Security Focused Code Audit of Java Applications and Middleware

Marc Schönefeld, University of Bamberg
www.illegalaccess.org

RSA Conference 2005

Hello !

Hello, welcome to San Francisco

The speaker

- Marc Schönefeld, Diplom-Wirtschaftsinformatiker
  - **For Science**: External doctoral student @ Lehrstuhl für praktische Informatik at University of Bamberg, Bavaria, Germany
    - Thesis project:
      REFACTORING OF SECURITY ANTIPATTERNS IN
      DISTRIBUTED JAVA COMPONENTS
  - **For Living**: Department for Operational Security Management @ GAD eG, Münster/Germany,
    - Java, J2EE, CORBA [CSMR 2002]
      - design and development
      - Security Hardening (code audit)

The situation

- Java (we cover J2SE here, some aspects also apply to J2EE)
  - is designed as a programming language with inherent security features [Gong, Oaks]
    - JVM-Level: Type Safety, Bytecode integrity checks
    - API-Level: SecurityManager, ClassLoader, CertPath, JAAS
    - Crypto-Support: JCA/JCE, JSSE
  - **So what‘s the problem ?**

Selected Java Security Alerts in 2003/2004:

- Java Runtime Environment May Allow Untrusted Applets to **Escalate Privileges**
- A Vulnerability in JRE May Allow an Untrusted Applet to **Escalate Privileges**
- ...Java Virtual Machine (JVM) May **Crash Due to Vulnerability** in the Java Media Framework (JMF)...
- …Java Runtime Environment Remote **Denial-of-Service** (DoS) Vulnerability …

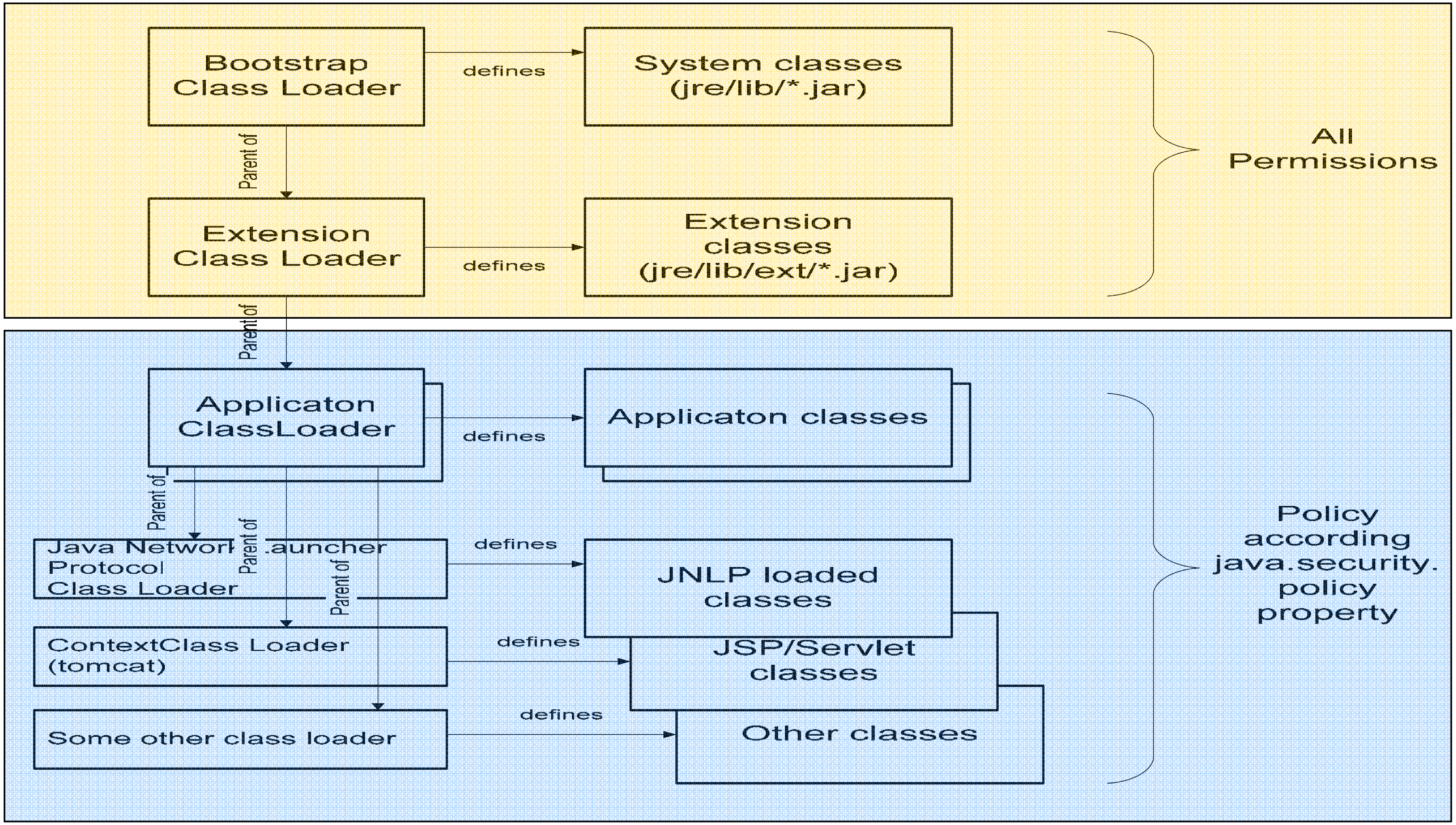
Despite of the precautions of the Java Security Architecture, a lot of attack potential …

what’s the cause?

The problem

- A platform (like the Java runtime environment) can only support the programmer’s intent
- What is programmer’s intent ? Reflects different perspectives …
  - **Functionality** [application programmer]
    - Java has a large API with lots of predefined functions (sockets, files, …)
  - **Quality** and **ReUse** [middleware programmer]
    - Java provides communication and marshalling on different semantic levels (Sockets, RMI, CORBA, Raw-Serialisation, XML-Serialisation, …)
  - **Safety** [security architect]
    - Java provides Isolation Support, Crypto-Objects and Secure Sockets out of the box
  - **Malicious Intent** [adversary]
    - Undermine security by finding the **weak spots**
    - Java VM and core libraries have (lots of?) **vulnerabilities**

