Learning security through insecurity

Michael Sonntag
Institute for Information processing and microprocessor technology (FIM)
Johannes Kepler University Linz, Austria

michael.sonntag@jku.at
Web application security

- 86% of all websites have at least one serious vulnerability (Ø: 56 vuln.)*
- Web applications are ubiquitous: Almost everything on the web is not a static webpage, but generated dynamically; most are based on a database
  - But see also the trend for mobile devices: HTML5 instead of Apps (which are similar anyway regarding security, but not the topic here)
- Practically all web applications are insecure
  - E.g. XSSed.com: Web site with XSS vulnerabilities. These included (and regularly occur again!): Google, Microsoft, Facebook, NYTimes, Ebay, McAfee, PayPal…
- Therefore: Education in website security is sorely needed!
  - Also: 57% provide some security training → 40% fewer vulnerabilities, solved 59% faster; but 12% lower remediation rate*

Part of the solution: more security education

- Security education is urgently needed – for all kinds of SW, but specifically for web applications because of their public attack surface

- Potential approaches:
  - Paper, book, lectures: Security problems and their solutions are described
    - Nice, but depends on attention & retention; lacks training in applying them
  - Penetration testing: Good for testers, but will help implementers little

- Proposed solution:
  
  **Interactive system where security problems are shown, can be tested, and remedies are also explained and can be tested too**
Existing systems

- **WebGoat: Insecure web application**
  - J2EE and ASP.NET environments; Java Runtime + Tomcat + Web application
  - Black-box testing to exploit vulnerabilities
  - No corrected (=secure) version available

- **HackThisSite: Numerous individual small web application**
  - Penetration testing: Users must hack them
  - No source code of the applications; no corrected versions
  - Includes also modifying executables, steganography etc.

- **Damn Vulnerable Web Application:**
  - Live-CD or existing web server; source code accessible
  - Three security levels available (insecure, programmed-tried-but-failed, secure)
  - Few vulnerabilities, no education (explanations e.g. for secure version lacking)
Proposed solution: aims & characteristics (1)

- Independent vulnerabilities: No “real” application
  - Easier to understand because code is short and little external elements contained
  - But: Differs significantly from real applications
    - Finding problems is more difficult, not-programming them gets easier!

- Providing solutions: For education
  - Each vulnerability exists twice: Once vulnerable and once fixed
  - This includes explanations why this is a fix and what other approaches are possible

- Versatile: Many security problems shall be exemplified
  - No artificial restrictions
  - Problems for server, other servers, and end-users
Proposed solution: aims & characteristics (2)

- Simplicity: “Pure” vulnerabilities; no pre-knowledge required
  - Avoiding frameworks, template engines, complex stylesheets etc.
  - To focus the attention on the security issues and remove extraneous clutter
- No hacking: Penetration testing is no goal
  - Testing only to understand problem & test own code; no exploitation (remote shell etc.)
- Self-contained: No additional software needed
  - Must be trivial to use; download should be small; no public webserver
- Own webserver implementation: Required for some vulnerabilities
  - Allows running several webservers and reduces complexity
- One-time bonus: Students implemented the individual vulnerabilities
The SecuritySampleServer

- Implemented in Java, as this is the main teaching language (here)
  - Also widely used for web applications
- Webserver: Implemented in Java too
  - Uncommon for a webserver, but here acceptable
  - Supports GET/PUT, cookies, and server-side-includes: Simple (and insecure!)
    - Started on port 8080 (80 might already be used), and 8081…
- Each “application” embodies a single vulnerability and is called directly
  - Specific “path” element for each, where individual subpaths are possible
- No database included in general, individual applications use an in-memory SQL database
Anatomy of a lesson (1)

- Each lesson consists of several webpages and two applications
- Introductory page
  - General description of the problem
  - Shows a color-coded table of the “danger” from this vulnerability:
    - Exploitability: How easy it is to exploit the vulnerability
    - Prevalence: How common is this problem
    - Detectability: How easy it is to find out if a problem has such a vulnerability
    - Impact: What can happen when it is exploited
- Insecure code: The full code of the example application
  - General elements are placed in a base class → typ. 50-150 lines of code
Anatomy of a lesson (2)

- Explanation of the vulnerability
  - Includes exploits and directions to try them (if possible direct links)
    - Can be tested manually, also with other parameters/data/…
  - Partially shows code and browser results
  - Database or other state: Reset link to restore the app to the initial version

- How to secure the application
  - Explanation how to prevent such an attack; what must be changed in the code
  - Links to same exploits as before, just not working any more
    - Own tests can be tried again too, obviously!

