Software Security Errors

Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

Content

- An overview of typical security-relevant software errors based on the taxonomy of the 7 (+1) Kingdoms

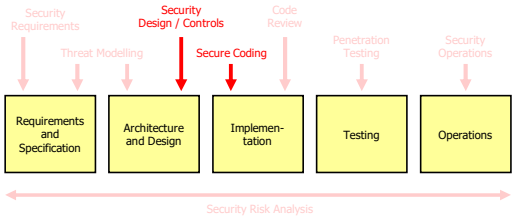
way of categorizing things

- A more detailed look at some specific security-relevant software errors
  - Buffer overflows
  - Race conditions

- Note: This chapter serves mainly as a first overview of software security errors – much more details will follow in the next chapters

Goals

- You know and understand the 7 (+1) kingdoms of software security errors 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 understand what a race condition is, why they can happen, why they can be security-critical, and know some countermeasures to prevent them
- Security activities covered in this chapter:



Find the Security Error Exercise 1

- Method *list* returns the contents of a directory specified with parameter *directory*
  - Imagine the code is part of a server application that allows listing the directory contents and that gets the directory over the network from the client
  - Example: `list("/etc")` returns the contents of the directory `/etc`
  - The program allows listing any directory, but should do nothing else
- Can you spot the (major) security-relevant error? How could it basically be fixed?

```
public class ListDirectoryContents {
  public String list(String directory) throws IOException {
    Runtime runtime = Runtime.getRuntime();
    String[] cmd = new String[3];
    cmd[0] = "/bin/sh";
    cmd[1] = "-c";
    cmd[2] = "ls " + directory;
    Process proc = runtime.exec(cmd);

    Scanner reader =
      new Scanner(proc.getInputStream());
    StringBuilder sb = new StringBuilder();
    while (reader.hasNextLine()) {
      sb.append(reader.nextLine()).append("\n");
    }
    return sb.toString();
  }
}
```

The directory content is accessed by invoking a shell command in the OS using the Java *Runtime* class

Executed command is `/bin/sh -c ls directory`
e.g., `/bin/sh -c ls /etc`

Read the directory content from the *Process* object and return it as a string

Find the Security Error Exercise 1 – Solution

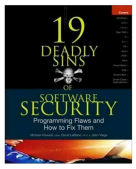
- Problem: The value received for the parameter *directory* is not validated at all and it is directly used in the executed command
  - `/bin/sh -c ls directory`
- An attacker can exploit this in a command injection attack to execute arbitrary commands (restricted by the rights of the running process)
  - To do this, he has to construct the parameter *directory* such that additional commands are executed
  - With `/bin/sh`, multiple commands can be specified with ;
- E.g., to show the contents of */etc/passwd*:
  - Use `"/etc; less /etc/passwd"` for parameter *directory*
  - `/bin/sh -c ls /etc; less /etc/passwd`
- E.g., to delete the contents in the current directory:
  - Use `"/etc; rm -rf *"` for parameter *directory*
  - `/bin/sh -c ls /etc; rm -rf *`
- Fix: Perform proper input validation

Fix: run /bin/ls instead of the full shell /bin/sh

Software Security Errors

- There's a wide spectrum of different types of software security errors, but they are usually based on a few fundamental problems
  - → Using some kind of classification scheme is reasonable

- There exist several classification schemes for security-related software errors
  - OWASP Top Ten, SANS Top-25,
    The 19 Deadly Sins of Software Security

- We are using here Gary McGraw's Taxonomy of Software Security Errors
  - Very general classification scheme, not just for specific application domains (such as OWASP)
  - It is continuously maintained and extended
  - Uses a reasonable number of main error classes: 7 (+1) Kingdom

7 (+1) Kingdoms of Software Security Errors

7 (+1) Kingdoms of Software Security Errors

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

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

* Environment

- The environment includes all of the stuff that is outside of your own code but still critical to the security of the software you create

Namen müssen nicht auswendig gelernt werden, werden in Spick mitgegeben

7 (+1) Kingdoms of Software Security Errors

- The taxonomy is also available online
- *https://vulncat.fortify.com*
- It's organized according to the kingdoms and categories within the kingdoms
- It contains examples in different programming languages to illustrate the problems and solutions
- It is continuously maintained and extended with
  - new types of software security errors as new attack variants are uncovered
  - new programming languages when they become popular

Kingdom 1: Input Validation and Representation (1)

- Primary problem: Data received by an application or a system is not or not correctly checked before it is processed
  - Data can be provided by users but also by other systems
- The solution is basically quite simple: Perform input validation with all data received before it is processed
  - This means to specify and enforce corresponding rules in the software
  - Example: During registration in a web application, the chosen username must contain between 8 and 16 characters consisting of letters and digits – otherwise the registration request is not processed
- What makes input validation (sometimes) difficult is that the same data can be encoded (represented) in different ways
  - It therefore may be possible for an attacker to circumvent input validation 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 an attacker to, e.g., execute system commands or 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 an attacker to access arbitrary files on the target computer
  - Typical example: Trying to access a web server using the URL *http://www.host.com/../../../../../../etc/shadow*

Kingdom 2: API Abuse (1)

- 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 the return value of a function means that unexpected situations will be overlooked during runtime
  - E.g., a reference containing null instead of referencing a valid object because something went wrong
    - When accessing this object, the program likely crashes → availability problems
- Wrong security assumptions
  - If one makes wrong assumption about the security of a function or method, this may have severe security implications

Kingdom 2: API Abuse (2)

Wrong security assumptions example:

- A server application is developed which should only be usable by selected clients
- To solve this, the developer integrates a list of client host names (e.g., *alice.zhaw.ch*) that are allowed to access the server
- When a client connects, the server knows its IP address and performs an inverse DNS lookup to get the host name of the client
  - E.g., using *getHostName* in Java or *gethostbyaddr* in C
- If the received host name is on the list of allowed host names, the client is granted access
- What is the security problem here?
  - DNS is (usually) not secure; it's relatively easy to spoof DNS responses assuming the attacker is on the communication channel
  - So the developer has made a wrong assumption about the security provided by the inverse DNS lookup functions

Kingdom 3: Security Features

- This kingdom deals with wrong usage of security functions
  - E.g., cryptography, secure communication, authentication, access control,...
  - To prevent problems, you shouldn't invent your own security functions (unless this is necessary), but use what has proven to work well in practice
    - Especially true with cryptography – you'll most likely fail
  - But even if established algorithms / protocols / libraries are used, experience shows that it's very easy to make mistakes when configuring and using them

Some examples of poor usage of security features:

- Insecure randomness
  - Using insecure pseudo random number generators (or seeding secure ones with predictable values) will result, e.g., 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 / RC4 / 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 or with multithreading or multiple processes on 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 (clients) 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)
- File access race condition: TOCTOU (time of check – time of use)
  - The time window between checking a file property and using the file may be exploited by an attacker to increase his file access privileges
- Re-using session IDs after authentication in web applications
  - Using the same session ID across authentication boundaries may allow an attacker to hijack authenticated sessions

Deadlocks in Real Life

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.»