Classloaders and Protection Domains

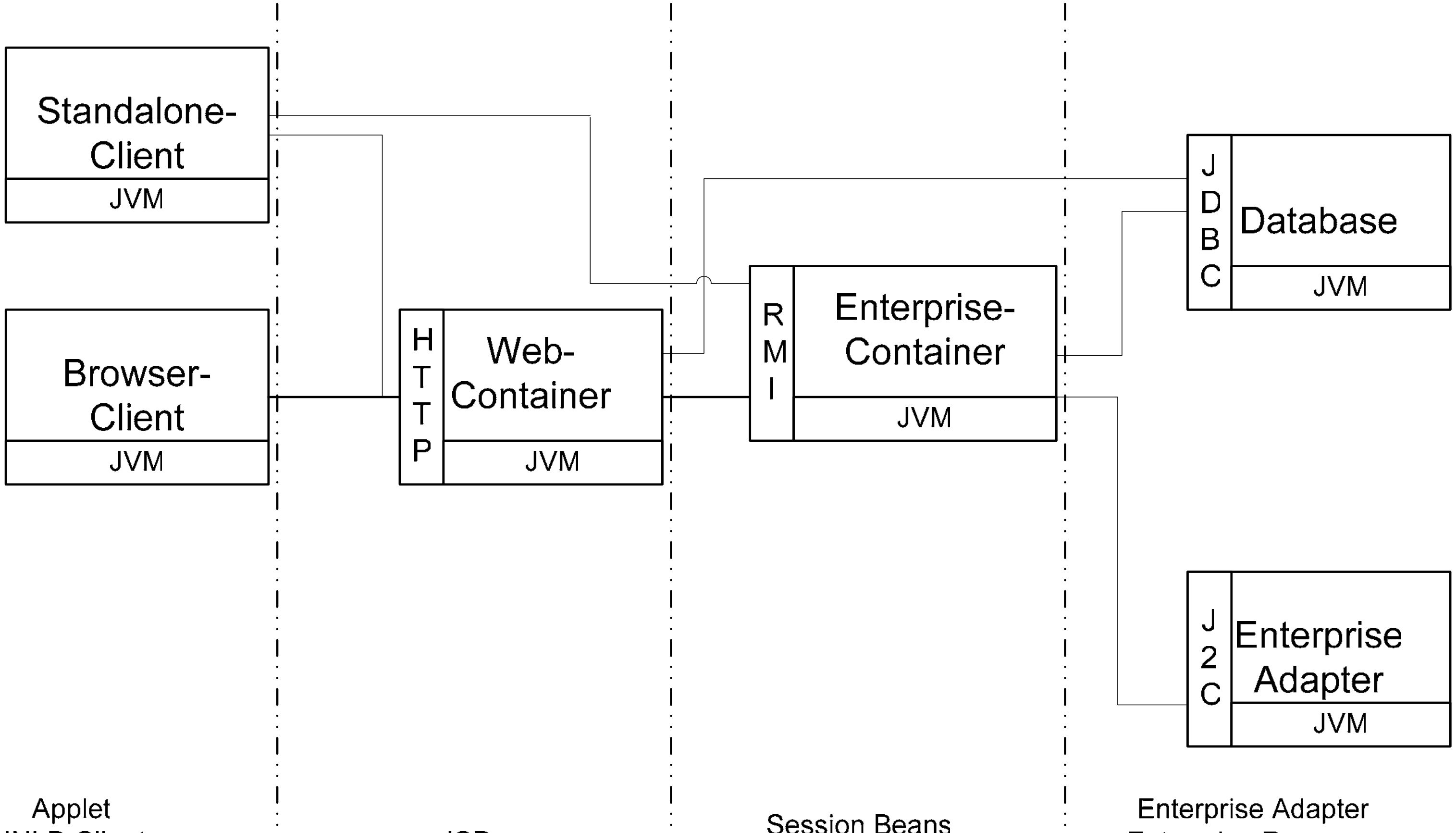


Why search for security bugs in java code ?

- **Component based software development**
  - 3rd party middleware components (web servers, graphics libraries, PDF renderer, …) are all over the place
  - We REUSE many of them in trusted places (bootclassloader)
  - But can we really trust them ?
- Questions:
  - Does my super duper 3rd-party graphics library include vulnerable object implementation that can be triggered by an attacker ?
  - Is the JDK secure in isolating my confidential XML data from other malicious applets loaded into the same VM ?
  - Object serialisation is safe, isn't it ?

J2EE multi-tier application types

Client Tier | Presentation Tier | Enterprise Tier | Backend Tier



Standalone-Client
JVM

Browser-Client
JVM

HTTP
Web-Container
JVM

RMI
Enterprise-Container
JVM

JDBC
Database
JVM

J2C
Enterprise Adapter
JVM

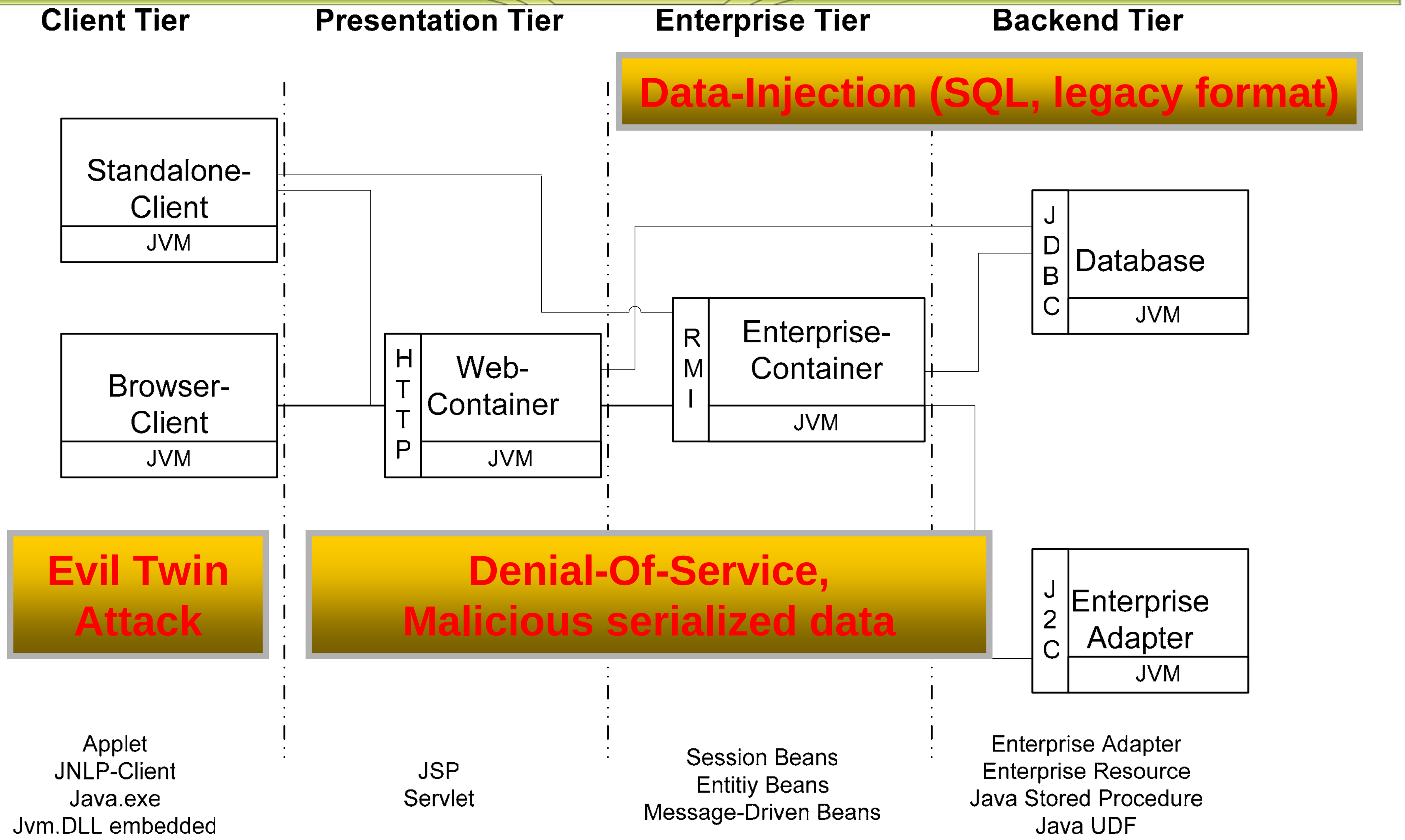
Applet
JNLP-Client
Java.exe
Jvm.DLL embedded

JSP
Servlet

Session Beans
Entitiy Beans
Message-Driven Beans

Enterprise Adapter
Enterprise Resource
Java Stored Procedure
Java UDF

J2EE multi-tier attack types



Client Tier

Presentation Tier

Enterprise Tier

Backend Tier

Data-Injection (SQL, legacy format)

Standalone-Client
JVM

Browser-Client
JVM

HTTP
Web-Container
JVM

RMI
Enterprise-Container
JVM

JDBC
Database
JVM

J2C
Enterprise Adapter
JVM

Evil Twin Attack

Denial-Of-Service, Malicious serialized data

Applet
JNLP-Client
Java.exe
Jvm.DLL embedded

JSP
Servlet

Session Beans
Entitiy Beans
Message-Driven Beans

Enterprise Adapter
Enterprise Resource
Java Stored Procedure
Java UDF

Java Security Patterns

– Sun’s **Security Code Guidelines (last update Feb 2, 2000!) :**

1. Careful usage of **privileged code**
2. Careful handling of **Static fields**
3. **Reduced scope**
4. Careful selected **public methods and fields**
5. Appropriate **package protection**
6. If possible Use **immutable objects**
7. **Never return a reference to an internal array** that contains sensitive data
8. **Never store user-supplied arrays directly**
9. Careful **Serialization**
10. Careful use **native methods**
11. Clear sensitive information

http://java.sun.com/security/seccodeguide.html

Java Security Antipatterns

– Security unaware coding create vulnerability by ignoring the security patterns

– Typical Java Secure Coding Antipatterns:

- Ignoring Language Characteristics (like Integer **Overflow**)
- Careless **Serialisation ,** careless use of **privileged code**
- **Inappropriate** Field and Method **Visibility**
- **Covert Channels** in non-final Static Fields

– They hide in your own code and the libraries you use

– Due to academic interest we audited parts of the **Sun JDK 1.4.x** and present the findings on the following slides:

How to search for security bugs in java code ?

