub  esp,0Ch

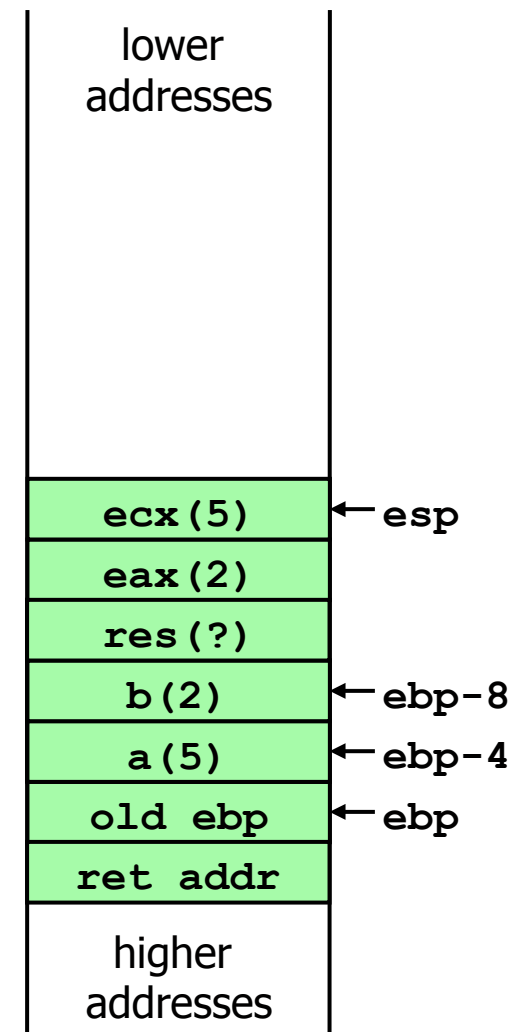
mov  dword ptr [ebp-4],5
mov  dword ptr [ebp-8],2

mov  eax,dword ptr [ebp-8]
push eax
mov  ecx,dword ptr [ebp-4]
push ecx

call area
add  esp,8
mov  dword ptr [ebp-0Ch],eax

xor  eax,eax
mov  esp,ebp
pop  ebp
ret
```

Stack before  
**call area:**



# The Stack – Example – area Function

Source code:

```
int area(
    int length,
    int width) {

    int scale;

    scale = 3;

    return (scale *
        length * width);

}
```

Assembler code:

```
area:
push ebp
mov  ebp,esp

sub  esp,4h

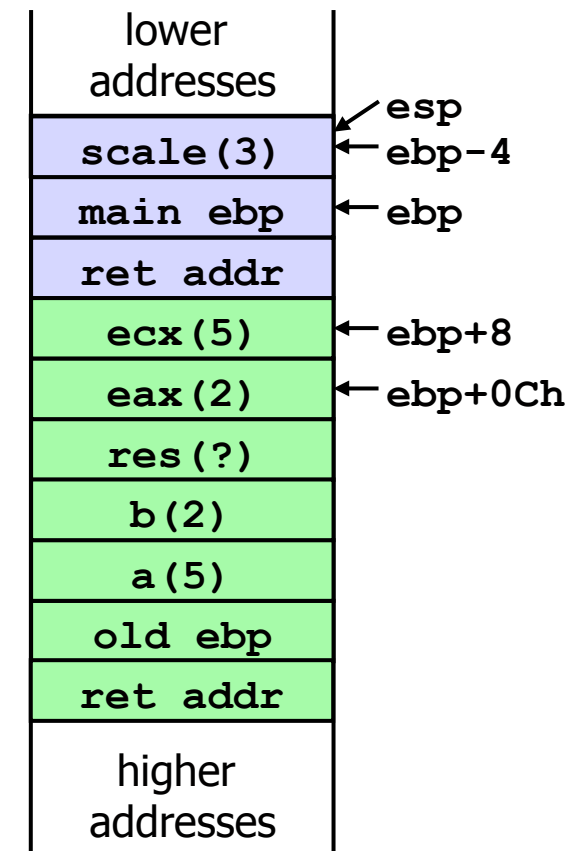
mov  dword ptr [ebp-4],3

mov  eax,dword ptr [ebp-4]
imul eax,dword ptr [ebp+8]
imul eax,dword ptr [ebp+0Ch]

mov  esp,ebp
pop  ebp
ret
```

Stack during

**call area:**



- After returning from the **area** function:
  - The return value is stored in register **eax**
  - The **area** stack frame has been removed
  - Program execution continues in **main** after the **call area** instruction

# The Stack – Example – main Function (2)

Source code:

