

Auditing, Vulnerabilities, and Exploits

System Architecture and Security

René Rydhof Hansen

Aalborg University

26 SEP 2008

Goals

- You should be *familiar* with basics of software auditing
- You should be able to *recognise* common vulnerability patterns
- You should be able to *avoid* common vulnerability patterns

Secure?

```
public class Foo
{
    public void Bar()
    {
        String name = "filename";
        String msg = ...

        File f = new File("/tmp", name);
        FileWriter fw = new FileWriter(f);
        fw.write(msg, 0, msg.length());
        fw.close();
        f.setReadOnly();
    }
}
```

What does “secure” mean?

- Security policy
- Risk assessment/management
- Context

- Usually include: no race condition

Software Auditing

- Often (traditionally) performed once or twice (if at all)
- Security: Hard to demonstrate improved security (cost/benefit, long term investment, risk management)
- Security analysis: after completing preliminary iteration of system design
- Architectural Audit
- Implementation Audit
- Creative process
- Requires experience

When to perform an audit?

- All the time!
- Once or twice during the project life-time
- When customers complain
- During every iteration of the design/analysis phase

Who should perform an audit?

- Everyone!
- Not people directly involved
- “Objective” outsiders
- Compose audit group of many diverse talents
- Requires skill, creativity, endurance, **experience**

How to perform an audit?

- Focus on high-risk vulnerabilities
 - Easily exploitable
 - High-impact
- Phases
 - Gathering intelligence (aka. casing the target)
 - Analysis
 - Reporting

A Very Simple Approach to Risk Assessment/Management

- Evaluate ease of exploit/attack: easy, medium, hard
- Evaluate potential consequence: trivial, medium, severe
- Easy exploit with severe consequences = high risk
- Start from high risk and work downwards

Architectural Audit

- Most important audit
- Requirements should state not only “what” and “how” but also “why”
- Quote:

Security audits that focus on code can find problems, but they do not tend to find the major flaws that only an architectural analysis can find.

What to audit?

- High-level overview
- Requirements
- Policies (esp. security)
- Dependencies
- Design decisions
- Modules
- Inter-module connections
- Overall data-flow
- Trust relationships

Attack Trees (aka. itemised lists)

- Inspired by “fault trees” from engineering
- Basically a hierarchical list of potential attacks
- Helps recording and structuring... not necessarily creating
- For brainstorming: use whatever works for you
- Quote

*Unfortunately, building attack trees isn't much of a science.
Nor using them.*

Implementation Audit

- Lot of hard and boring work
- Don't accept the "no exploit" -fallacy
 - Don't waste time developing an exploit
 - Re-write suspicious code (the OpenBSD approach)
- Scour all resources for known vulnerabilities
- Think about user controlled "data"
 - Input (defined very broadly and generally)
 - Environment (logical as well as physical)
- Tool support
 - Two classes: validation and bug finding
 - Exploding market
 - Often: many many false positives

- Quote:

Usually it's not worth anyone's time to look for implementation-level problems...

Vulnerability: Race Condition

```
public class Foo
{
    public void Bar()
    {
        String name = "filename";
        String msg = ...

        File f = new File("/tmp", name);
        FileWriter fw = new FileWriter(f);
        fw.write(msg, 0, msg.length());
        fw.close();
        f.setReadOnly();
    }
}
```

Vulnerability: Race Condition

- Multi-threaded programs or multi-process environment
- Time-of-Check, Time-of-Use (TOCTOU)
- May exist at all levels (CPU to high-level architecture)
- Hard to guard against
 - Bug-free
 - Platform dependent
 - Performance issues (esp. locking, atomicity)

Secure?

```
public class Foo
{
    public void Bar()
    {
        String name = System.getenv("FOOFILERS");
        String msg = ...

        File f = new File("/tmp", name);
        FileWriter fw = new FileWriter(f);
        fw.write(msg, 0, msg.length());
        fw.close();
    }
}
```

Vulnerability: Input Validation

```
public class Foo
{
    public void Bar()
    {
        String name = System.getenv("FOOFILERS");
        String msg = ...

        File f = new File("/tmp", name);
        FileWriter fw = new FileWriter(f);
        fw.write(msg, 0, msg.length());
        fw.close();
    }
}
```

Vulnerability: Input Validation

- Very popular
- Good entry point
- Many many variations
 - Direct user input
 - Indirect user input (length of filenames, number of arguments, ...)
 - File access (config, data, log, ...)
 - System based (interrupts, signals, page faults, ...)
 - Modules, plug-ins, scripts, ...
 - Network
 - ...
- How to avoid?
 - Validate, validate, validate
 - Don't trust anything or anyone
 - Validate, validate, validate
 - Whitelist not blacklist
 - Validate, validate, validate

Secure?

```
public class Foo
{
    private int[] baz = new int [1];
    public int[] Bar()
    {
        ...
        return baz;
    }
}
```

Vulnerability: Private data, public method

```
public class Foo
{
    private int[] baz = new int [1];
    public int[] Bar()
    {
        ...
        return baz;
    }
}
```

Vulnerability: Private data, public method

- Reference to **private** data is returned by a **public** method
- Language-specific vulnerability
- Assume hostile, or at least compromised, virtual machine
- Defense in depth

Secure?

```
public final class Program {
    private String password;
    ...
    private final class InternalPasswordManager {
        private void setPassword() {
            ...
            password = ...;
            ...
        }
    }
    ...
}
```

Vulnerability: Inner Class Scope

```
public final class Program {
    private String password;
    ...
    private final class InternalPasswordManager {
        private void setPassword() {
            ...
            password = ...;
            ...
        }
    }
    ...
}
```

Vulnerability: Inner Class Scope

- JVM does not support inner classes; implemented as a “normal” class
- Results in surprising extension of scope, especially of private data of the **enclosing** class
- Language specific vulnerability
- Don't depend (too much) on specific language features
- Ensure: inner classes scope modifiers at least as restrictive as enclosing class'
- Compromised/hostile JVM

Secure?

```
public final class Program {  
    public String filename = new String("...");  
  
    ...  
}
```

Vulnerability: Non-final public field

```
public final class Program {  
    public String filename = new String("...");  
  
    ...  
}
```

- Language specific vulnerability
- Compromised/hostile JVM

Secure?

```
String cmd = System.getProperty("cmd");  
cmd = cmd.trim();
```

Vulnerability: Null-dereference

```
String cmd = System.getProperty("cmd");  
cmd = cmd.trim();
```

- Possible null-pointer dereference
- Normally: “only” an availability issue
- Not-normal: Mark Dowd exploit of Adobe Flash

Vulnerability: (Non-)Randomness and (Low-)Entropy

- Built-in pseudo-random number generators (PRNG)
- Home-made PRNGs
- Lowering the entropy (randomness) essential for crypto
- Case: Debian OpenSSL

Vulnerability: Buffer Overflow

```
void main()  
{  
    char buf[1024];  
    gets(buf);  
}
```

- Statically bounded buffer
- Dynamically bounded (i.e., unbounded) input
- Overwrites runtime memory (stack, heap, ...)
- Case: Too numerous

Summary

- Auditing
- Vulnerabilities
 - Race conditions
 - Input validation
 - Buffer overflows
 - Randomness and entropy