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:

- Leakage of internal information
  - Error messages presented to the users sometimes contain detailed internal information (system state, failed database queries,...), which may be helpful to an attacker to carry out subsequent attacks
- Empty catch block
  - Ignoring exceptions may result in unexpected program behavior (similar to ignoring return values → program may crash / availability problems)

Overly Broad Catch Block

Very «broad» catch blocks are also often seen in programs; in Java, this would mean, e.g., catching the base exception class *Exception*. The problem with this is that if the corresponding code block is extended at a later time, then exceptions that are now newly thrown (by the added code) are already «handled» by the existing catch block. As a result of this, the newly thrown exceptions might not get the attention they should get (e.g., it could be they require a very different treatment than the other exceptions handled by the same catch block).

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 analyze and maintain)
    - Poor names for variables, methods etc.
    - Too long and too complex classes or methods
    - Poor class design (high coupling among classes, violating information hiding,...)
    - Forgetting to remove old code
    - ...
  - Because the developer is not careful enough during programming and does not think about various details, e.g.:
    - Allocating resources but not thinking about whether they are released again
    - Using objects / variables and not thinking about whether they have been correctly 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 (→ availability problems)
  - 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
- 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 defects!
  - 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

Deprecation for Security Reasons

An example are some methods of the Java Thread class, which were deprecated because using them makes it virtually impossible to implement multithreaded programs in a secure way. For details, see *https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html*.

Another example is the C function *gets()*, which is also considered to be deprecated due to security reasons. A better alternative is the *getline()* or *fgets()* functions.

Some further Code Quality Issues

Null dereference

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

Uninitialized variable

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

Kingdom 7: Encapsulation

- Encapsulation 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 web 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 (such as the authorization of the user) – 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)

Hidden Form Fields

Imagine a web application that uses a hidden field to remember the authorization level of the user (e.g., guest, registered user, administrator). Whenever a request is received, the web application checks if the authorization level is high enough to perform the requested operation. This can easily be exploited by the attacker to elevate his privileges by making sure the value is set to administrator in every request he makes (which can easily be automated using the appropriate tools). In general, session state information should only be maintained on the server, which effectively prevents such attacks.

Kingdom *: Environment

- The software you write does not run by itself, but usually relies on other software
  - Compilers, operating systems, execution environments (JVM, .NET,...)
  - Frameworks, libraries, other software on other computers (in a networked environment), network services (e.g., 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:

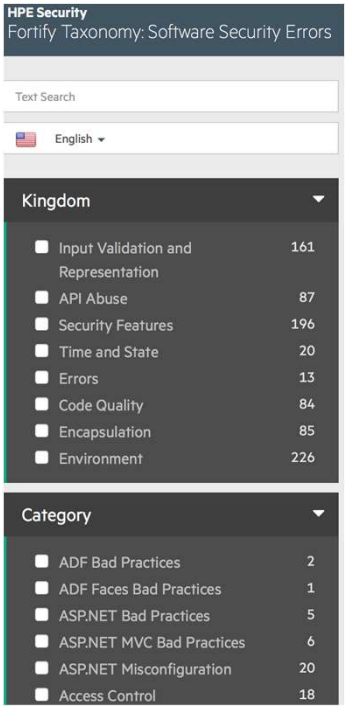
- Insecure compiler optimization
  - E.g., the developer overwrites sensitive data in memory, but the compiler removes this write operation to optimize the code
- Issues with respect to web application frameworks
  - E.g., insufficient session ID length or randomness (may allow session ID guessing attacks)

Insecure Compiler Optimization

For more details, refer to
https://vulncat.fortify.com/en/detail?id=desc.semantic.cpp.insecure_compiler_optimization

Further Information about Software Security Errors

- The taxonomy presented here is available online
  - *https://vulncat.fortify.com*
  - It's organized according to the kingdoms and categories within the kingdoms
  - It contains examples in different programming languages to illustrate the problems and solutions
  - It is continuously maintained and extended with
    - new types of software security errors as new attack variants are uncovered
    - new programming languages when they become popular



Find the Security Error Exercise 2

- An application uses the method *checkPassword* to check the correctness of a submitted password
  - The poor password and the fact that it is stored in plaintext within the code is not considered a vulnerability here

```
public class PasswordCheck {
  private String correctPassword = "zhaw";

  public boolean checkPassword(String submittedPassword) {
    if (submittedPassword.length() != correctPassword.length())
      return false;
    for (int i = 0; i < correctPassword.length(); i++) {
      if (submittedPassword.charAt(i) != correctPassword.charAt(i)) {
        return false;
      }
    }
    return true;
  }
}
```

- This contains a security-critical error – can you spot it? To which kingdom does it belong? How could it basically be fixed?

Find the Security Error Exercise 2 – Solution

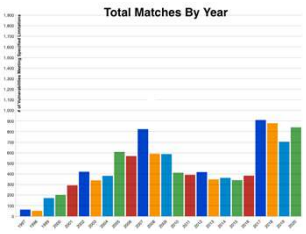
Problem: The duration of the password check depends on the number of correct characters in the submitted password

- If the first *n* characters are correct, then *n+1* characters are tested
- This allows to incrementally find the correct password, left to right, which is much faster than a brute force attack
  - `a--- , b--- , c--- , .... , x--- , y--- , z---`
  - `za-- , zb-- , .... , zh-- , .... , zy-- , zz--`
  - `zha- , zhb- , zhc- , .... , zahx , zahy , zahz`
  - `zaha , zahb , zahc , .... , zahw , .... , zahz`
- Exploiting this in reality may be difficult (esp. over the Internet), but exploiting it locally or within a LAN is often possible
- Kingdom: Time and State (timing issue)
- Solve it by always checking all characters of the submitted passwords, which should eliminate substantial timing differences