```
int main(

    int a, b, res;

    a = 5;
    b = 2;

    res = area(a, b);

    return 0;

}
```

Assembler code:

```
main:
push ebp
mov  ebp,esp

sub  esp,0Ch

mov  dword ptr [ebp-4],5
mov  dword ptr [ebp-8],2

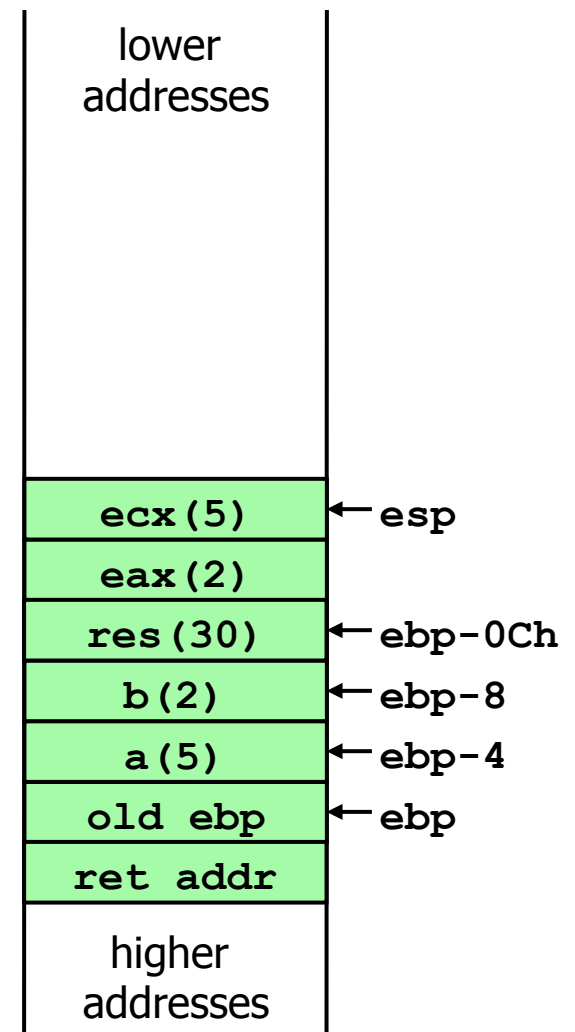
mov  eax,dword ptr [ebp-8]
push eax
mov  ecx,dword ptr [ebp-4]
push ecx

call area
add  esp,8
mov  dword ptr [ebp-0Ch],eax

xor  eax,eax
mov  esp,ebp
pop  ebp
ret
```

Stack after

**call area:**



# Why do Buffer Overflows happen? (1)

- Buffer overflows can only happen with languages such as C, C++ that do **not check array boundaries** during runtime
- With languages/environments such as Java and .NET, it is not possible to write a program that itself has a buffer overflow vulnerability
  - Because the runtime environment (e.g. JVM) **checks every array access** and makes sure it happens “within the array boundaries”
  - Access beyond array boundaries is prevented (e.g. **ArrayIndexOutOfBoundsException**)
- But: the **underlying virtual machine** (usually written in C or C++) may actually contain buffer overflow vulnerabilities (this has happened)
  - Which may allow an attacker to write a specifically crafted Java program which exploits the vulnerability to get, e.g., access to the underlying operating system

## Why do Buffer Overflows happen? (2)

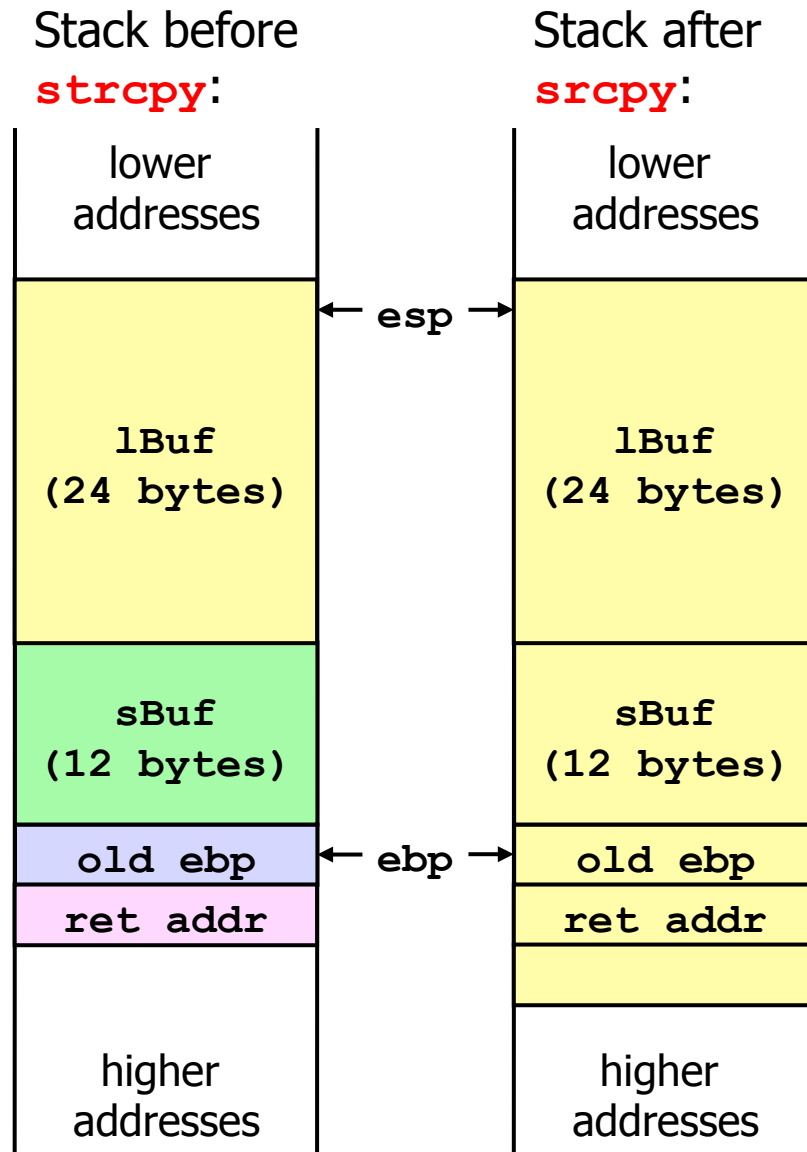
- This works in C without compile/runtime-error:

