Secure software engineering

Chapter 8: Secure coding – part 1

Fraunhofer IESE
Institut Experimentelles Software Engineering

Fraunhofer-Platz 1
67663 Kaiserslautern
Germany

Dr. Holger Peine

Holger.Peine@iese.fraunhofer.de

Lecture at Kaiserslautern University of Applied Sciences, winter term of 2007/08

"Secure Software Engineering" - Overview of this course

1. Introduction: IT security and software security
2. Fundamental notions and definitions of software security
   - *(Crash course on web application security*
     - *Not really software engineering, but needed for many examples)*
3. Vulnerabilities and attacks (2 lessons)
4. Security in the software development process
5. Security requirements elicitation (½ lesson)
6. Threat analysis (1½ lessons)
7. Security in architecture and design (2 lessons)
8. **Secure coding (2 lessons)**
9. Quality assurance: Inspections, testing, static analysis (1½ lessons)
10. Process models
11. Usability, conclusions

Security at the source code level

```
GetInput(input);
strcpy(buf, input);
```

- Arguably, the majority of vulnerabilities arise during coding
  - Ever more complex programming platforms with their own problems
  - Many implicit assumptions about the platform that can be subverted
  - Secure designs can be refined to insecure implementations
- Be an expert about your programming platform‘s problems
  - Needs deep understanding and learning long lists of „Do‘s“/„Don‘t“s (no short cuts available here ☹) – way longer than we can enumerate here
  - Choose a (more) secure platform right away (e.g., avoid C)
- Make the above expertise easily accessible and documented
  - Coding guidelines (general, API-specific, company-specific)
  - Integrate advice and guideline adherence checking with the IDE

8. Secure coding

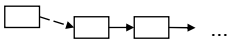
8.1 Advice for specific programming languages

8.1.1 C

C's main problem: Manual memory management

- In C, there are three classes of memory available for program objects:
- **Static**: Allocated at program start, valid until program termination (global object)
  - `struct ComplexObject x;` // defined outside any function
- **Automatic**: Objects local to a function, implicitly allocated on function entry, implicitly deleted on function exit
  - ```
    function f() {
        ComplexObject y; // y is valid now
        ...
    } // y is no longer valid now
    ```
- **Dynamic**: Must be ***manually*** allocated and deleted; remains valid until deleted
  - `ComplexObj* cp = malloc(sizeof(ComplexObj)); *cp = z; free(cp);`
  - ... makes **`cp`** point to a memory block large enough to hold one **`CObj`**; **`*cp`** *is* that **`CObj`**

Double free errors



- Deleting a dynamically allocated object marks its memory as free again
  - Implemented by the C run-time system: Adds the block to the list of free blocks
- Deleting the same pointer more than one time will corrupt the memory management system
  - ```
    if (condition) {
        free(p);
    }
    DoSomeThings();
    free(p); // double free if condition was true
    ```
  - Depending on the run-time system implementation, the various effects may occur, including buffer overflows ☹
- For each type, have a clear structure regarding who creates and who deletes objects

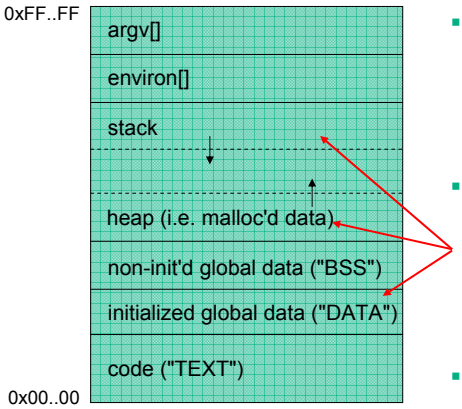
Memory leaks

- The opposite of the double free error: No **`free()`** called at all
  - **`int* p = malloc(...);`**
    `...` // use **`*p`**
    `p = q;` // forgot to call **`free(p)`**
    // now the old **`*p`** object can no longer be accessed (nor deleted!)
- Forgetting to delete an object that is no longer needed may exhaust the available memory ("leak" memory)
  - ... because C has no implicit garbage collection like Java, C#, PHP etc.
- If the memory leak may be triggered by an attacker, a denial-of-service attack by memory exhaustion is possible
- Again: Have a clear structure for object creation and deletion