- Secure code: The code of the same application without the vulnerability
Restrictions and limitations

- Only “pure” vulnerabilities, but none specific to a language or a framework
  - Different languages might be included through a bridge
    - Especially interpreted languages, which learners could change directly
  - No buffer overflows (→ Java!): They are just explained

- Not available online: Too insecure
  - A secure tunnel to a cloned VM would be too complex/expensive

- Simplistic view because of independent vulnerabilities
  - Investigating code for them is different in reality; so less suited for educating testers

- No assessment: Learners can try out, but their activities are not logged
  - Formative self-assessment could be built into it
Example lesson: OS command injection (1)

A1-2: Command injection (Injection attacks)

The first section was about SQL injection. This flaw also consists in a kind of injection and because of this is part of the most popular attacks published by OWASP in its last edition.

The command injection follows the same principle as the SQL injection and other kinds of injection attacks: to insert code within the victim webserver. The command injection has a lot of possibilities for the attackers and could even cause the total compromise of the victim system and also to execute malicious code with the legitimate application’s (or web servers) privileges.

- Vulnerability explanation and how to exploit it
- Code Insecure
- How to secure the application
- Code Secure

Exploitability EASY
Prevalence COMMON
Detectability AVERAGE
Impact SEVERE
Example lesson: OS command injection (2)

**A1_2: OS command injection**

Output of the Echo command:

value

http://127.0.0.1:8080/A1_2/insecure/CmdEchoApp/?param=value **Try it in a new window**

But the attacker could try the following as input value:

`param=value&netstat -an -p UDP`

Or what is the same:

`param=value%26netstat -an -p UDP`

And the insecure application will answer with the private information:
Example lesson: OS command injection (3)

A1.2: Code Secure

```java
package simplewebserver.application.A1_2;

import java.io.IOException;
import java.io.OutputStream;

import simplewebserver.SimpleWebServer;

public class SecureCmdEchoApp extends CmdEchoApp {

    public SecureCmdEchoApp(final SimpleWebServer server) {
        super(server);
    }

    @Override
    public boolean handleRequest(String host, String path) {
        return path.equalsIgnoreCase("/A1_2/secure/CmdEchoApp/") && super.handleRequest(host, path);
    }

    @Override
    protected String[] getCmdLine(OutputStream responseStream, String cmd, String osName) throws IOException {
        // remove all non-alphanumeric characters except for some punctuation marks
        return super.getCmdLine(responseStream, cmd.replaceAll("[^A-Za-z0-9_.-]", ""), osName);
    }
}
```
Example lesson: OS command injection (4)

```
14
```

http://127.0.0.1:8080/A1_2/secure/CmdEchoApp/?param=value

But when the attacker tries to "steal" some information, the program avoids this by filtering all illegal characters from the passed string (here extremely strict: only letters, numbers and , - and . are allowed).

A1_2: OS command injection

Output of the Echo command:

value netstat -an -p UDP

The code changed is the function "getCmdLine":

```
@override
protected String[] getCmdLine(OutputStream responseStream, String cmd, String osName) throws IOException {
    // remove all non-alphanumeric characters except for some punctuation marks
    return super.getCmdLine(responseStream, cmd.replaceAll("[A-Z0-9_-.]", " "), osName);
}
```

Apart from method used before, we have some others tips in order to prevent OS Command injection:

- We can also use Java library calls to avoid direct calls to operating system commands.
- Ideally only predefined and fixed commands are issued and the application merely selects from them based on user input.
- We can use a Sandbox. A Sandbox is a security mechanism in order to separate the programs in execution from the rest of our machine. With this method we
Conclusions

- Practical evaluation is planned for next summer term (=next course)

- Planned extensions:
  - Gathering statistics on usage locally (potentially anonymized upload)
    - Might be extended to scoring: Provide hints based on tries, show completed elements etc. → Towards a tutoring system
  - Bridge to PHP, for interpreted languages
  - Differential view on code: Comparing the insecure and the secure code
  - Adding a different educational approach: Secure version is not available initially
    - Students only receive explanations and must correct the code, which is then automatically tested by the system (examples shown + additional ones)
The server can be downloaded under


Running it:

— Download and unpack to a local folder
— Run Start.bat (or start.sh - depending on OS; requires a JRE!)
— Open http://127.0.0.1:8080/ in your webbrowser
Thank you for your attention!

Michael Sonntag
Institute for Information processing and microprocessor technology (FIM)
Johannes Kepler University Linz, Austria

michael.sonntag@jku.at