Buffer Overflows

Buffer Overflows

- A buffer overflow happens when a program writes or reads data beyond the end of an allocated buffer in memory
  - If a buffer overflow can be exploited, an attacker may be able, e.g., to modify the program flow, crash the program, inject (malicious) own code, access sensitive information, etc.
  - Part of the Input Validation and Representation kingdom
- Despite advances with respect to countermeasures, buffer overflow attacks are still frequently used (about 1'000 new CVE entries per year) and many prominent malware incidents (past & present) exploited buffer overflow vulnerabilities:
  - 1988: Morris worm, 2001: Code Red, 2003: SQLSlammer, 2010: Stuxnet, 2012: Flame, 2014: Heartbleed, 2017: WannaCry



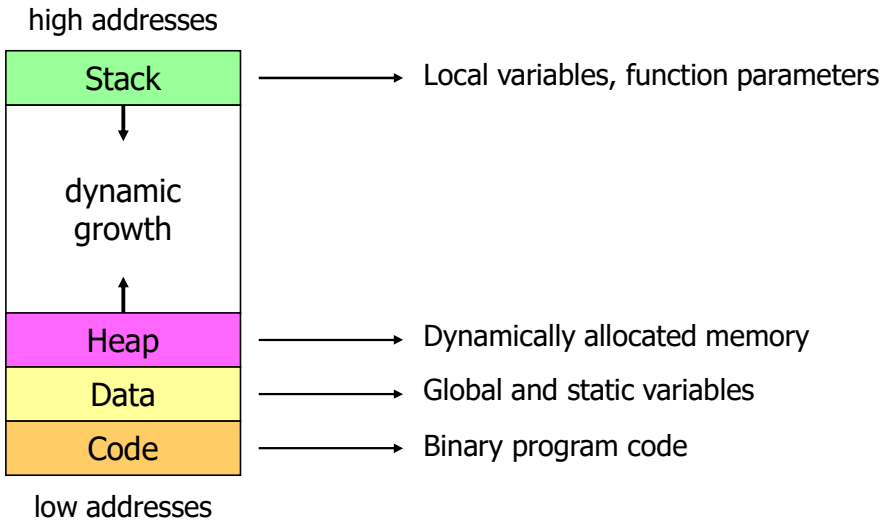
- To understand buffer overflow attacks, one must understand the memory layout of a process and how the stack is used
  - Note that buffer overflows can also happen on the heap, but in most cases, the stack is involved

The National Vulnerability Database

The National Vulnerability Database is sponsored by the US-CERT (Computer Emergency Response Team) and contains reported software vulnerabilities. Each vulnerability gets its CVE number and the database can be searched at *https://nvd.nist.gov/vuln/search*. The statistics above has been created as follows: Use *Advanced* for *Search Type*, select *Statistics* for *Result Type*, use *buffer overflow* for *Keyword Search*, and in the *Published Date Range* fields, use *01/01/1997* and the desired end date.

Memory Layout of a Process

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



Memory Layout of a Process

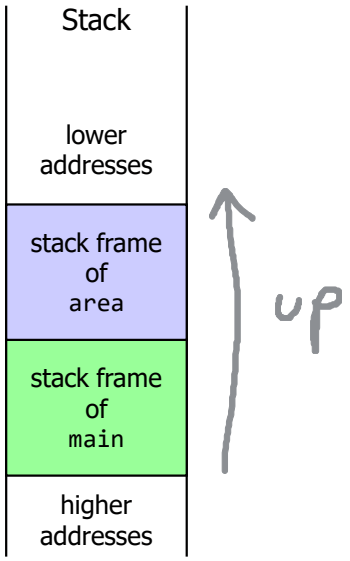
- The **Code** segment (often also called Text segment) contains the executable program (in machine language). This is where the instruction pointers should normally point to.
- The **Data** segment contains all data, that exist exactly once during program executions (static and global variables).
- The **Stack** segment contains data, that are produced and destroyed dynamically during run-time in a "well-defined order". This includes function arguments and local variables.
- The **Heap** segment is used for all other, dynamically during run-time generated data (*malloc* in C, *new* in Java)

The Stack

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

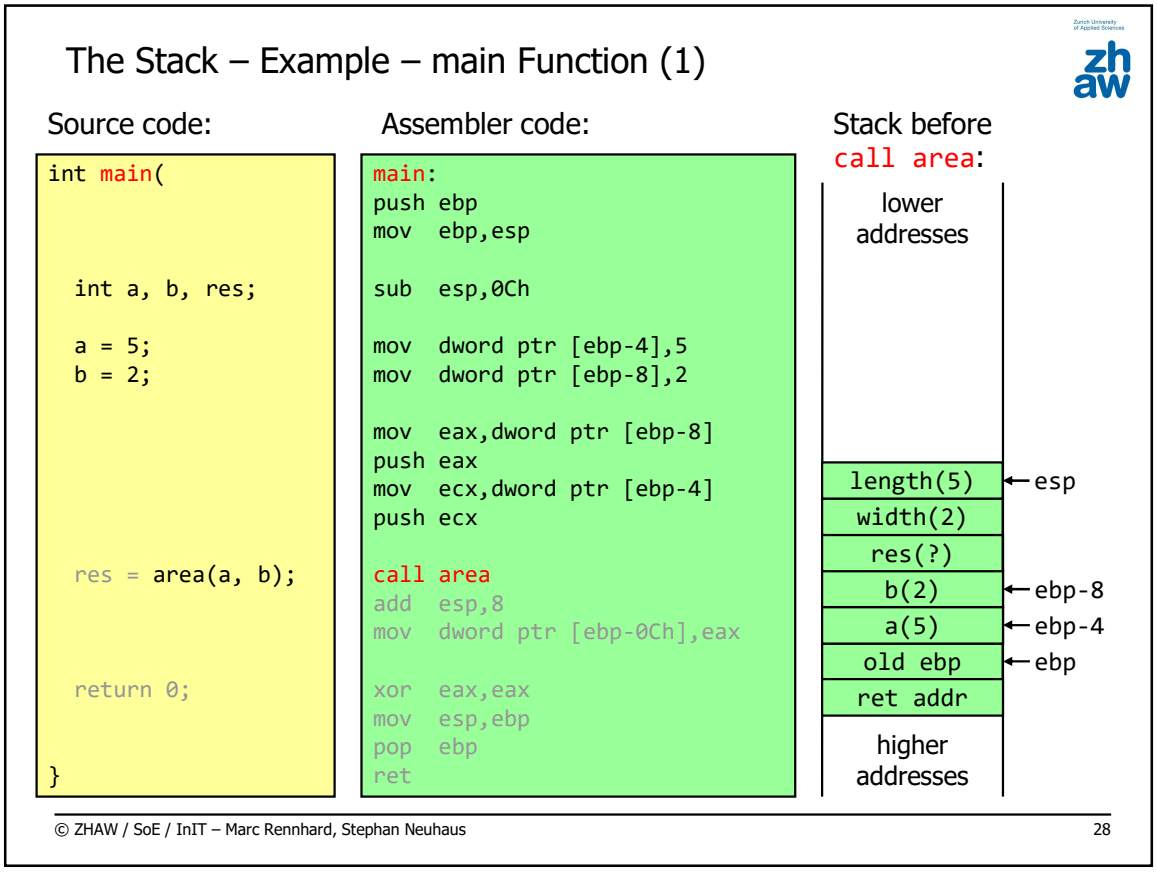
```
int area(int length, int width) {
  int scale;

  scale = 3;
  return (scale * length * width);
}

int main() {
  int a, b, res;

  a = 5;
  b = 2;
  res = area(a, b);
  return 0;
}
```