Accessing deleted memory

- Accessing a dynamic object that has been deleted before may result in unpredictable effects if the block has been allocated again meanwhile
  - **`int* p = malloc(...);`**
    `...` // use **`*p`**
    `free(p);`
    `...` // other things, including new **`malloc()`** calls
    `x = *p;` // reading from deleted memory
    `*p = 5;` // writing to deleted memory
- Reading will return unpredictable data from another, unrelated place
- Writing will destroy the data in another, unrelated place of the program
- Again, many such errors can be exploited by an attacker
- Again: Have a clear structure for object creation and deletion

The C model of a process's virtual address space



- Fundamental feature (flaw?): C programs may access any* arbitrary byte in its address space

```
*(char*)0x4EF71208 = i;
```

- A (malfunctioning) program can write arbitrary data to any* place in its address space
  - including places not meant for data, but for execution control
- Strict handling of memory access crucial for execution integrity

A simple buffer overflow example

- Cause: Writing too much data into a memory buffer of limited size

```
void f(char* data) {
   char buf[512];    // the buffer
   strcpy(buf,data); // overflows if data > 512 bytes
}
```

- Effect: The "overspilling" part of the input bytes (**`data[512+]`**) overwrites the program memory (anything behind **`buf[511]`** in memory)
- Possible consequences
  - In general: The program shows arbitrary, mysterious behavior
  - In practice: With most data[], the program simply crashes
  - But what if the overflow data[] is chosen very carefully, even maliciously?

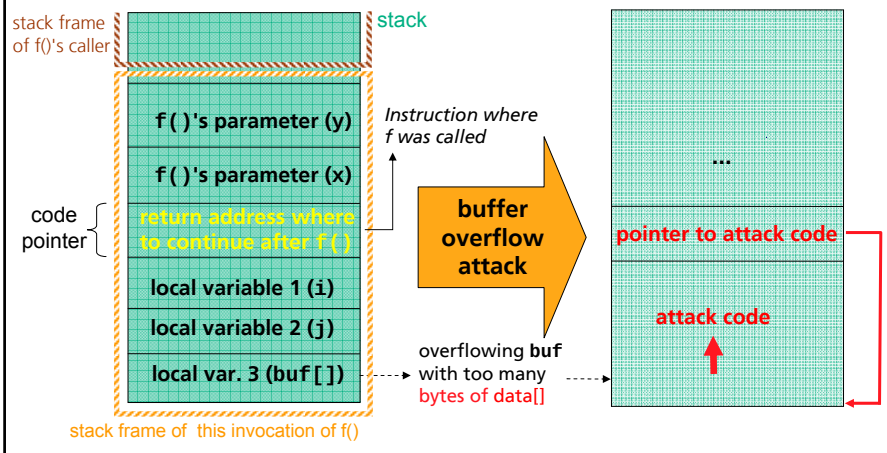
Exploiting a buffer overflow to inject attack code

- A buffer overflow is ***exploitable*** if the overflow data can be influenced (usually: provided as input data) by an attacker

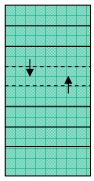
```
void f(char* attackerinput) {
    char buf[512];
    strcpy(buf, attackerinput);
}
```

- Exploitation for code injection
  1. By providing ***attack code bytes*** as `attackerinput`, an attacker can inject code of their choosing into the program's address space
  2. By overwriting a ***code pointer*** to point to the injected code, the attacker can redirect the program's flow of control into that code and take over control
     - Problem: Must know the address of the buffer in order to point to it

Stack buffer overflow: Overwrite a function's return address



Various targets for buffer overflows with varying danger



- Stack
  - Pointer and buffer easy to guess (stack layout relatively predictable)
- Heap
  - Function pointers at known offsets in dynamically allocated objects
    - C++: Function pointer in nearly every object (**`_vtable`** pointer!)
  - Difficult (but possible) to predict address of an object or a buffer on the heap
- Internal data of the run-time system
  - Global offset table (GOT), GNU C++ .ctors/.dtors lists, exception handlers, ...
  - Many code pointers, often predictable buffer addresses
- Data and BSS segment: Rarely code pointers here (though easy to predict)
- (Code segment: impossible – no data buffers here, and often not writable)

Defence against overflow attacks: Programmer level

- Don't use C ☺
  - Sometimes impossible (e.g., only C compiler available for target hardware)
- Careful programming?
  - Always track the size and non-allocated rest size of each buffer
  - Use **`strncpy()`** instead of **`strcpy()`** (with correct **`n`** parameter!) etc.
  - Not very reliable ☹
- Use data structures with implicit bounds checking instead of raw arrays
  - In C++: **`vector<int>::at`** instead of **`int[], string`** instead of **`char[]`**
  - In C: Use a better string library than libc, e.g. SafeStr, MS Strsafe
  - Small performance penalty implied in all cases