```
void function() {  
    char sBuf[12];  
    char lBuf[24] = "A looooooooooong string!";  
  
    strcpy(sBuf, lBuf); /* Copies lBuf to sBuf */  
}
```

- In C, local array variables are stored on the stack (unlike Java)
- Copies the 24 characters from lBuf into the 12-character array sBuf
- This simply overwrites the 12 characters following sBuf on the stack
- Overflows the buffer sBuf → therefore the name buffer overflow

## Effect on Program Execution

- 24 bytes from `lBuf` are written into `sBuf`
- This overwrites `old ebp`, `ret addr` in the current stack frame (and an additional 4 bytes)
- When returning from function, the program jumps to what is stored in `ret addr`
- This is not what was originally stored there → arbitrary behaviour
- Probably: segmentation fault as the CPU likely tries to access memory locations it is not allowed to access



# How to Exploit a Buffer Overflow (1)

- The previously discussed buffer overflow **cannot be exploited** by an attacker, as a fixed string is copied
  - Will probably easily be detected during testing, as it always occurs when `function` is called
- Much more interesting: buffer overflows, **where the user can submit the data** to be copied into a buffer
  - Command-line arguments
  - Input submitted to a program that is executed locally
  - **Most interesting**: input submitted to a program over the network
- Scenario:
  - **Networked application**, the server gets data (char array) from the client
  - The data is read and processed on the server in a function `processData`
  - The programmer **assumes the client sends at most 256 bytes** and does therefore not check the input length

## How to Exploit a Buffer Overflow (2)

```
void processData(int socket) {
    char buf[256];
    char tempBuf[12];
    int count = 0, pos = 0;

    /* Read data from socket and copy it to buf */
    count = recv(socket, tempBuf, 12, 0);
    while (count > 0) {
        memcpy(buf + pos, tempBuf, count)
        pos += count;
        count = recv(socket, tempBuf, 12, 0);
    }

    /* Do something with buf */
    ...