no bufferoverflow, just example of stack usage



The Stack and Stack Frames

During program execution, a stack grows and shrinks. The stack grows from higher to lower addresses, which means from bottom to top in the illustration above. When the program is started and `main` is called, a **stack frame** for `main` is created. A stack frame belongs to a function and is used when the function is executed. When `main` calls another function `area`, a new stack frame is created and used by `area`. When execution of `area` completes, the stack frame of `area` is removed from the stack and program execution continues in the `main` function (which called area) using again its own stack frame. At any time, only the topmost stack frame is “in usage” by the function that is currently executed.



Assembly Language

The example above uses x86 assembly language («Intel syntax»). It was created using *gcc* on a 32-bit Linux system.

Base and Stack Pointers

There are two important pointers (CPU registers) associated with the stack. The **stack pointer** (esp) point always to the topmost entry on the stack. It is used to indicate where data shall be pushed onto the stack or popped from the top of the stack. The **base pointer** (ebp) always points to the bottom of the currently used stack frame (not exactly the bottom, but one position above the return address, see below). It remains constant even if the stack frame grows and shrinks and is therefore used to relatively address all other positions in the current stack frame.

main Function until `call area`

In a C program, the main function is the entry point. The executable file that was built during compiling and linking contains additional code (this is not shown here), which basically passes command-line arguments to the main function and calls the main function by executing a `call main`. This is where we start looking at the assembler code of the example program at what happens with the stack in more detail.

Whenever a function is called, the current value of the **instruction pointer** (eip) is put automatically onto the stack by the CPU. This value corresponds to the address to return to after execution of the called function (therefore we call it `ret addr` in the illustration of the stack above). So when the main function that is called by the code that was automatically added to the executable file terminates (using the `ret` instruction, see later), program execution returns to the address that directly follows `call main` in this automatically added code.

The following happens in the main function until `call area` is executed:

- **push ebp** The base pointer of the previous stack frame (the automatically added code that calls the main function) is pushed onto the stack, so it can be restored later.
  - **mov ebp,esp** The value of the stack pointer is copied to ebp, so ebp points to the bottom of the new stack frame.
- **sub esp,0Ch** esp is moved 12 bytes (C in hex) so there's room for the three int-values a, b and res.
- **mov dword ptr [ebp-4],5** The value 5 (of a) is inserted at the position ebp-4. Note that we assume here that the variables are put onto the stack in the order they are listed in the code, which is the case with many compilers, but which does not necessarily have to be this way. In fact, one can imagine any order; it's up to the compiler to arrange them – and to keep track which variable is located where.
- **mov dword ptr [ebp-8],2** The value 2 (of b) is inserted at the position ebp-8.

The following four instructions are used to pass the parameters (a and b) to the function `area`. In C, parameters are passed via the stack in opposite order of the parameter list (first b, then a).

- **mov eax,dword ptr[ebp-8]** The content of ebp-8 (b) is copied into register eax.
- **push eax** The value in eax (b) is pushed onto the stack (second function parameter).
- **mov ecx,dword ptr[ebp-4]** The content of ebp-4 (a) is copied into register ecx.
- **push ecx** The value in ecx (a) is pushed onto the stack (first function parameter).

The next instruction, `call area`, instructs the program to jump to the code of the area function. When returning from the function, program execution continues at the instruction following `call area`.