Defence against overflow attacks: System level

- C compilers or C library with built-in stack checking
  - StackGuard / StackShield / SSP compiler extensions, VC++.NET /GS, libsafe library
  - All leave some loopholes open (e.g. frame pointer overwriting, "return-to-libc")
- Non-executable stack segments
  - Needs OS and hardware support for efficiency (only on 64 bit, e.g. Itanium, AMD64)
- Random positioning of segments in address space
  - Repositioning code needs position-independent code ⇨ recompilation, loss of efficiency
  - Tricks with relative code pointers ("trampoline") still possible
- Static analysis tools to detect overflows in source code
  - Free: Flawfinder, ITS4, RATS, ... rather immature, lots of false alarms
  - Commercial: Fortify, Coverity, Klocwork, etc.: More mature, unclear abilities, expensive

C format strings

- C's standard printing function **`printf`** uses a "format string" as its first parameter, controlling how the other parameters will be interpreted
  - *Ex.* **`printf("%s%d ", msg, i);`** prints **`msg`** as a string, then **`i`** as a decimal, then a blank
  - **`printf`** cannot check that the right number and type of parameters are actually present – it simply uses whatever "parameters" it finds on the stack!
- A **`%n`** in a format string will make **`printf`** write a number to a **memory(!)** address designated by an integer pointer parameter in the parameter position matching the **`%n`**
  - The number written is the number of characters printed so far (meant for debugging complex format strings)
  - *Ex.* **`printf("%s%n", msg, &x);`** prints a string **`msg`**, then writes to **`x`**
  - If no **`&x`** parameter given, **`printf`** writes to an address found on the stack

C Format string attack

- Programmers are lazy and sometimes simply write **`printf(msg)`**
  - Harmless if **`msg`** contains no % formatting characters
- If the format string comes from an attacker, the attacker may supply a very long format string containing **`%n`** in certain positions
  - *Ex.* **`"aaaa%naaaaaaaaaaa%n..."`** writes the bytes 4 and 10 to memory
  - Often, the attacker succeeds to write arbitrary bytes to arbitrary addresses ☹
- May also read from arbitrary addresses, starting at the current stack
  - **`%x...%x`** prints (as hex bytes) many bytes on the stack and upwards
- Precaution: Don't use untrusted input for a **`printf`** format string
  - Always use a format string with **`printf`**
  - Best use constant format strings if at all possible

C++ security pitfalls

- C++ is a superset of C
  - If you use C features like raw memory arrays, pointer arithmetic, format strings etc., you will have the same vulnerabilities, too
- C++ has a **`vector<`*`Type`*`>`** parameterized class: Safer than C arrays ☺
  - C++ has a **`string`** class that is safer than C character arrays
- C++ still has pointers, and its library ("STL") is largely pointer-based ☹
- Function pointers are more much frequent than in C ☹
  - e.g. implicit **`_vtable`** pointer in every polymorphic object
  - More targets to redirect control to injected code (no matter where that code is)
- If you use C++ with extreme discipline and rewrite all libraries, it could probably be secure – but the result would no longer be regarded as C++

8. Secure coding

8.1 Advice for specific programming languages

8.1.2 Java

The good things about Java (regarding security)

- Type-safe
  - Compiler, byte code verifier (load time), run-time checks (type, bounds, ...)
  - ⇨ No way to forge data (or even code)
- Garbage collection (automatic memory management)
  - No memory leaks, double deletions, accesses to deleted memory
- Support for running untrusted code
  - SecurityManager, "sandbox", name spaces / class loaders
- Enormous wealth of security APIs
  - All sorts of crypto, identity mgmt / PKI, authentication & authorization, XML / web service security, ... much more than this lecture can even mention

Authorization in Java: The SecurityManager

- If a **`SecurityManager`** is active (NOT the default!), any security-relevant resource access will automatically be checked by it
  - Checks invisible to the programmer if they succeed
  - SecurityManager throws a **`SecurityException`** if a check fails
-
```
public FileInputStream(File file) throws ... {
   String fname = (file != null) ? file.getPath() : null;
   SecurityManager secMgr = System.getSecurityManager();
   if (secMgr != null) {  // SecurityManager is active
      secMgr.checkRead(fname);
      // throws SecurityException if no permissions to open this file
   }
   ... // physically open the file
}
```