Source Code Detectors	**PMD , Checkstyle**	useful only if source code is available and complete [in most of the cases it isn’t]
Decompilers	**JAD (!), JODE**	Time consuming analysis, needs experience
Bytecode audit analyzers	**Findbugs (bases on Apache BCEL)**	Bytecode detectors (visitor pattern): • predefined (software quality) • Self-written (for security audit)
Policy evaluation tools	**jChains** **(http://jchains.dev.java.net)**	• Test if program needs specific permissions Useful to reverse engineer protection domains

Bytecode analyzers

- The following discussion bases on JVM bytecode analysis
- **Findbugs** (http://findbugs.sourceforge.net)
  - Statical Detector for bug patterns in java code
  - Developed by the University of Maryland (Puth and Hovemeyer)
  - Open Source
    - based on the BCEL (Apache Bytecode Engineering Library)
    - Visitor-pattern analysis of
      - class structure and inheritance
      - control and data flow
    - GUI/command line
    - **And**: Extensible, **allows to write own detectors**

Java Security Antipatterns

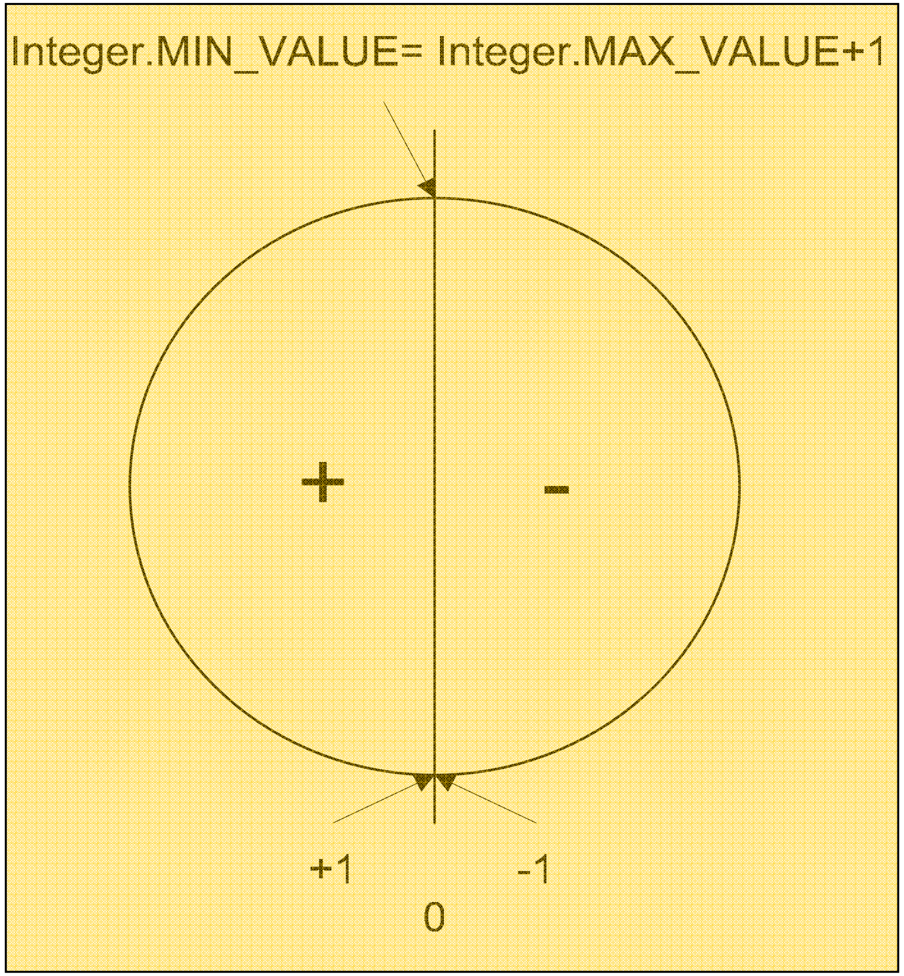
– **Antipatterns** (bugs, flaws) in trusted code (like **rt.jar**) cause Vulnerabilities

- **Availability**:
  - AP1: Integer, the Unknown Type(`java.util.zip.*`)
  - AP2: Serialisation side effects (`java.io.*`)
- **Integrity**:
  - AP3: Privileged code side effects (Luring attacks break sandbox)
  - AP4: Inappropriate Scope (Access control violation)
  - AP5: Non-Final Static Variables (Covert channels between applets)
- **Secrecy**:
  - AP6: Insecure Component Reuse (`org.apache.*` , Sniff private XML data between applets)

– **Goal**: Define a binary audit toolset to **detect the antipatterns** in your own and the 3rd-party components to be able to **fix the vulnerabilities**

Java Antipattern 1: Integer overflow

- According to *blexim* (Phrack #60) , integer overflows are a serious problem in C/C++, so they are in Java:
  - All Java integers are bounded in the [**-2^{31},$+2^{31}-1$**] range
  - In Java this is true: $-2^{31=}2^{31}+1$
  - Silent Overflow is a problem: Sign changes are not reported to the user, no JVM flag set
- Code of JDK **1.4.1_01** was based on the false assumption that java integers are unbounded, which led to a range of problems in the **`java.util.zip`** package



Integer.MIN_VALUE= Integer.MAX_VALUE+1

Java Antipattern 1: Integer overflow

The crash is caused by a parameter tuple

(**new byte [0],x,Integer.MAX_VALUE-y**), where **x>y x,y≥0**

➔silent overflow in the trusted JDK routines by fooling the parameter checks, so the overflow is neither detected by the core libraries nor the JVM.

➔The native call **updateBytes** to access a byte array leads to an illegal memory access. Consequently the JVM crashes.

```
D:\ > java CRCCrash
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : EXCEPTION_ACCESS_VIOLATION occurred at PC=0 x6D3220A4
Function= Java_java_util_zip_ZipEntry_initFields+0x288
Library=c:\java\1.4.1\01\jre\bin\zip.dll
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)
Dynamic libraries:
0x00400000 - 0x00406000 c:\java\1.4.1\01\jre\bin\java.exe
```

Java Antipattern 1: Integer overflow

The CRC32 class allows to calculate a checksum over a buffer:

If you have a byte buffer (1,2,3,4) and want to calculate the checksum over it you need to call:

```
CRC32 c = new java.util.zip.CRC32 ();

c.update (new byte []{1,2,3} ,0 ,3);
```

But if you do the following:

```
c.update (new byte [0] ,4 ,Integer. MAX_VALUE -3);
```

You will crash the JVM of JDK 1.4.1_01 and some versions of JDK 1.3.1

Java Antipattern 1: Integer overflow, Risk and extent

Risk:

If the attacker manages to exploit this function in an environment were multiple users share a single JVM (like a Lotus Domino server or a Tomcat HTTP server) he may cause a denial-of-service condition.