The Stack – Example – area Function

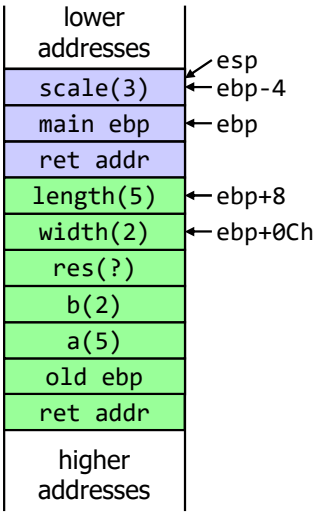
Source code:

```
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

area Function

Like with the main function, the current value of the instruction pointer (eip) is put automatically on the stack by the CPU when `call area` is executed. Here, eip corresponds to the address of the instruction add esp,8 in the main function, which is the instruction directly following the `call` instruction and which is the first instruction that will be executed when returning to the main function. The following happens during execution of the area function:

- **push ebp** The base pointer of the previous stack frame (the one of the main function) is pushed onto the stack, so it can be regenerated later.
- **mov ebp,esp** The value of the stack pointer is copied to ebp, so ebp points right above the stored return address in the new stack frame.
- **sub esp,4h** esp is moved 4 bytes so there's room for the int-variable scales.
- **mov dword ptr[ebp-4],3** The value 3 (of scale) is inserted at the position ebp-4.

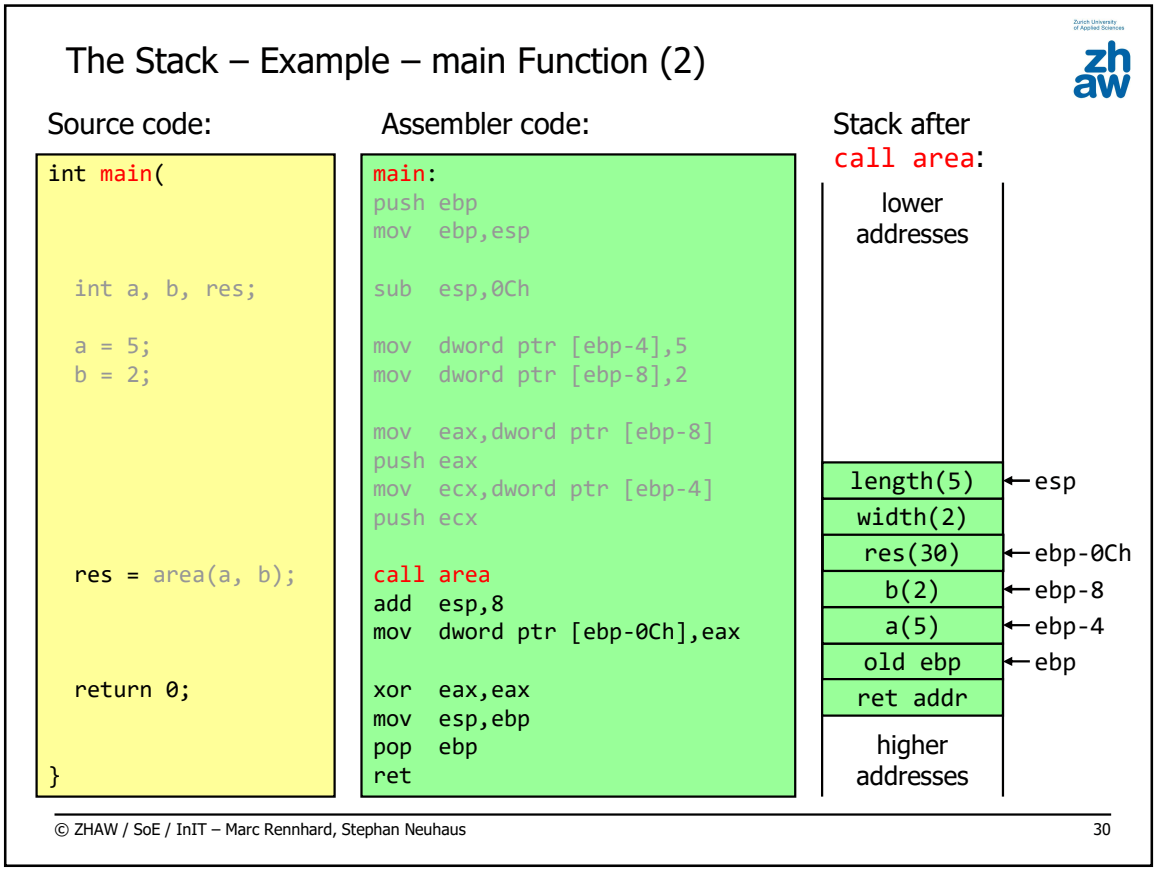
The following three instructions build the product scale * length * width. Register eax is used to carry out the computation, because eax is used to return a value from a function.

- **mov eax,dword ptr[ebp-4]** The content of ebp-4 (scale) is copied into register eax.
- **imul eax,dword ptr[ebp+8]** The content of ebp+8 (length) is multiplied with the value in eax; the result is put into eax.
- **imul eax,dword ptr[ebp+0Ch]** The content of ebp+12 (width) is multiplied with the value in eax; the result is put into eax.

Finally, the stack frame is "removed" by moving esp to the appropriate position and by copying the main ebp value into ebp, so program execution can resume after the call instruction that was used to jump to the area function.

- **mov esp,ebp** The value of the base pointer is copied to esp, so esp points right above the return address in the current stack frame. Everything above the new value of esp has therefore been "removed" from the stack.
- **pop ebp** Copy the topmost value of the stack (main ebp) into ebp and move esp one position down.
- **ret** Copy the topmost value of the stack (ret addr) into the instruction pointer eip and move esp one position down.

Now, the stack looks exactly as it has before executing call area in the main function. The return result is stored in register eax. Program execution continues at the instruction add esp,8 in the main function.



area Function after `call area`

When returning from the area function, program execution continues at the instruction add esp,8:

- **add esp,8** This increments the stack pointer by 8, i.e. moves it two positions down. This is done to remove the function parameters, which were put onto the stack by the main function before calling the area function.
- **mov dword ptr [ebp-0Ch],eax** The value eax (the return value of the area function) is copied to the position ebp-12 (res).

The following four instructions are used to return the value 0 (which in C means “normal program termination”) to the automatically added code to the main function. Again, the return value is passed using register eax. The last three instructions are the same as used in the area function; in fact, these three instructions are always used to return from a function.

- **xor eax,eax** This is an efficient way to write the value 0 into eax.
- **mov esp,ebp** The value of the base pointer is copied to esp, so esp points right above the return address in the current stack frame. Everything above the new value of esp has therefore been “removed” from the stack.
- **pop ebp** Copy the topmost value of the stack (old ebp) into ebp and move esp one position down.
- **ret** Copy the topmost value of the stack (ret addr) into the instruction pointer eip and move esp one position down.

Now, all stack entries added by the main function have been removed again. Program execution continues after call main in the automatically added code, which basically terminates the program and returns the return value (0) to the environment that started the program (e.g., a shell)

Buffer Overflows – Basics (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 has an exploitable 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 contain exploitable buffer overflow vulnerabilities (this has happened)
  - Which may allow an attacker to construct a specifically crafted Java program which exploits such a vulnerability to get, e.g., access to the underlying operating system

No Buffer Overflows in Java / .NET?

As the runtime system checks whether access to, e.g., arrays cannot be made beyond the boundaries, it is not possible to write a program that itself has an exploitable buffer overflow vulnerability. However, here may be (in fact, this has happened) vulnerabilities in the underlying runtime systems (which are often written in C or C++), which may allow an attacker to write a specifically crafted Java program which exploits such a vulnerability, to get, e.g., access to the underlying operating system.

Buffer Overflows – Basics (2)

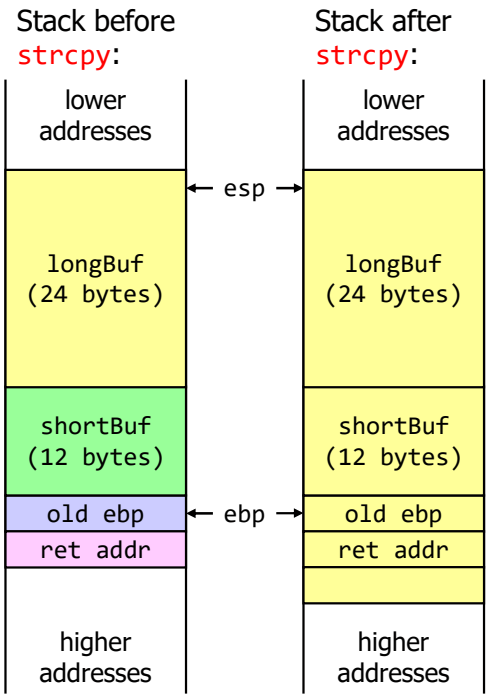
- Buffer overflow example in C:

```
void function() {
  char shortBuf[12];
  char longBuf[24] = "A looooooooooong string!";

  strcpy(shortBuf, longBuf);  /* Copies longBuf
                                 to shortBuf */
}
```

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

Effect during Program Execution

- 24 bytes from *longBuf* are written into *shortBuf*
- This overwrites `old ebp` and `ret addr` in the current stack frame (and additional 4 bytes)
- When returning from *function*, the program jumps to the address stored in `ret addr`
- This is address is different from the address that was originally stored there → unpredictable behaviour
- Probably: segmentation fault as the CPU likely tries to access a memory location 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
  - This implies that this buffer overflow is «just a programming error», but no vulnerability
  - Will probably easily be detected during testing, as it always occurs when *function* is called

- To truly exploit a write buffer overflow, it's usually required that a user can choose the data that is copied into a buffer
  - Command-line arguments
  - Data submitted to a program that is executed locally
  - Most interesting: Data submitted to a program over the network
    - This allows remote exploitation of a buffer overflow vulnerability
    - The example on the following slides discusses such a scenario

How to Exploit a Buffer Overflow (2)

- Scenario: Networked application, *processData* is part of the server program and gets and processes data from the client program

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

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

  /* Do something with the data in buffer */
}
```

  - Exercise: Study the function and try to identify and understand the vulnerability

Reading data from a Socket

A few explanations to the loop that reads data from the socket:

- **count = recv(socket, tempBuffer, 12, 0)** reads at most 12 bytes from socket and copies them into `tempBuffer`. It also writes the number of bytes read into `count`. If there are no more bytes to read, the function returns 0. The fourth parameter is set to 0 because we do not use any special flags that could also be passed to the function. Note that no buffer overflow can happen here because at most 12 bytes are read from the socket at a time.
- **memcpy(buffer + pos, tempBuffer, count)** copies the first `count` bytes from `tempBuffer` into the array `buffer`. The bytes are copied into positions `pos`, `pos+1`, …, `pos+count-1` of `buffer`.

The while-loop continues to read data from the socket and copies it into `buffer` until there's no more data from the client to read (`recv` returns 0). It doesn't matter whether the user has entered 5 or 5000 bytes, the function `processData` simply copies them into the 256-byte buffer.

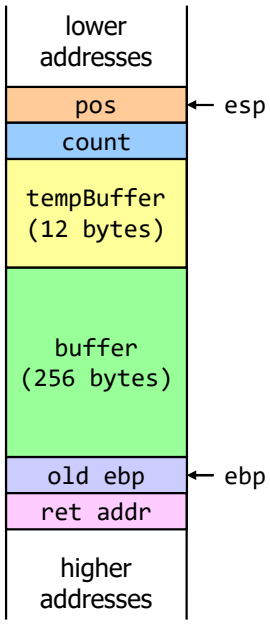
Assumption of the Programmer

One can argue the programmer is "quite stupid" to assume the server only gets at most 256 bytes at a time from the client. However, assume that the client program that is used with this server makes sure that no more than 256 bytes are submitted (maybe the programmer has developed it himself) at a time, and in this case, the programmer's assumption makes sense. The problem is, however, that attackers usually do not use the "official" client programs but directly craft packets they send to the server, which bypasses all client-side security/validation mechanisms.

How to Exploit a Buffer Overflow (3)

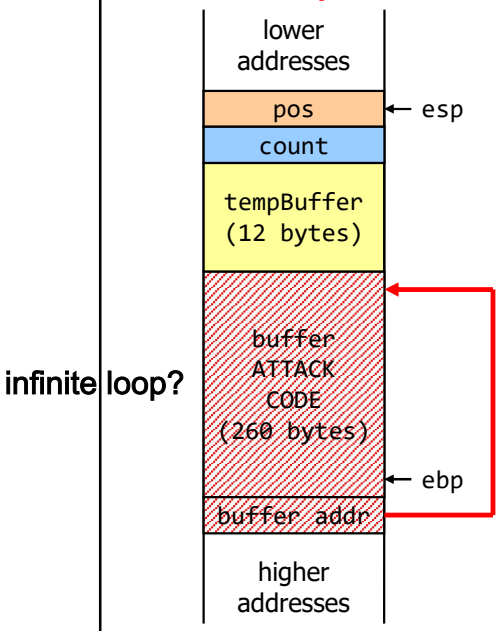
- What happens if more than 256 bytes are sent by the client:
  - the 257th - 260th bytes overwrite `old ebp`
  - the 261st - 264th bytes overwrite `ret addr`
- A user accidentally submitting more than 256 bytes of data will probably «just» crash the server program
- An attacker can do the following:
  - Submit 264 bytes data
  - The first 260 bytes contain code crafted by the attacker
  - The 261th - 264th bytes contain the address of `buffer` from where is the address?
  - When the function is left (`ret`), program execution jumps to the start of `buffer` 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
  - Because of this, you should run software always with minimal privileges to limit the damage that can be done by the attacker
- Besides code injection, the attacker can also perform other attacks by exploiting a buffer overflow vulnerability (see security lab)

Crafting a Buffer Overflow Attack

Discovering a buffer overflow vulnerability and crafting a corresponding attack is certainly not easy and typically requires access to the source code or at least to the binary executable to experiment with. In the latter case, one can also use disassembler to get a better understanding of the program code and to possibly detect vulnerabilities. In general, finding and exploiting buffer overflow attacks without having access to the source code often involves a lot of trial-and error. Buffer overflow vulnerabilities can be found by simply «throwing» various different inputs at a program and waiting until it crashes. If it crashes, it's likely a buffer overflow occurred that corrupted the stack and caused a segmentation fault. The next step is then to learn more about the stack layout, which can be done by using a debuggable version of the program or by doing core dumps during program execution to get an understanding about what is going on within the program during execution.

Sometimes, «black hats» (the bad guys) also get help from the «white hats» (the good guys, ethical hackers) who discover a buffer overflow vulnerability and publish a non-malicious proof-of-concept exploit of the attack. While this is certainly a good idea to put pressure on the software manufacturer to quickly release a patch and to increase the quality of their software in the future, it also gives the attacker a good basis to exploit the vulnerability in a malicious way. Whether publishing discovered vulnerabilities is a good idea or not has been and still is discussed extensively by security experts, although today, most believe it's indeed a good idea. In the foreword of the book "Hacking Exposed, 2nd Edition", Bruce Schneier, a well-respected security professional, argues why publishing vulnerabilities is a good idea.

