Engineering Secure Software

DEFENSIVE CODING TECHNIQUES

Part 2

Last Time…

- Always code defensively
- Principles
  - Writing insecure code is easy
  - Maintainability still counts
  - Know thy APIs
  - Complexity is the enemy of security
  - Don’t be paranoid
  - Tree of Knowledge
- Validating Input
- Sanitizing Input
- Exception Handling
- Subclassing
- Immutability
- Concurrency
- Double-free

Concept: Attack Surface

- Most exploits enter in through the UI
  - Often the same interface the users see
  - Hence: input validation & sanitization
- Attack surface
  - The number & nature of the inputs for a given system
  - Can be quantified
  - Usually compared
- **Attack surface** increases with…
  - More inputs
    e.g. new input fields, new features
  - Larger input space for a given input
    e.g. allowing a markup language instead of plaintext

Let your GET mean GET

- HTTP protocols have different actions
  - GET – for retrieving data (typical usage)
  - POST, DELETE, etc. – modify stuff
- HTTP protocol specifies that GET actions should never have a *persistent* effect (e.g. model)
  - Even though you can encode parameters into URLs
  - Greatly helps mitigate cross-site request forgery (CSRF)
  - Rarely respected
- This is okay:

```
<a href="index.jsp?navigation=home">Home</a>
```

- This is not:

```
<a href="index.jsp?changeName=Bobby">Change Name</a>
```

Native Wrappers

- If you use another language, you inherit all of the risks in that language

  e.g. Java Native Interface (JNI) can execute a C program with a buffer overflow

- Also: treat native calls as external entities
  - Perform input validation & sanitization
  - Loaded at runtime → spoofing opportunity

Cloning is Insecure (and Medically Unethical!)

- Every Java object has a `clone()` method
  - Often error-prone
  - Doesn't do what you think it does
  - Most people don't abide by the contract
- Even the Java architects don't like it
  - The Java Language Secure Coding Guidelines from Oracle recommend not using `java.lang.Cloneable` entirely.
  - Use your own copy mechanism if needed

public static → final

- Global variables are evil
- Mutable global variables are an abomination
  - Increases complexity unnecessarily
  - Tampering concern in an untrusted API
  - Constants are the only acceptable use of globals
- Nice try, but still doesn't count:

```
public static final List<String> list = new ArrayList<String>();
```

Serial Killer

- Serialization is often unnecessary, difficult to get right
- Deserializing is essentially constructing an object without executing the constructor
  - If your system uses it, don't assume the constructor will be executed
  - Can reverse-engineer to violate constructor post-conditions
  - Complex input!
- Also, serialized != encrypted
  - Confidentiality disclosure
  - Use `transient` for variables that don't need serialization
    e.g. environment info, timestamps, keys

Memory Organization Assumptions

- Don’t rely upon the memory organization of the compiler and OS
- E.g. C-code:

```
char a=5;
char b=3;
*(&a+1)=0; /* b is now 0 */
           /* this works, but not advisable */
```

- Lots of problems with this
  - Compilers change
  - OS’s change
  - Dev environment vs. Customer environment
  - Really difficult to debug

Dead Store Removal

- Don't leave sensitive data sitting in memory longer than needed
  - Hibernation features dump RAM to HDD
  - Segfault → core dump → passwords!
- The following is usually a good idea...

```
void GetData(char *MFAddr) {
  char pwd[64];
  if (GetPasswordFromUser(pwd, sizeof(pwd))) {
    if (ConnectToMainframe(MFAddr, pwd)) {
    // Interact with mainframe
    }
    memset(pwd, 0, sizeof(pwd)); //clear password
  }
}
```

- ***…BUT!!!*** C++ .NET and gcc 3.x will optimize away that last call since `pwd` is never used again
  - So watch out for zealous compiler optimizations

Environment & File Confusion

- In C/C++, the `putenv()` and `getenv()` vary OS to OS
  - Change depending on the compiler and platform
  - Sometimes case-sensitive, sometimes not
  - An attacker can add an environment variable that overrides yours (e.g. to his own JVM)

```
putenv("TEST_ENV=foo“);
putenv("Test_ENV=bar“);
const char *temp = getenv("TEST_ENV");
if (temp == NULL) {   /* Handle error */ }
printf("%s\n", temp); /* foo on Linux, bar on Windows*/
```

- Same with file names in Windows and Linux
- Do not rely on case sensitivity when interacting with the platform

Watch Character Conversions

- Most apps require I18N in some form
  - You will need to convert one character set to another for translation
  - When apps “catch on”, I18N is usually an afterthought
- Not all character sets are the same size!
  - Assume 4-bytes for a character? Buffer overrun on Chinese chars
  - Not every byte maps to a character
  - Sometimes multiple bytes map to a single character
- Recommendations
  - Use unicode: UTF-8 or UTF-16
  - Don’t roll your own converters
  - Check: web servers, database systems, command inputs

DoS in Many forms

- Denial of service occurs in many, many ways
  - Overflow the hard drive
  - Overflow memory → page faults
  - Poor hashcodes → constant hash collisions
  - Slow database queries
  - Poor algorithmic complexity
  - Deadlocks, race conditions, other concurrency
  - Network bandwidth issues
- Recommendations:
  - Black-box stress testing
  - White-box, unit-level stress testing
  - Focus less on user inputs, more on the logic
  - Learn the art of profiling
    e.g. `java -agentlib:hprof`

Don’t Forget Config Files!

- Vulnerabilities can also exist in system *configuration*
  e.g. log overflow, hardcoded credentials, authorization problems
- Makefiles & Installation definitions
  - Insecure compiler optimizations
    e.g. dead store removal optimizations
  - Using out-of-date, vulnerable dependencies
- Also:
  - I18N configurations
  - General configuration
  - Example configurations
- Recommendation
  - Bring these up in code inspections
  - Look at the *defaults*, and *what is missing*

Other Defensive Coding via VotD

- Resource exhaustion
- Check the limits of your input
  - Integer overflows
  - Buffer overflows
- Error message information leakage
- Secure logging
  - Log overflow
  - Avoid logging stuff that's sensitive anyway
- Limit use of privileged features of the language
  - Use the Java Security Manager
  - Classloader override
  - Complex file system interaction
  - Reflection Abuse
  - More serialization restrictions