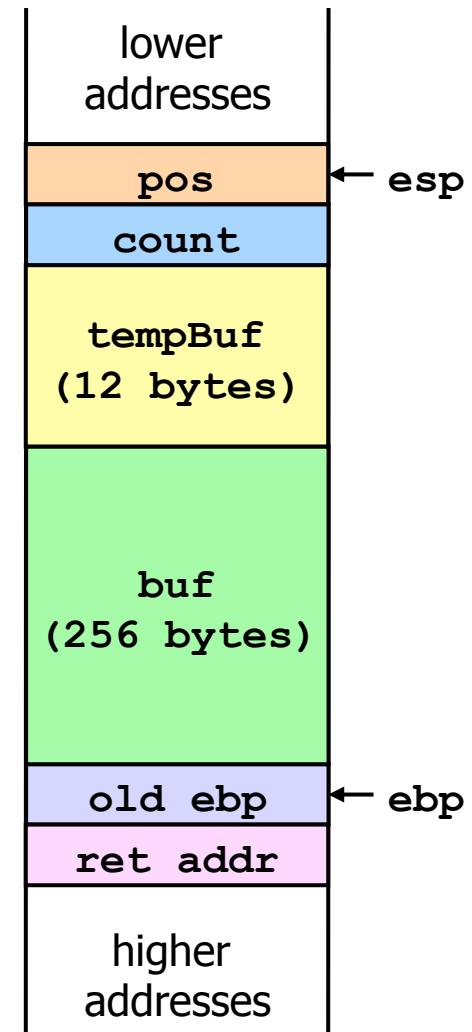
    /* Return */
}
```



## How to Exploit a Buffer Overflow (3)

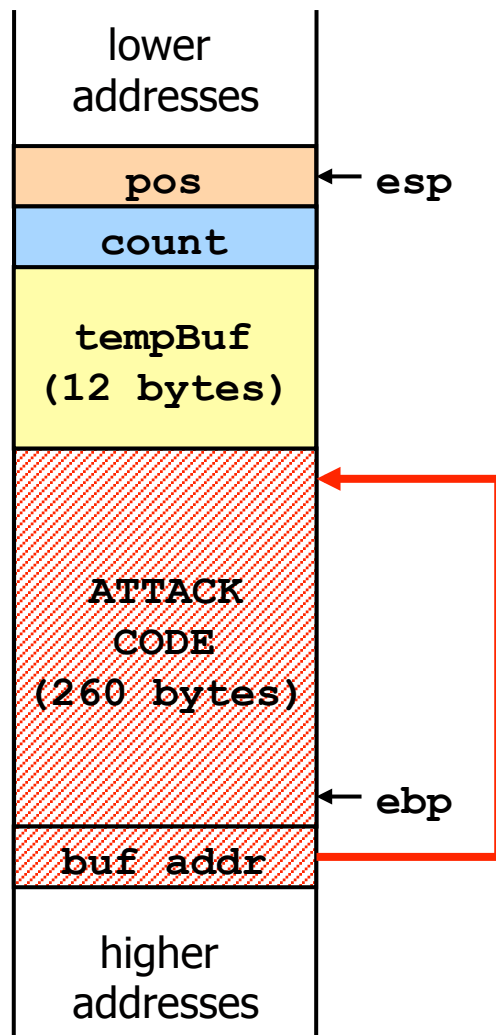
- What happens if **more than 256 bytes** are sent by the client
  - the 257<sup>th</sup> - 260<sup>th</sup> bytes overwrite **old ebp**
  - the 261<sup>st</sup> - 264<sup>th</sup> bytes overwrite **ret addr**
- A **user accidentally submitting** more than 256 bytes of data will probably “just” crash the server program
- An **attacker** can do the following:
  - Submit **264 bytes data**
  - The first 260 bytes contain **code crafted by the attacker**
  - The 261<sup>th</sup> - 264<sup>th</sup> bytes contain the **address of buf**
  - If the function is left (**ret**), program execution **jumps to the start of buf** and executes the code submitted by the attacker!

Stack frame layout:



# How to Exploit a Buffer Overflow (4)

Stack frame layout  
after **code injection**:



- The attacker can **insert any code** that fits into the available memory range:
  - Access the local file system
  - Create a user account
  - Spawn a new process
  - Download and install software (malware/spyware/backdoor...)
  - ...
- The inserted code is run with the **privileges of the exploited program**
  - This is one reason you should run software with minimal privileges

# Buffer Overflows – Countermeasures (1)

- **Good programming techniques**
  - Make sure buffer overflows don't happen by checking the length of submitted data (input validation)
  - Avoid unsafe C/C++ functions (e.g. `gets`) and use the corresponding safer alternatives whenever possible (e.g. `fgets`)
- **Automated software tests**
  - Static code analysis
  - Fault injection: submit various inputs and analyse program behaviour
- **Compiler checks and extensions**
  - Some compilers can insert additional code to check boundaries when data is copied into a buffer, but this is only possible if the compiler can determine the buffer size
  - Stack-canaries to prevent overwriting of the return address

## Buffer Overflows – Countermeasures (2)

- **No eXecute (NX) bit / eXecute Disable (XD)** (SPARC, IA64, AMD64...)
  - Disallows executing program instructions in certain memory areas (stack)
  - Specified in the paging table (mapping virtual to physical memory) using one of the 64 bits
  - E.g.: part of Linux kernel since 2.6.7
- **Address Space Layout Randomisation (ASLR)**
  - Randomises the address layout of a process when it is loaded, e.g. part of Linux kernel since 2.6.12, is constantly improved
  - Causes the segments (code, heap, stack...) to be placed at **non-predictable locations** in the virtual memory space
  - Prevents an attacker from predicting the address layout, which is a prerequisite for many buffer overflow attacks
  - Works against a wide variety of buffer overflow attacks that go beyond inserting code into the stack, such as:
    - **Manipulate program flow only** by overwriting the return address
    - Only **overwrite a pointer variable** such that it points to a different address
    - Exploit memory organisation on the heap

## Dangerous Functions

# Dangerous Functions (1)

- Notably in C and C++, some functions are **dangerous to use** and should be avoided
- Many of these functions **work on strings** (in C, this is a char array terminated with a null byte) and are prone to buffer overflows
- Part of the "API Abuse" kingdom
- Example:

```
char *gets(char *s) ;
```

The *gets()* function shall read bytes from the standard input stream, stdin, into the array pointed to by *s*, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall be discarded and a null byte shall be placed immediately after the last byte read into the array.

## Dangerous Functions (2)

- Consider the following simple program that uses `gets`:

```
#include <stdio.h>

main() {
    char buffer[25];
    printf("\nEnter Text : ");
    gets(buffer);
}
```

- What happens if we enter 10 or 25 characters?

```
rennhard@ubuntu-generic:~$ ./gets
```

```
Enter Text : 1234567890
```

```
rennhard@ubuntu-generic:~$ ./gets
```

```
Enter Text : 1234567890123456789012345
```

```
*** stack smashing detected ***: ./gets terminated
```

10 characters OK  
(requires 11 bytes  
in buffer)

25 characters NOK  
(requires 26 bytes  
in buffer)

## Dangerous Functions (3)

- The example on the previous slides also shows that gcc (on Ubuntu) per default has the **stack protection feature enabled**
  - gcc uses a stack canary approach
  - That's why it states "stack smashing detected"
- **Disabling the feature results in a "normal" segmentation fault**, which is a typical result of a buffer overflow

```
rennhard@ubuntu-generic:~$ gcc -fno-stack-protector -o gets gets.c
/tmp/ccEte5v3.o: In function `main':
gets.c:(.text+0x24): warning: the `gets' function is dangerous and should not be
used.
rennhard@ubuntu-generic:~$ ./gets

Enter Text : 1234567890123456789012345
Segmentation fault
```

- Sidenote: Depending on the used version, the compiler may issue a warning about using gets



## Dangerous Functions (4)

- **That's great news** – we have stack canaries so there's no need to avoid unsafe functions such as `gets`
- **Nope**, because...
  - The stack canary is only included during **compilation**, but one can never know what compiler is going to be used with what options
  - There may be **flaws** in the stack canary approach that can be used to defeat it (e.g. if the canaries can be predicted by the attacker)
  - The program **still crashes** when a buffer overflow occurs, which leads to availability problems
- As a result, the only truly secure way is **get rid of the bug where it occurred**: in the code
  - Additional security measures such as stack canaries (and other buffer overflow protection mechanism) should therefore **only be considered as a second line of defense** (i.e. defense in depth)

## Dangerous Functions (5)

- A better alternative to using gets is fgets

```
char *fgets(char *restrict s, int n, FILE *restrict stream) ;
```

The *fgets()* function shall read bytes from *stream* into the array pointed to by *s*, until *n*-1 bytes are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null byte.

- Resulting program:

```
#include <stdio.h>

main() {
    char buffer[25];
    printf("\nEnter Text : ");
    fgets(buffer, 25, stdin);
}
```

## Dangerous Functions (6)

- But `fgets` is far from foolproof and requires that the programmer supplies the **correct value for the number of bytes to be read**

- Therefore, this program...

```
#include <stdio.h>

main() {
    char buffer[25];
    printf("\nEnter Text : ");
    fgets(buffer, 26, stdin);
}
```

- ...still results in a buffer overflow

```
rennhard@ubuntu-generic:~$ ./fgets
Enter Text : 1234567890123456789012345
*** stack smashing detected ***: ./fgets terminated
```

## Dangerous Functions (7)

- There exist **safer versions for many string functions** in C, for instance `strncpy` instead of `strcpy`

```
char *strcpy(char *restrict s1, const char *restrict s2);
```

The `strcpy()` function shall copy the string pointed to by `s2` (including the terminating null byte) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