Extent:

More trusted functions were found vulnerable:

- java.util.zip.Adler32().update();
- java.util.zip.Deflater().setDictionary();
- java.util.zip.CRC32 ().update();
- java.util.zip.Deflater().deflate();
- java.util.zip.CheckedOutputStream().write();
- java.util.zip.CheckedInputStream().read();
- java.text.Bidi.<init >;

– http://developer.java.sun.com/developer/bugParade/bugs/4811913.html

- also bugnr = {4811913, 4812181, 4812006 , 4811927 , 4811917, 4982415, 4944300, 4827312,4823885}

Java Antipattern 1: Integer overflow, the Refactoring

Before
JDK
1.4.1
01
	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);
	}

After
JDK
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);
	}

Java Antipattern 1: Integer overflow, the Refactoring (bytecode)

Before (1.4.1_01)	After (1.4.1_02)
12: iload_2 13: iflt 28 16: iload_3 17: iflt 28 **20: iload_2** **21: iload_3** **22: iadd** **23: aload_1** **24: arraylength** 25: if_icmple 36 **Integer Overflow Bytecode Pattern**	12: iload_2 13: iflt 28 16: iload_3 17: iflt 28 **20: iload_2** **21: aload_1** **22: arraylength** **23: iload_3** **24: isub** 25: if_icmple 36 **Bytecode of Refactoring**

Java Antipattern 1: Harmful integer overflow, How to find during auditing ?

1. find candidate methods by detecting `iadd` opcodes
2. Does the `iadd` use user-supplied data (does it use data from the stack supplied by `iload` ?) to perform a range check
3. Is a native method called afterwards (`invokevirtual`, `invokestatic`), that takes the same data

This process can be implemented by a **Findbugs** bytecode detector

AP1: Conclusion and Suggestions

- The JVM does not provide an overflow flag like a normal x86 processor (designed in 1978), so there is no way to detect those conditions during runtime. The JVM in Java 1.5 (aka 5.0 aka Tiger) 27 years later does not improve this shortcoming
- Suggestions for JDK 6.0:
  - To avoid burdening the (security unaware) programmer, a bounded primitive integers (like in Ada) is helpful

    subtype Month_Type is Integer range 1..12;
  - If this is all too complex for the java compiler to handle, it could at least list a potential overflow as **compiler warning**

    (maybe in Java 6.0?)

Antipattern 2: Serialisation side effects

- The normal way to create a java object is to use the `new` instruction, which calls the constructor of a class
- But: The Java serialisation API (part of `java.io` package) allows to **bypass constructors** and create new instances of an object type by simply sending them to an `java.io.ObjectInputStream` (OIS), which is bound to a socket, a file or a byte array
- OIS objects are commonly used by remote communications such as RMI or persistency frameworks to import pre-built objects into the JVM
- When an object is read from an OIS the most derived `readObject` method of the class is called

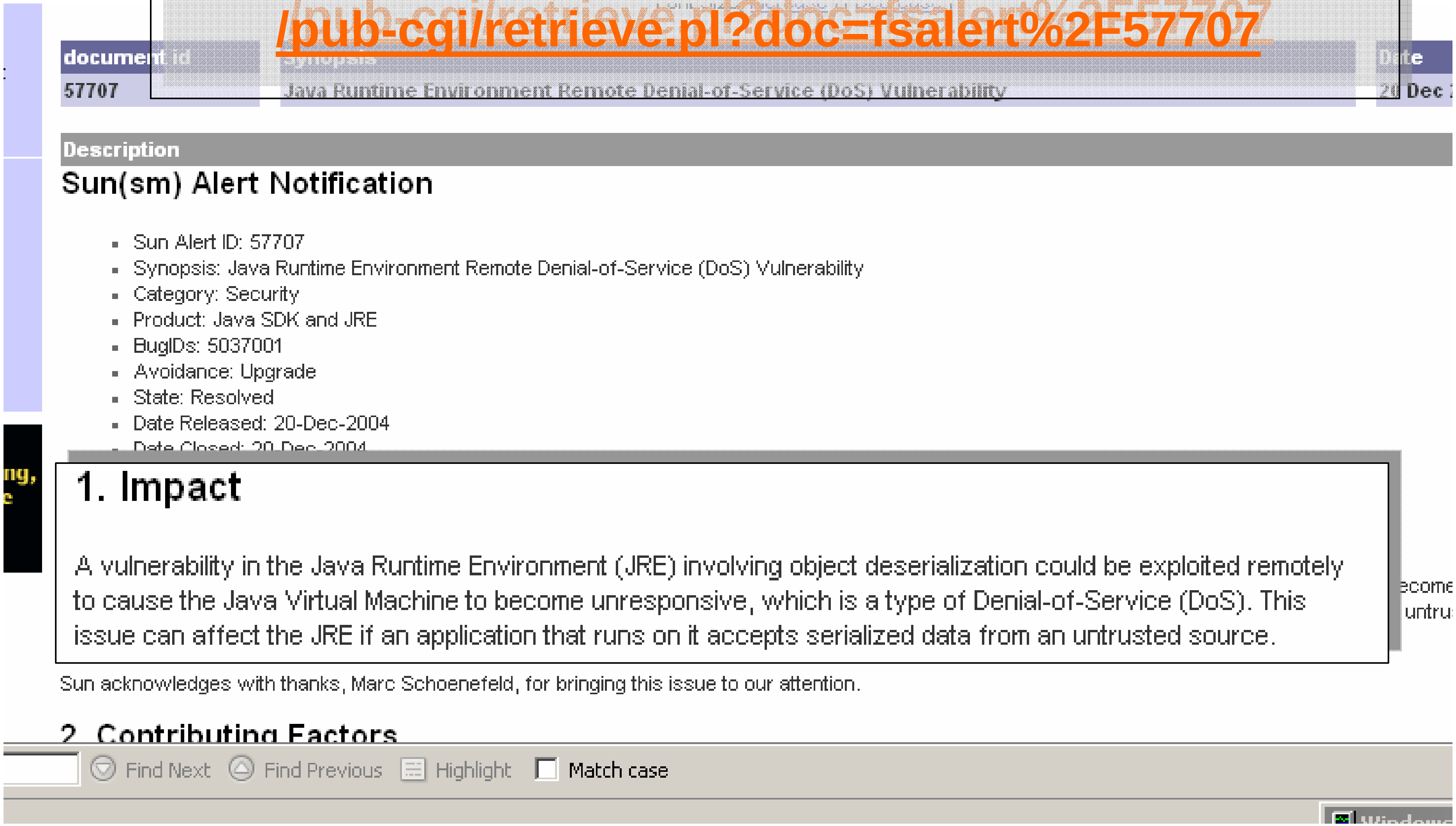
AP 2: Risk and Extent

- **Risk**
  - Reading serialized objects may force the JVM to branch into complex or vulnerable code regions that are called in the **readObject** method
  - **readObject** methods may linger in in your own code, the JDK classes and any 3rd party library you use
  - Attacker may prepare special handcrafted data packets with serialized data

- **Extent**

java.util.regex.Pattern	Triggers complex computation, „JVM may become **unresponsive**“ [Sun Alert 57707]
java.awt.font.ICC_Profile	Causes JVM **crash** on Win32
java.util.HashMap	Triggers an unexpected **`OutOfMemoryError`** which may kill the current listening thread and disable the service (as an error it bypasses most try/catch checks)

AP 2: Risk and Extent



http://classic.sunsolve.sun.com /pub-cgi/retrieve.pl?doc=fsalert%2F57707

document id	Synopsis	Date
57707	Java Runtime Environment Remote Denial-of-Service (DoS) Vulnerability	20 Dec

Description

Sun(sm) Alert Notification

- Sun Alert ID: 57707
- Synopsis: Java Runtime Environment Remote Denial-of-Service (DoS) Vulnerability
- Category: Security
- Product: Java SDK and JRE
- BugIDs: 5037001
- Avoidance: Upgrade
- State: Resolved
- Date Released: 20-Dec-2004
- Date Closed: 20-Dec-2004

1. Impact

A vulnerability in the Java Runtime Environment (JRE) involving object deserialization could be exploited remotely to cause the Java Virtual Machine to become unresponsive, which is a type of Denial-of-Service (DoS). This issue can affect the JRE if an application that runs on it accepts serialized data from an untrusted source.