Security that is based on publishing vulnerabilities is more robust. Yes, attackers learn about the vulnerabilities, but they would have learned about them anyway. More importantly, defenders can learn about them, product vendors can fix them, and sysadmins can defend against them. The more people that know about a vulnerability, the better chance it has of being fixed. By aligning yourself with the natural flow of information instead of trying to fight it, you end up with more security rather than less.

Buffer Overflows – Countermeasures (1)

- Good programming techniques
  - Make sure buffer overflows don't happen by checking that any write or read access to the buffer happens within its boundaries (input validation)
    - E.g., in the *processData* function discussed before, make sure no more than 256 bytes are written into *buffer* by checking this before every call of *memcpy*
  - Avoid unsafe C/C++ functions (e.g., *gets*, *strcpy*) and use the corresponding safer alternatives whenever possible (e.g., *fgets*, *strncpy*)

    Dozent: do not use C/C++ if possible
- Automated software tests
  - Static code analysis
  - Fuzzing: submit various inputs and analyze program behavior
- Protection mechanisms provided by compilers
  - 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
  - Some compilers insert code for «stack canaries» to detect overwriting of the return address

Endpoint Firewalls

«Normal» computer users can already do a lot to protect their systems from buffer overflow attacks by using an endpoint firewall and blocking any access from outside to local services because client systems rarely have to provide services to other computers. If needed, one can always selectively grant access to some services. Especially, on MS Windows systems, block access to TCP ports 135-139 and 445 (unless you really must provide access to them) as the corresponding services have a long history of buffer overflow vulnerabilities.

Protection Mechanisms provided by Compilers

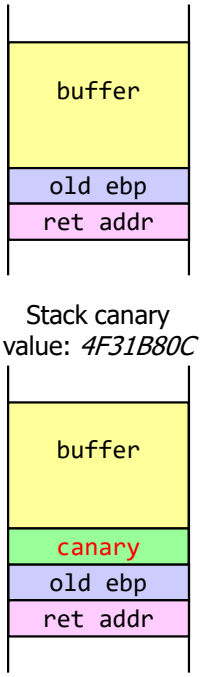
A stack canary means that the compiler inserts code into the functions of the program that puts a random value (the «canary», generated at the start of the program) onto the stack right after the return address and the old ebp whenever a function is called. In addition, the inserted code checks if the canary was not changed before returning from the function. The idea is that if a buffer overflow vulnerability is exploited to modify the return address, this most likely also results in an overwritten canary. If the code determines a canary has been changed, the program terminates immediately.

The *gcc* compiler offers several options that help preventing buffer overflow attacks. The option *-fstack-protector* advises the compiler to insert stack-canaries. The Option *-D_FORTIFY_SOURCE=2* tells the compiler to insert boundary checks (if possible) around potentially critical operations such as buffer copying.

An interesting article about stack canaries and other protection measures can be found here: *http://www.phrack.org/issues.html?issue=67&id=13*.

Stack Canaries

- Problem: If `buffer` is overwritten, `ret addr` can be overwritten, which allows, e.g., code injection
- The purpose of stack canaries is to detect overwriting of `ret addr`:
  - When starting the program, a random value is generated → stack canary value (e.g., 4 bytes)
  - When entering a function, this value is pushed onto the stack (just after saving `old ebp`) → `canary`
  - Before leaving the function, it is checked if the value of `canary` has not changed (still contains *4F31B80C*)
  - If the value has changed, the program is terminated
- Why does this work:
  - If the attacker overflows `buffer` to modify `ret addr`, the `canary` is also affected (changed)
    - Unless the attacker knows the correct value or is very lucky
  - Therefore, such an attack (virtually) always results in a changed `canary` – and as the program is immediately terminated in this case, any further damage will be prevented



Buffer Overflows – Countermeasures (2)

- Prevent code execution in data segments (e.g., stack)
  - By setting a specific bit in the paging table
  - E.g., part of Linux kernel since 2.6.7
- Address Space Layout Randomization (ASLR)
  - Randomizes the address layout of a process when it is loaded, e.g., part of Linux kernel since 2004
  - Causes the segments (code, heap, stack,…) to be placed at non-predictable locations in the virtual address 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
    - Corrupt memory organization on the heap
- So with all these safeguards, is the buffer overflow problem solved?

ASLR

Note that ASLR can only be used (at least on Linux) if the program is compiled as a Position Independent Executable (PIE), see, e.g., *http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/*.

Buffer Overflows – Countermeasures (3)

NOPE – buffer overflow problems are not solved with this:

- There's no guarantee that the protection features are enabled / available in every case
  - Depends on the used compiler and the target platform
  - Especially with small devices (mobile, embedded, sensors,...), the features are sometimes not supported
- Against some types of buffer overflows, the countermeasures won't help at all (two such scenarios will be discussed in the security lab)
- There exists buffer overflow techniques that can circumvent the protection measures to a certain degree (using, e.g., return-oriented programming (ROP), see SWS2)
- The measures usually mean the program is terminated in case a problem is detected, which is critical from an availability point of view
- → It's still important that you prevent buffer overflow vulnerabilities in your code and consider the protection techniques as a second line of defense (defense in depth)

So, is the Buffer Overflow Problem solved?

Looking at all the features offered by compilers and operating systems, can we conclude the buffer overflow problem is solved? No! First of all, there's no requirement for these features to be enabled – software you have developed and deployed will run independent of whether the features are used or not. Second, modern (mobile) platforms or small platforms for embedded systems may not offer such protection features in all cases yet. Third, there are some types of buffer overflows (notably read overflows), where the protection measers don't help at all. Fourth, new buffer overflow techniques are continuously published that sometimes make the protection features useless or at least reduce their effectiveness (e.g., using so-called gadgets to circumvent all measures that are used today, see, e.g., *http://antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/*). And fifth, these measures often mean the program is terminated in case a problem is detected, which is critical from an availability point of view.

As a result, it's still important that you try to prevent buffer overflow vulnerabilities in your code and consider the protection techniques as a second line of defense (defense in depth). Note that the CVE statistics (*http://web.nvd.nist.gov/view/vuln/statistics*) shows that buffer overflow vulnerabilities are still very frequent, but in general, exploiting them has become more complicated due to the technical protection measures.