```
char *strncpy(char *restrict s1, const char *restrict s2,  
size_t n);
```

The `strncpy()` function shall copy **not more than `n` bytes** (bytes that follow a null byte are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

- But in all cases, **correctly specifying `n` is mandatory**, otherwise buffer overflows will likely result

## Race Conditions

# Race Conditions

- Race conditions are possible in scenarios **where multiple threads or processes are run simultaneously** that may potentially interact
  - In such scenarios, race condition bugs are very common
- Race conditions are a **very special type** of bug
  - Race conditions can easily be overlooked during testing, because they often don't appear in highly controlled testing environments
  - A specific race condition in a program may occur only rarely, which makes them hard to reproduce and debug
  - Fixing them is not always easy
- Most of the time, race conditions present robustness problems, but sometimes, race conditions **also have security implications**
- Part of the "Time and State" kingdom

# What is a Race Condition – the “Two Elevators Problem”

- Alice and Bob, working at the same company, agree to **meet for lunch in the lobby** at noon
  - But they don't agree whether they mean the lobby at their office level or the main lobby
- At 12:15, Alice is waiting for Bob in the office lobby
  - She realizes Bob may be in the main lobby and **takes one elevator down**
- If Bob's there – good; but if he isn't, Alice can't conclude Bob is late
  - Just like to Alice, it could have occurred to Bob that Alice is waiting in the other lobby, so he may have taken the other elevator to go up
- This is a **race condition**: The result will only be 100% clear for Alice if we **assume** Bob does not take the other elevator at the same time



A race conditions occurs when for the correct outcome or behavior

- An assumption must be valid for a period of time...
- ...but it is not guaranteed the assumption is valid in any case

## Example: Java Servlet (1)

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;

public class Counter extends HttpServlet {
    private int count = 0;

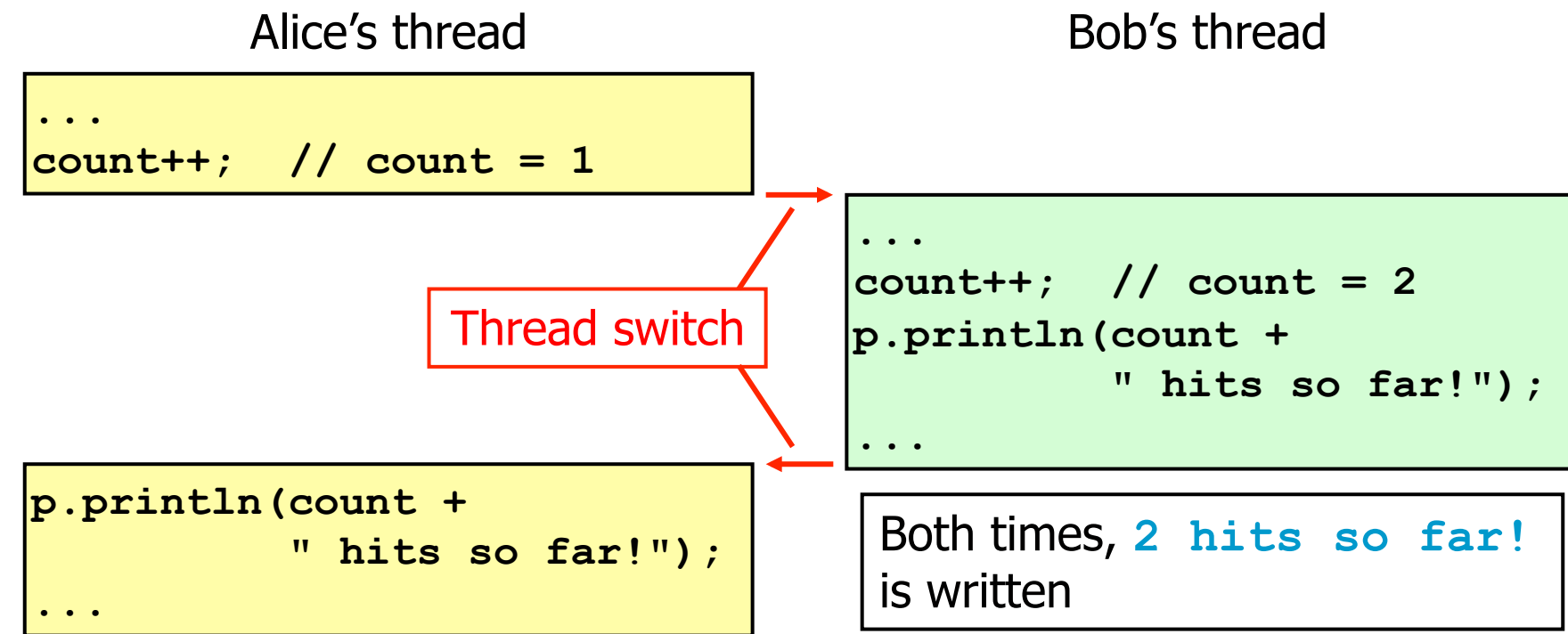
    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
                      throws ServletException, IOException {
        out.setContentType("text/plain");
        PrintWriter p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```

- Question: can you spot a race condition here?



## Example: Java Servlet (2)

- The example contains a race condition, because **Java servlets are multithreaded**
  - Multiple threads that handle requests from different users use the same servlet instance, which may result in a wrong value of `count` being written
- Assume Alice and Bob use the servlet **at nearly the same time**
  - We assume `count = 0` at the beginning



## Example: Java Servlet (3)

- The likelihood that the wrong behavior occurs is small
  - It happens only, if a thread switch happens between incrementing `count` and the call of `println`