Sun acknowledges with thanks, Marc Schoenefeld, for bringing this issue to our attention.

2. Contributing Factors

AP2: Serialisation side effects, a refactoring

Before

JDK 1.4.2 05

```
private void readObject(java.io.ObjectInputStream s)throws… {
  s.defaultReadObject();        // Initialize counts
  groupCount = 1;
  localCount = 0;  // Recompile object tree
  if (pattern.length() > 0)
    compile();// so we compile for the next 1600 years
  else
    root = new Start(lastAccept);
}
```

After

JDK 1.4.2 06

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

AP2: How to find during code audit ?

1. find candidate classes by detecting `readObject` definitions
2. For these classes determine if the control flow branch into harmful code
   - I. Search for algorithmic complexity (does it compile a regex for the next 800 years?)
   - II. Search for endless loops (bytecode backward branches)
   - III. Does to code call into vulnerable native code and propagates the total or some part of the payload ?

This process can be implemented by a **Findbugs** bytecode detector

AP2: Conclusion and Suggestions

- The `readObject` method is designed primarily for accepting and **checking** `Serializable` data
- Nested `readObject` invocations occur for nested `Serializable` classes, so the malicious payload does not have to be in the root object
- Try to defer complex operations from the time of creation to the time of first usage
- Similar considerations apply for the `readExternal` method which implements the receiving part of the `Externalizable` interface

AP3: Privileged Code Side Effects

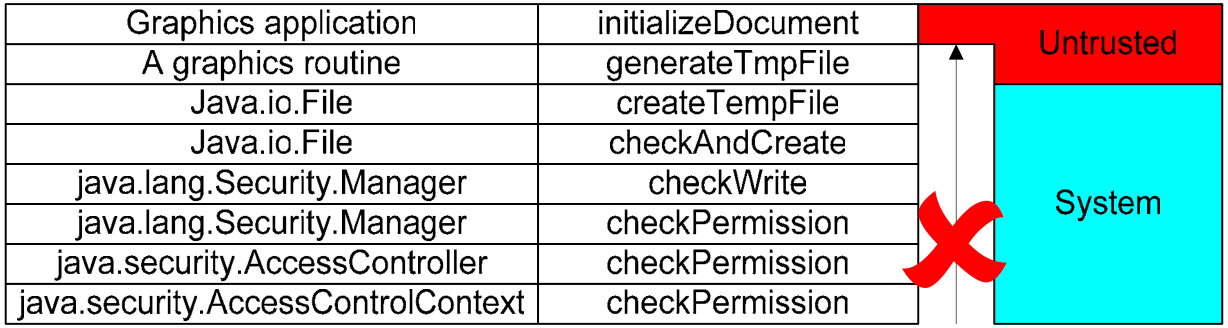
– The Basic Java Access Algorithm:

- **A request for access is granted if**, and only if every protection domain in the current execution context has been granted the said permission, that is, if **the code and principals specified by each protection domain are granted the permission.**
- A permission is only granted when all protection domain $\boldsymbol{D_i}$ contain the permission $\boldsymbol{p}$

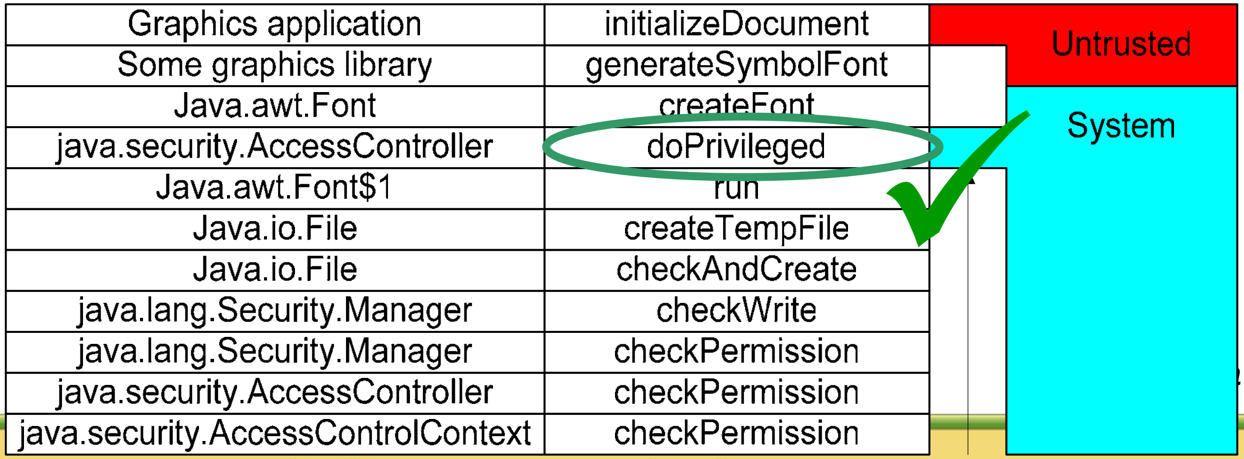
$$p \in \left\{ \bigcap_{i=1}^{n} D_i \right\}$$

AP3: Privileged Code Side Effects

- Privileged code (**doPrivileged**) is used to break out of the stack inspection algorithm
- Needed where the permissions on the application level (**user classes**) do not match the needed permissions to perform necessary operations on the middleware/system level (`rt.jar`)



Graphics application	initializeDocument
A graphics routine	generateTmpFile
Java.io.File	createTempFile
Java.io.File	checkAndCreate
java.lang.Security.Manager	checkWrite
java.lang.Security.Manager	checkPermission
java.security.AccessController	checkPermission
java.security.AccessControlContext	checkPermission