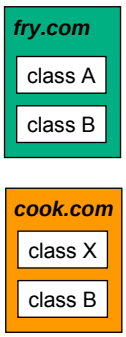

Policies in Java

- **`SecurityManager`** is the policy enforcement point (only one per JVM)
- Policy file defined in a text file as a list of permission grants as follows:
  - **grant** [**signedBy** <signer>] [**codeBase** <code source>] {
    permission <permission class> [<resource> [<action>]] ; };*
  - *Ex.*
    - **grant codeBase** "http://someserver/myjar.jar" {
      permission java.util.PropertyPermission "file.encoding", "read"; } ;
    - **grant signedBy** "John Doe" {
      permission java.io.FilePermission "myfile.txt", "read, write"; }
- Subjects: Code source, signer (when using JAAS, also logged-in user)
- Objects: Anything that has a permission class associated to it

Running untrusted Java code

- Use a **`SecurityManager`** (implies using a policy)
  - **`java -Djava.security.manager`** OR
  - **`System.setSecurityManager( new SecurityManager());`**
  - You may create your own **`SecurityManager`** by subclassing it
- Use a policy allowing only what the untrusted code really needs
  - Custom permission classes possible by extending existing ones
  - **`java -Djava.security.policy=my-policy.txt`**
- Use the bytecode verifier (OFF by default for code compiled with JDK ≤ 1.4!)
  - Checks code at load time for "well-behavedness"
  - **`java -verify`**

Name space separation for security



- Each code source constitutes a name space of its own
  - Implemented by using one class loader per code source
  - To prevent smuggling code across a trust boundary by class spoofing
  - Equal-named classes may coexist if in different name spaces
- Don't compare classes by name
  - **`if(obj.getClass().getName().equals("Foo"))`** // Wrong!
  - Opens the door to class spoofing across name spaces
  - Better: Compare by identitity of the classes' **`Class`** object (singleton!)
    - **`if(a.getClass() == b.getClass()) { ...`**
    - **`if(obj.getClass() == getClassLoader().loadClass("Foo")){`**

Java package scope

- Meant for visibility (to prevent unintentional misuse), not for security!
- A package is not a trust domain, since it can be added to by an attacker
  - (... except the java.* packages: A check to prevent this is hard-coded in)
  - Attacker code:
    ```
    package com.victim.somepackage; // possible even for attacker
    public AddedAttackerClass {
       // access the package-visible methods of the victim
    ```
- To prevent addition to a package, either ...
  - ... put the package in a sealed **`.jar`** file, or
  - ... set the property **`package.definition=`***mypackagename* in the **`lib/security/java.security`** properties file

Inner classes

- Inner classes may be defined within an outer class to ...
  - ... hide the name (but only the name!) of the inner class to other classes
  - ... provide "insider" access to the private(!) members of the outer class
- ```
  class Outer {
    private int i = 1;
    class Inner {
      public void f() {
        i = 5;  // access outer class's private member
    } }
    public static void main() {
      Inner in = new Inner(); // Inner defined here
    }
  }
  // Inner undefined here
  ```
  ```

Inner classes do not increase, but harm confidentiality

- At run-time, an object of an inner class is NOT hidden
  - The JVM does not know inner classes, so the Java compiler makes **`Inner`** a "normal" class at package scope, named **`Outer$Inner`**
  - **`Outer$Inner`**'s members may be accessed by all classes of the package
- And now the best thing ...
  - To realize access to the outer class's private members, inner classes will silently change the **`private`** fields of the outer(!) class to **`package`** scope
  - Any class in the package (remember packages are not closed!) may access the private members of the outer class, thus making it *less* secure
- Recommendation: Use inner classes for visibility limitations only, but not to implement security

Ensure that sensitive classes are not (de)Serializable

- Serialization is the process of turning a live Java object into a byte string
  - Useful e.g. for persistent storage or network transfer of Java objects
  - Implicit default implementation will be applied to all classes that implement ***`Serializable`*** unless they define their own (de)serialization methods:
    - **`void writeObject(ObjectOutputStream out) throws ...`**
    - **`void readObject(ObjectInputStream in) throws ...`**
- A serialized object is out of control of Java:
  Just a simple byte string that an attacker could manipulate at will, e.g.
  - Read secret data from private fields not accessible on a live object
  - Set sensitive data to values the attacker could not legally set on a live object
- Prevention: Make sensitive classes un(de)serializable by making their **`write/readObject`** methods throw an **`IOException`**

All security grants are off when using native code

- All protection provided by the Java system (compiler, verifier, JVM) does not have *any* power over native machine code accessing Java objects.That is, ...
  - Private data may be freely accessed
  - Any byte of any object may be read or written arbitrarily, ignoring all typing rules, invariants implemented by setter methods etc.
  - Objects may be forged (e.g. created without running its constructor)
  - No **`SecurityManager`** is ever invoked
- Consequences:
  - Never run untrusted native code
  - Check your trusted native code with extreme care

Croft/Tan (Boston College) are reviewing JDK's native code: Found about 100 vulnerabilities so far (covered 40 K of 800 KLoC so far)

8. Secure coding

8.1 Advice for specific programming languages

8.1.3 PHP

PHP

- PHP is a script language, mostly to generate dynamic web pages
- Server-side scripting: PHP code embedded in HTML pages will be executed by the server before sending the HTML page to the browser
  - Embedded PHP scripts will be replaced by their output: Dynamic content!
- *Server side*