```
count++;  
p.println(count + " hits so far!");
```

- Nevertheless, this is a bug and it should be fixed
  - Even this is no security-critical bug – but similar security-critical bugs can easily be imagined
    - Imagine that instead of updating a count-variable a security token for session tracking is generated and sent to the user
    - If a switch happens between generating a token and sending it to the user, a user could receive the token of another user

## Example: Java Servlet (4)

- The easiest “fix” would be to avoid the problem at all, i.e. **not using an instance variable**
  - Because only instance variables are shared among threads
  - But here, this is not an option as we count the number of calls of the doGet-Method, so we must use an instance variable
- To fix of a race condition we must make sure that the made **assumptions for correct behavior are indeed valid**
  - Servlet example: make sure **no thread switch** can take place between updating the count variable and reading its new value
- First naïve attempt: write the two lines **in one line**

```
p.println(++count + " hits so far!");
```

- This most likely **won't change anything**
  - In Java bytecode, this **still results in several instructions** and the thread switch may take place between any two instructions

## Example: Java Servlet (5)

- To truly solve the problem, we must make sure the relevant code section is **atomic**
  - Atomic means the code section executes as if the operations were a single unit in the sense that “no other thread can get between” during program execution
- To make multiple operations atomic, **locking mechanisms** are usually used
  - In Java, this can be achieved by using the per-object lock with the **synchronized** keyword
  - The synchronized keyword guarantees that at any time and with multiple threads accessing an object, at most one thread can be in any code section that is protected by the synchronized keyword

## Example: Java Servlet (6)

- Second attempt: **protect the method** with synchronized

```
public class Counter extends HttpServlet {  
    private int count = 0;  
  
    public synchronized void doGet(HttpServletRequest in,  
                                   HttpServletResponse out)  
                                   throws ServletException, IOException {  
        out.setContentType("text/plain");  
        PrintWriter p = out.getWriter();  
        count++;  
        p.println(count + " hits so far!");  
    }  
}
```

- This works, but is **not as efficient as it could be**
  - In particular, including the writing to the network into the critical section means that any blocked writing operation blocks the application

## Example: Java Servlet (7)

- Third attempt: keep the **synchronized section as small as possible**

```
public class Counter extends HttpServlet {
    private int count = 0;

    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
                      throws ServletException, IOException {

        int myCount;
        out.setContentType("text/plain");
        PrintWriter p = out.getWriter();
        synchronized(this) {
            mycount = ++count;
        }
        p.println(my_count + " hits so far!");
    }
}
```

# TOCTOU – Time-of-Check, Time-of-Use

- Race conditions not only occur in multithreaded programs, but in general **when different processes on a system share resources**
- This is where **file-based race conditions** come into play, which are most prominent among **security-critical** race conditions
- Most security-critical file-based race conditions follow this pattern:
  - There's a **check of a file property** that precedes the use of the file
    - E.g. it is checked whether a user is allowed to read a file
  - When **using the file**, it is assumed that the previous check was in indeed done on the current file → this is the assumption that must be valid
    - It is assumed that if the result of the previous check was positive, the user is indeed allowed to read the file that is used
  - If an attacker manages to **invalidate this assumption**, he can access files he shouldn't be allowed to access
    - Such defects are called **time-of-check, time-of-use (TOCTOU)** bugs

## Example: Checking File Access Rights (1)

- The following code is part of a program that allows **writing to files**
  - The program runs **setuid root**, so the program has access to any file (so the effective user ID (EUID) of the process is root)
  - Therefore, the program checks if the user running the program (real UID, or simply UID) has **access to a particular file** before granting access