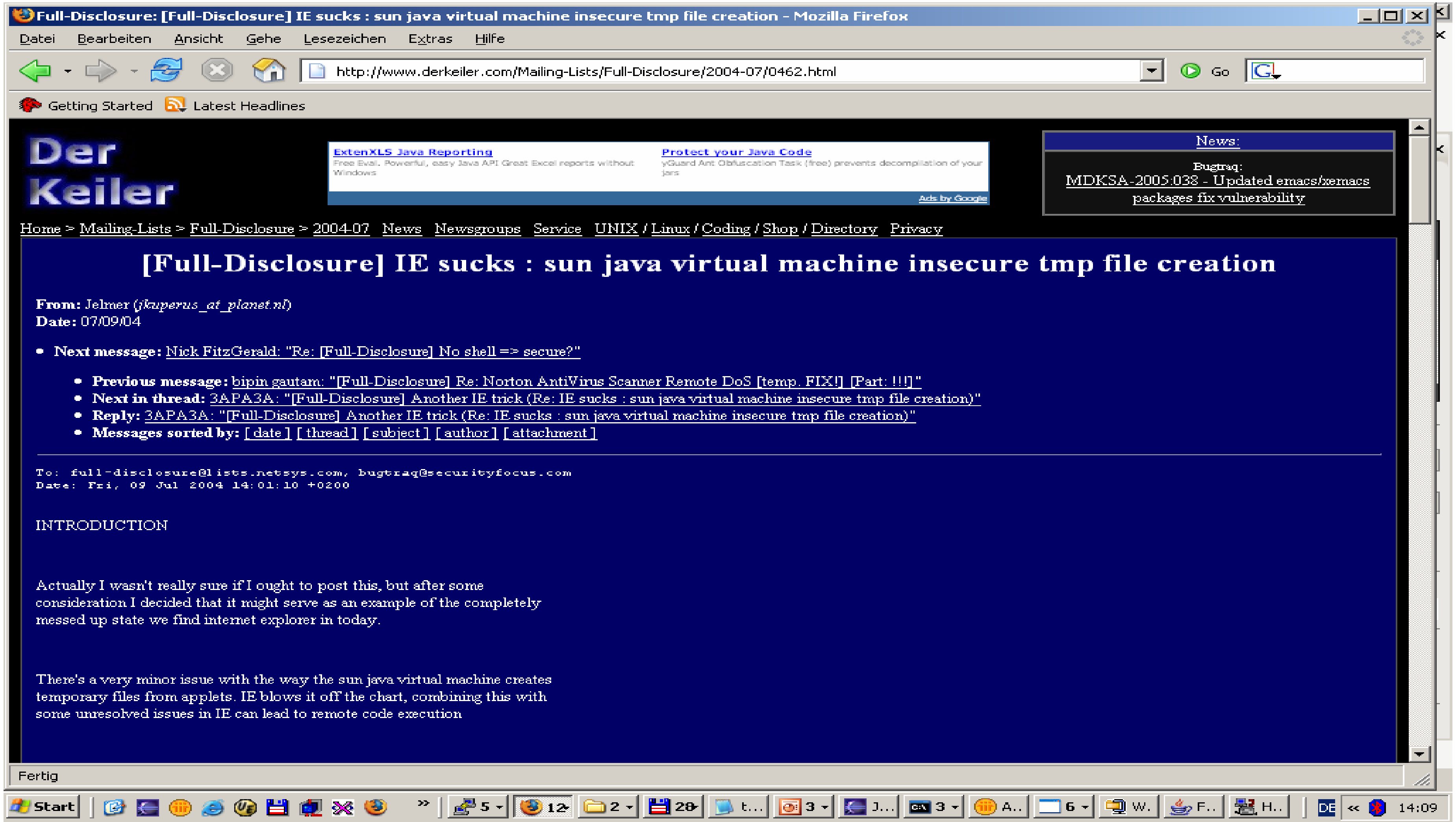
Graphics application	initializeDocument
Some graphics library	generateSymbolFont
Java.awt.Font	createFont
java.security.AccessController	doPrivileged
Java.awt.Font$1	run
Java.io.File	createTempFile
Java.io.File	checkAndCreate
java.lang.Security.Manager	checkWrite
java.lang.Security.Manager	checkPermission
java.security.AccessController	checkPermission
java.security.AccessControlContext	checkPermission

AP3: Privileged Code Side Effects: Risk and Extent

- **Risk**
  - An attacker may misuse this condition to escalate privileges and escape a limited protection domain (such as the JNLP or applet sandbox)
    - he knows the privileged code blocks in the JDK and the privileged codesources of the application
    - by a luring attack he tries to trick control into privileged code blocks and force that block to use parts of his injected payload
- **Extent**

java.awt.font.ICC_Profile	escape the applet sandbox and test existence of files on the client's machine
java.awt.Font (i)	transport temporary files (such as executables) to the client's machine, which can be launched later (http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-07/0462.html)
Java.awt.Font(ii)	fill up the remaining free space of file system of the client machine with a large file containing zero bytes
…	…

AP3: Privileged Code Side Effects: Risk and Extent



Full-Disclosure: [Full-Disclosure] IE sucks : sun java virtual machine insecure tmp file creation - Mozilla Firefox
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-07/0462.html
Go
Getting Started Latest Headlines
Der Keiler

News:
Bugtraq:
MDKSA-2005:038 - Updated emacs/xemacs packages fix vulnerability

Home > Mailing-Lists > Full-Disclosure > 2004-07 News Newsgroups Service UNIX / Linux / Coding / Shop / Directory Privacy

[Full-Disclosure] IE sucks : sun java virtual machine insecure tmp file creation

From: Jelmer (*jkuperus_at_planet.nl*)
Date: 07/09/04

- **Next message:** Nick FitzGerald: "Re: [Full-Disclosure] No shell => secure?"
  - **Previous message:** bipin gautam: "[Full-Disclosure] Re: Norton AntiVirus Scanner Remote DoS [temp. FIX!] [Part: !!!]"
  - **Next in thread:** 3APA3A: "[Full-Disclosure] Another IE trick (Re: IE sucks : sun java virtual machine insecure tmp file creation)"
  - **Reply:** 3APA3A: "[Full-Disclosure] Another IE trick (Re: IE sucks : sun java virtual machine insecure tmp file creation)"
  - **Messages sorted by:** [date] [thread] [subject] [author] [attachment]

```
To: full-disclosure@lists.netsys.com, bugtraq@securityfocus.com
Date: Fri, 09 Jul 2004 14:01:10 +0200
```

INTRODUCTION

Actually I wasn't really sure if I ought to post this, but after some consideration I decided that it might serve as an example of the completely messed up state we find internet explorer in today.

There's a very minor issue with the way the sun java virtual machine creates temporary files from applets. IE blows it off the chart, combining this with some unresolved issues in IE can lead to remote code execution

AP3: Refactorings

- No refactorings available
  - The described bugs are still in the JDK , so unfortunately no refactorings available
  - Although Most of those were reported to Sun in Q2/2004 or earlier

AP3: Privileged Code Side Effects: How to audit ?

1. find candidate classes by detecting **`doPrivileged`** calls
2. For these classes determine if user-supplied data is propagated to the privileged code block that causes to
   I. Pass access to protected resources
   II. leak secret data
   III. Perform unwanted modifications

   to untrusted code

This process can be partially implemented by a **Findbugs** bytecode detector

AP3: Conclusion and Suggestions

- **Conclusion**
  - **doPrivileged** is a **powerful** but **dangerous** construct to tweak protection domains

- **Suggestion**
  - To Sun:
    - Please fix bugs in privileged code JDK blocks
  - To Component Users:
    - Check 3rd party libraries for vulnerable **doPrivileged** blocks before usage, as they may break your security policy
  - To Middleware Developers:
    - Keep privileged code in own code as short as possible [http://java.sun.com/security/seccodeguide.html]
    - Detaint user-supplied data before propagating it to privileged code

AP4: Inappropriate Scope

- *As a rule,* ***reduce the scope*** *of methods and fields* ***as much as possible****. Check whether package-private members could be made private, whether protected members could be made package-private/private, etc. [Sun Security Code Guidelines]*