```
<html>
<body>
Hello user, today is
<?php print_r(getdate()); ?>
</body>
</html>
```

Client side (browser)

```
<html>
<body>
Hello user, today is
Tue, May 23th 2007
</body>
</html>
```

- PHP scripts may access any data in the HTTP request, like URL parameters, form input etc. via predefined variables

PHP and security

- Design goals of PHP (especially noticeable in the default choices) were ...
  - Ease of programming
    - In practice, many programmers with superficial knowledge
  - Network transparency
    - Large and complex remote attack surface
- Non-goal: Security
  - Some signs of change recently, but very few, slowly and painfully
  - Already several failed attempts at better security (magic_quotes_gpc, safe_mode partially)
  - No coherent security model, many local inconsistencies

Accessing user input in PHP

- **`register_globals off/on`**:
  URL parameters overwrite(!) equal-name PHP program variables
  - Before PHP-4.2.0, **`register_globals on`** was the default
  - Best choice: **`register_globals = off`** (default since PHP 4.2.0)
- Second-best solution: Reset all global variables not explicitly designed for user input before their first use
  - Difficult if the uses are spread across many files: Which is the first use?
- Never access user input by global variables; preferably not by **`$_REQUEST`** (mixes GET and POST ☹) either, but through the predefined "superglobal" arrays **`$_GET`**, **`$_POST`** etc.

Treatment of local and remote files in PHP

- PHP often accepts URLs where file names would be expected
  - **`include $libdir/$usersFileChoice`**
  - Most frequently exploited vulnerability at present: Used to inject attack code
  - Deny loading of remote files by **`allow_url_fopen = off`**
- Deny uploading of files (attack code!) if possible
  - **`file_uploads = off`**
  - If indispensable, mistrust file name and type and save outside document root
- Limit file access to the directories intended for this:
  - **`open_basedir = /path/to/application/files`**

Miscellaneous PHP advice

- Disable displaying internal error information: **`display_errors = off`**.
- Disable failed input validation mechanism: **`magic_quotes_gpc = off`**
- Disable dynamic loading of modules: **`enable_dl = off`**
- If possible, use **`safe_mode[gid]`**: Frequently sensible restrictions on ...
  - ... available functions (e.g. no **`system(), exec()`**, ...)
  - ... file access (compares owner of script and file; causes problems with CMSs)
  - ... uploading files (forbidden)
  - ... any many other things (some of which are questionable)
- Don't make configuration file **`php.ini`** accessible on the web

8. Secure coding

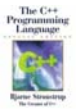
8.1 Advice for specific programming languages

8.1.4 How much does the language matter for security?

How much does the programming language matter for security? (1)

- Less than you perhaps think by now ☺
- Informal study by Cigital Inc.:
  - C programs have 4-5 vulnerabitlities per KLoC
  - Java programs still have 1-2 vulnerabitlities per KLoC

- Admittedly: Some languages have certain types of errors which simply don't exist in other languages
  - *Ex.* buffer overflows in C, register_globals in PHP



- *However*, in many cases these well-known problems can be mitigated by coding guidelines
  - e.g. use StrSafe in C, use safe_mode in PHP

How much does the programming language matter for security? (2)

- Modern languages like Java and C# are normally used for complex applications, which imply a corresponding potential for vulnerabilities
  - *Ex.* Race conditions between Java servlets processing HTTP requests may be harder to identify than in C code accessing files
- Many vulnerabilities (50%??) are caused at analysis and design time: The programming language is not relevant there
- Bottom line:
  - No language will automatically produce a secure application – it may only help or hinder somewhat
  - *If* the programmer is *aware* of a language's security problems and uses *discipline* to avoid them, security becomes a minor consideration in the multi-dimensional decision for a programming language for a certain project