Stack Canaries in Action (1)

- Consider the function *gets* in C:

```
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.

- This is a «dangerous function» (and therefore part of the API abuse kingdom) because it's not possible to use this function in a secure way
- Can you explain why?

gets

For a description, see: *http://www.opengroup.org/onlinepubs/009695399/functions/gets.html*

Stack Canaries in Action (2)

- Consider the following simple C program that uses *gets*:

```
#include <stdio.h>

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

- This program obviously has a buffer overflow vulnerability
  - If more than 25 characters are entered, `buffer` is overflowed and it can be expected that `old ebp` and `ret addr` are overwritten
  - The program is compiled with *gcc* on Linux, using no specific options: *gcc -o gets getc.c*

- If fewer than 25 characters are entered, the program works correctly:

```
user@securitylab-ubuntu:~$ ./gets
Enter Text : AAAAAAAAAA
user@securitylab-ubuntu:~$
```


Usage of *gcc*

The program is compiled as follows: *gcc -o gets gets.c*. In this case, stack canaries are enabled, i.e., it is the default behavior of *gcc*.

Note that *gcc* produces a warning during compilation that gets should not be used: *gets.c:(.text+0x35): warning: the `gets' function is dangerous and should not be used.* For you as a programmer, this should by a clear indication to switch to another function!

Stack Canaries in Action (3)

- If more than 25 characters are entered, the following happens:

```
user@securitylab-ubuntu:~$ ./gets
Enter Text : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

  - The reason is that *gcc* uses stack canaries per default – in this case, the canary on the stack was overwritten, which was detected during runtime
  - The program is aborted before jumping to the overwritten `ret addr`, so no harm can be done by overwriting it (e.g., in a code injection attack)

- Compiling the program without using stack canaries results in the «typical» behavior of a buffer overflow: a segmentation fault
  - *gcc -fno-stack-protector -o gets getc.c*

```
user@securitylab-ubuntu:~$ ./gets
Enter Text : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

  - In this case the program jumped to the overwritten `ret addr`, which means attacks by overwriting `ret addr` are possible

Disabling Stack Canaries

Disabling stack canaries in *gcc* is possible with the *gcc* option *-fno-stack-protector*, which means the program is compiled as follows: *gcc -fno-stack-protector -o gets gets.c*

Race Conditions

Race Conditions

- Race conditions are possible in scenarios where multiple threads or processes are run simultaneously and share resources
  - Today, that's the case in all operating systems and many applications
- Race conditions are a very special type of software error
  - 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 (availability), but they can also have more serious security implications
- Part of the Time and State kingdom

Find the Security Error Exercise 3

- Assume a multithreaded web application uses the class below to generate random, 16 bytes long session IDs to track the users
  - Other components use this class by first calling *createSessionID* and then *getSessionID* to get a new session ID

```
public class SessionIDGenerator {
  private static Random rng = new Random();
  private static String newSessionID;

  public static void createSessionID() {
    byte[] randomBytes = new byte[16];
    rng.nextBytes(randomBytes);
    newSessionID = Util.toHexString(randomBytes);
  }

  public static String getSessionID() {
    return newSessionID;
  }
}
```

- This contains two major security errors. Can you spot them? To which kingdoms do they belong? How could they basically be fixed?

Find the Security Error Exercise 3 – Solution

1st Problem: *Random.java* is not suited to create cryptographically secure random data – a stated in the API documentation:

> Instances of `java.util.Random` are not cryptographically secure. Consider instead using `SecureRandom` to get a cryptographically secure pseudo-random number generator for use by security-sensitive applications.

- Kingdom: API Abuse (wrong security assumptions)
- Solve it by switching to *SecureRandom.java*

2nd Problem: Generating session IDs is not thread-safe

- Thread A calls *createSessionID* => Sets *newSessionID* to AAAA
- Thread B calls *createSessionID* => Sets *newSessionID* to BBBB
- Thread B calls *getSessionID* => Gets BBBB
- Thread A calls *getSessionID* => Also gets BBBB
- This means two users are using the same session ID → which likely means one user can access the (possibly authenticated session) of the other user
- Kingdom: Time and State (race condition)
- Solve it by implementing a thread-safe session ID generator
  - E.g., by just using one method with creates and returns the session ID and that does not use a variable that's shared among multiple threads to temporarily store the session ID!

File Access Race Conditions

- Race conditions not only occur in multithreaded programs, but also when different processes on a system share resources
  - This is where file access race conditions come into play

- These 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 indeed done on the current file
    - 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) errors

Find the Security Error Exercise 4

- The following code is part of a program that allows writing to files
  - The user starts the program and can specify the file (pathname) to write
  - The program runs with root rights (setuid root) → has access to any file
  - Therefore, the program checks if the user running the program has write access to the specified file before write access is allowed

```
if(!access(file, W_OK)) {  /* checks if the user that started the
                              program has write access to file
                              (pathname), returns 0 if true */
  printf("Enter data to write to file: ");
  fgets(data, 100, stdin);
  fd = fopen(file, "w+");  /* open the file for writing */
  if (fd != NULL) {
    fprintf(fd, "%s", data);
  }
} else {  /* user has no write access */
  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?

access

The access() function checks the file named by the pathname in the first argument for accessibility according to the access right in the second argument. It uses the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

The access function is defined by the POSIX.1 standard (unistd.h), the base of the Single Unix Specification, and should therefore be available in any conforming (or quasi-conforming) operating system/compiler (all official versions of Unix, including macOS, Linux, etc.).

For a description, see: *http://www.opengroup.org/onlinepubs/009695399/functions/access.html*

This example is based on an example in *John Viega, Gary McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley Professional Computing*. The book's chapter about race conditions is also available online: *http://www.informit.com/articles/article.aspx?p=23947*.

Find the Security Error Exercise 4 – Solution

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

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

  - Now the attacker starts the program to write to *pointer*
  - Between checking the access rights and opening the file, the attacker deletes the link and sets a new link with the same name, 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
```

- How could this be fixed?

Avoiding TOCTOU Problems

- Minimize the number of function calls that take a filename as 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

- In general, avoid checking of file access rights on your own and leave that to the underlying operating system
  - E.g., by not running the program with root rights (setuid root) but with the rights of the user
  - But this is not always possible as the program may require root rights for other activities
    - If you really have to run a program with high privileges (e.g., setuid root), think especially hard about security!

Operating System File Access Control

This can usually only be used when local system users use local applications.

Summary

- There's a wide spectrum of different types of security-relevant software problems that can happen
- One way to classify them is according to the 7 (+1) Kingdoms of Software Security Errors
  - Input Validation and Representation, API Abuse, Security Features, Time and State, Error Handling, Code Quality, Encapsulation, Environment
- Buffer overflows are a prominent type of security error and belong to the Input Validation and Representation kingdom
  - They happen if data is written or read beyond the end of an allocated buffer
- 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)