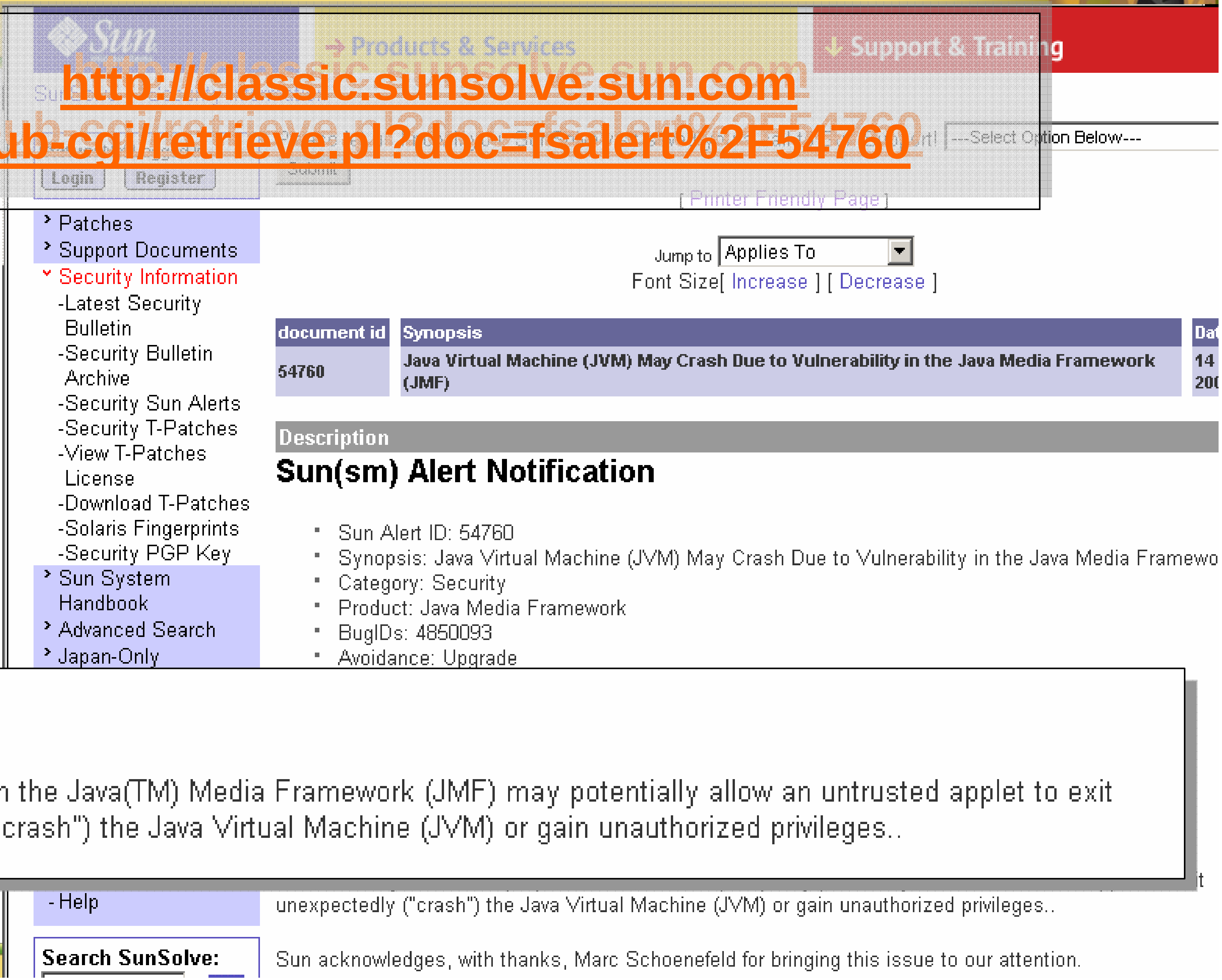
- *This should be especially true when you design trusted JDK extensions, such as the Java Media Framework (JMF)*

AP4: Inappropriate Scope: Risk and Extent

- **Risk**
  - An attacker can exploit the trusted protection domain “AllPermissions” of a java extension in **jre/lib/ext** to **escalate privileges**. For example the JMF
    - installs extra trusted classes to **jre/lib/ext**
    - accesses system memory via native routines
    - The public JMF class **com.sun.media.NBA** exposes a public pointer to physical memory [long value data]
    - So untrusted applets may read your system memory

AP4: Inappropriate Scope: Risk and Extent

http://classic.sunsolve.sun.com /pub-cgi/retrieve.pl?doc=fsalert%2F54760



document id	Synopsis	Dat
54760	Java Virtual Machine (JVM) May Crash Due to Vulnerability in the Java Media Framework (JMF)	14 200

Description

Sun(sm) Alert Notification

- Sun Alert ID: 54760
- Synopsis: Java Virtual Machine (JVM) May Crash Due to Vulnerability in the Java Media Framewo
- Category: Security
- Product: Java Media Framework
- BugIDs: 4850093
- Avoidance: Upgrade

unexpectedly ("crash") the Java Virtual Machine (JVM) or gain unauthorized privileges..

Sun acknowledges, with thanks, Marc Schoenefeld for bringing this issue to our attention.

1. Impact

A vulnerability in the Java(TM) Media Framework (JMF) may potentially allow an untrusted applet to exit unexpectedly ("crash") the Java Virtual Machine (JVM) or gain unauthorized privileges..

AP4: Inappropriate Scope: Refactoring

Before (JMF 2.1.1c)		After (JMF 2.1.1e)
public class NBA {	1	public final class NBA {
public void finalize() public Object getData() public Object clone() public void copyTo(NBA nba) public void copyTo(byte javadata[])	2	protected final synchronized void finalize() public synchronized Object getData() public synchronized Object clone() public synchronized void copyTo(NBA nba) public synchronized void copyTo(byte javadata[])

public long data;
public int size;
public Class type;
	3	private long data;
		private int size;
		private Class type;

}		}

1) **Creation of subclasses is forbidden, to prevent leaking of secret data by new methods**

2) **Scope of public finalize method degraded to protected, so no class can overwrite it**

3) **Data fields were moved to appropriate private (class local) scope**

AP4: Inappropriate Scope Side Effects: How to audit ?

1. find candidate classes by detecting `public` classes
2. For these classes determine if
   I. Data fields are declared as `public`
   II. Methods are declared as `public`
   III. Internal references to private, protected data are returned by a `public` method

The candidate selection can be implemented by using the predefined detectors of **Findbugs**

AP4: Conclusion and Suggestions

- **Conclusion**
  - Inappropriate Scope on fields and methods may allow to bypass access control mechanisms
- **Suggestion** [http://java.sun.com/security/seccodeguide.html]
  - *Refrain from using public variables. Instead, let the interface to your variables be through **accessor methods**. In this way it is possible to **add centralized security checks**, if required.*
  - ***Make sure that any public method** that has access to and/or modifies any sensitive internal states **includes a security check.***

AP5: Non-Final Static Fields

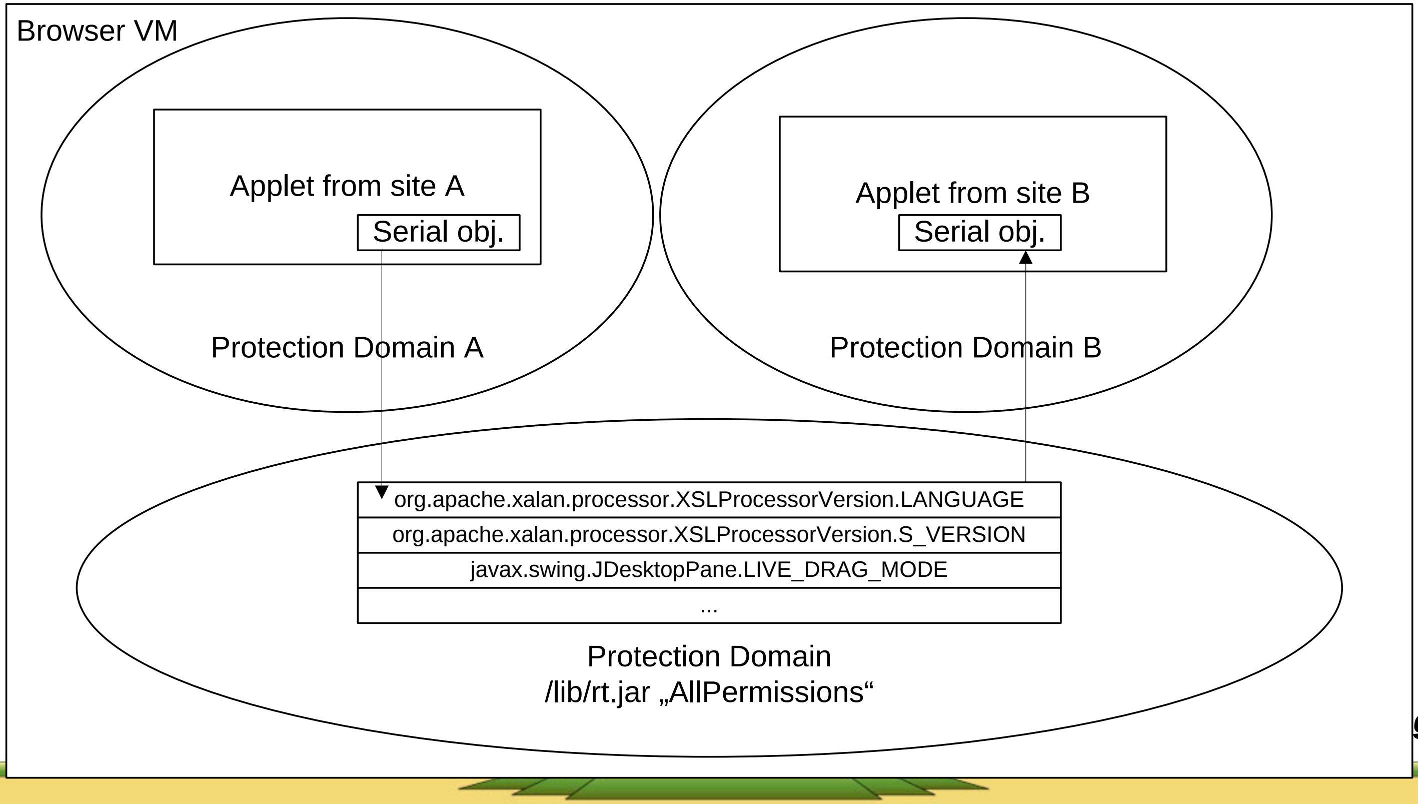
- *„Refrain from using non-final public static variables*
  - To the extent possible, refrain from using non-final public static variables because there is no way to check whether the code that changes such variables has appropriate permissions.
  - In general, be careful with any mutable static states that can cause unintended interactions between supposedly independent subsystems“