```
if(!access(file, W_OK)) { /* checks if the user has write
                           access to file (pathname),
                           returns 0 on success */
    f = fopen(file, "w+"); /* open the file for writing */
    write_to_file(f);
} else {
    fprintf(stderr, "Permission denied when trying to \
                   open %s.\n", file);
}
```

- This sounds reasonable... but unfortunately, the program contains a **major security vulnerability** – can you spot it?



## Example: Checking File Access Rights (2)

- An attacker that has **local access to the system** (legitimate or not) can exploit the vulnerability to get write access to any file
  - The idea is that the attacker **switches the files between checking and opening** it
  - This can best be done using **symbolic links**
- The **attacker's strategy** is as follows
  - In his home directory, he creates a file and sets a symbolic link to it

```
$ touch dummy  
$ ln -s dummy pointer
```

- Now the attacker starts the program to write to **pointer**
- Just between the check of the access rights and the opening of the file, the attacker **changes the link**, which gives him access to a file he shouldn't be able to access based on his rights

```
$ rm pointer;  
$ ln -s /etc/shadow pointer
```

## Example: Checking File Access Rights (3)

- Of course, executing these **commands manually** (redirecting the link) most likely won't be successful
- One therefore **automates such attacks** by writing a small program that
  - Sets the link (`pointer`) to a file the attacker is allowed to access (dummy)
  - Starts the program to write the file (`pointer`)
  - Redirects the link to the desired file (`/etc/shadow`)
  - Checks the desired file to see whether it was overwritten, e.g. by inspecting the modification date
  - Repeating the attack several times with minor timing variations will increase the probability that it will eventually be successful

# Avoiding TOCTOU Problems

- Minimize the number of function calls that take a **filename** for an input
  - Use the filename only once for initial file access, which usually returns a **file handle or a file descriptor** for further access
    - This guarantees that as long the file handle is used, it is not possible for an attacker to exchange the underlying file by redirecting a symbolic link
  - In this example, first open the file and then use the returned file descriptor to check the access rights
- **If possible, avoid doing your own access checking on files**, leave that to the underlying operating system
  - E.g. by not running the program setuid root but with the rights of the user
    - But this is not always possible, e.g. server applications that are accessed over the network usually implement their own user management and access control
- In general, avoid running a program with **high privileges** (e.g. setuid root) unless absolutely necessary
  - If high privileges are needed think “especially hard” about security

# Final Exercise

- The following Java program allows **listing the contents of a directory specified as command line parameter**, by invoking a shell command using the java Runtime class
  - Imagine the code is part of a server application that allows listing the directory contents
  - Example: `java ListDirectoryContents .` lists the contents of the current directory
  - The program allows listing any directory, but should do nothing else
- Can you spot the (major) security-relevant bug? To which kingdom does it belong? How could it basically be fixed?

```
public class ListDirectoryContents {
    public static void main(String[] args) throws IOException {
        if(args.length != 1) {
            System.out.println("Specify the directory to list");
            System.exit(1);
        }
        Runtime runtime = Runtime.getRuntime();
        String[] cmd = new String[3];
        cmd[0] = "/bin/sh"; cmd[1] = "-c"; cmd[2] = "ls " + args[0];
        Process proc = runtime.exec(cmd);

        InputStream is = proc.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}
```

The executed command is  
`/bin/sh -c ls args[0]`  
 e.g.  
`/bin/sh -c ls .`  
 and the output is displayed on stdout

# Final Exercise – Solution

# How to Start Reducing Coding Errors?

- There's a **vast range of possible security-relevant mistakes** one can do during the implementation of software
  - It's certainly overwhelming, especially for people that are just "entering" the field of secure software
- So what can you do to **start reducing your number of coding errors**?
  - Use a static code analysis tool (we will look at this later) → it will likely detect several of your bugs
  - During your projects, get "more security aware" and consult secure software guidelines for your language (C, Java, PHP, C#...) → there are many resources available (online & books)
    - E.g. the online documentation of the here used taxonomy of coding errors

# Summary

- There's a **wide spectrum** of different types of security-relevant coding errors
- One way to classify them is according to Gary McGraw's **Taxonomy of Coding Errors**
  - Uses 7 (+1) main classes of errors ("Kingdoms")
- **Buffer overflows** are a prominent type of bug and belong to the "Input Validation and Representation" kingdom
  - They happen if data is written beyond the end of an allocated buffer
- **Using dangerous functions** is a typical error of the "API Abuse" kingdom
  - In C, there exist safer versions for many string functions, such as `strncpy` instead of `strcpy` or `fgets` instead of `gets`
- A **race condition** is an error related to the "Time and State" kingdom
  - They typically happen in multithreaded programs or if multiple programs access shared resources (e.g. a file)