  [Sun Security Code Guidelines]

  - According to Sun Microsystems [http://www.sun.com/software/security/glossary.html] the term **covert channel** has the following definition:
    - A communication channel that is not normally intended for data communication. It allows a process to transfer information indirectly in a manner that violates the intent of the security policy.
  - We will show that the Antipattern **careless use of Static Variables** allows malicious code to exploit **covert channels between protection domains**

AP5: Non-Final Static Variables, Risk & Extent

- **Risk**
  - Static Variables that are loaded by the boot classloader (like the ones in **rt.jar**) or by the extension classloader are singleton objects in a JVM
  - **Non-final static String fields may transport serialized java objects** to protection domains that are not privileged to access them



Browser VM

Applet from site A

Serial obj.

Protection Domain A

Applet from site B

Serial obj.

Protection Domain B

org.apache.xalan.processor.XSLProcessorVersion.LANGUAGE

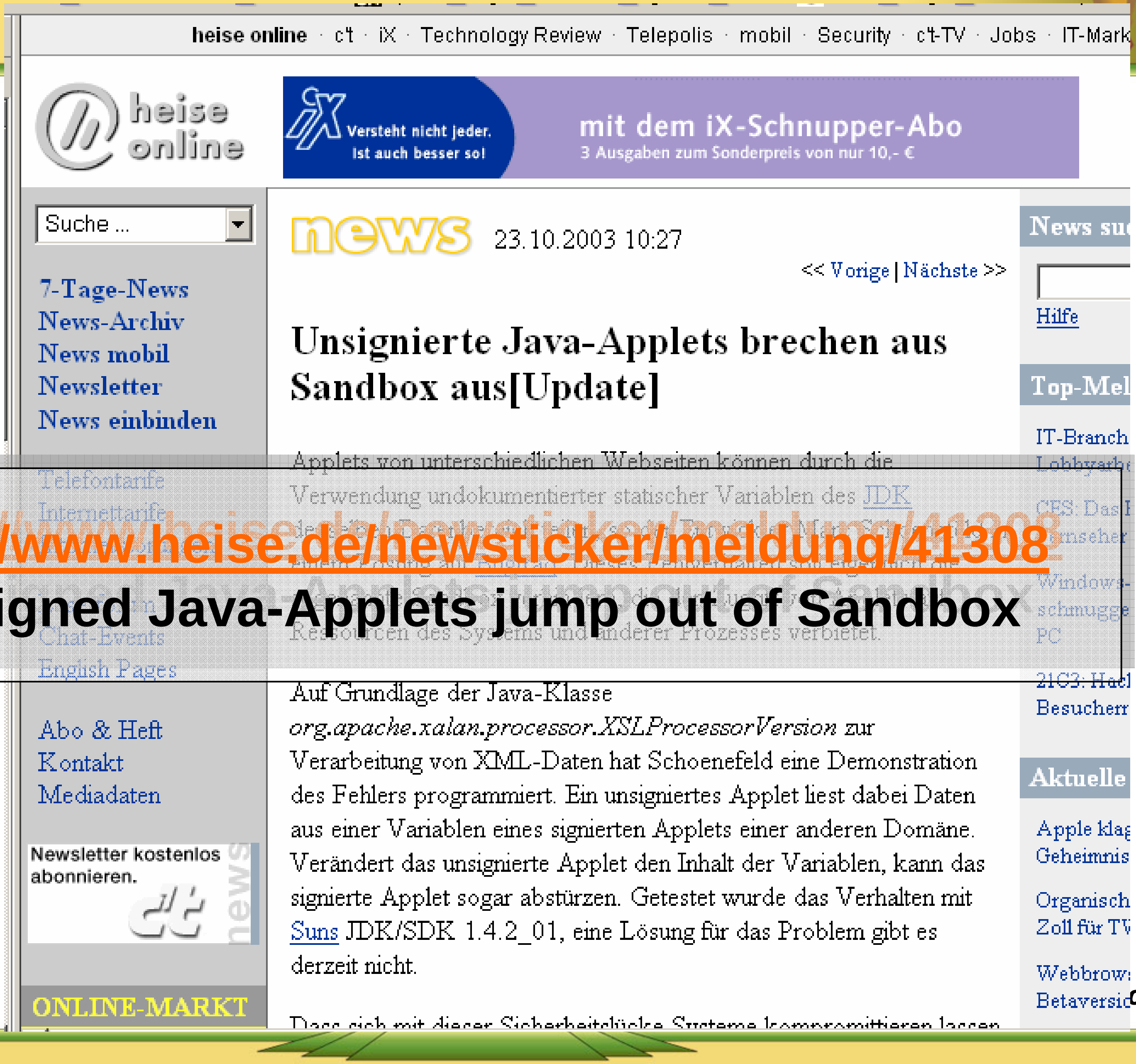
org.apache.xalan.processor.XSLProcessorVersion.S_VERSION

javax.swing.JDesktopPane.LIVE_DRAG_MODE

...

Protection Domain
/lib/rt.jar „AllPermissions“

AP5: Non-Final Static Variables, Risk & Extent



news 23.10.2003 10:27

<< Vorige | Nächste >>

Unsignierte Java-Applets brechen aus Sandbox aus[Update]

Applets von unterschiedlichen Webseiten können durch die Verwendung undokumentierter statischer Variablen des JDK [illegible] Ressourcen des Systems und anderer Prozesses verbietet.

Auf Grundlage der Java-Klasse *org.apache.xalan.processor.XSLProcessorVersion* zur Verarbeitung von XML-Daten hat Schoenefeld eine Demonstration des Fehlers programmiert. Ein unsigniertes Applet liest dabei Daten aus einer Variablen eines signierten Applets einer anderen Domäne. Verändert das unsignierte Applet den Inhalt der Variablen, kann das signierte Applet sogar abstürzen. Getestet wurde das Verhalten mit Suns JDK/SDK 1.4.2_01, eine Lösung für das Problem gibt es derzeit nicht.

Dass sich mit dieser Sicherheitslücke Systeme kompromittieren lassen

http://www.heise.de/newsticker/meldung/41308

Unsigned Java-Applets jump out of Sandbox

AP5: Non-Final Static Variables: Refactoring

Before (JDK1.42_04)	After (JDK1.42_05)

public class org.apache.xalan.processor.
XSLProcessorVersion {
public static final java.lang.String PRODUCT;
public static java.lang.String LANGUAGE;
public static int VERSION;
public static int RELEASE;
public static int MAINTENANCE;
public static int DEVELOPMENT;
public static java.lang.String S_VERSION;
}
	public class org.apache.xalan.processor.
	XSLProcessorVersion {
	public static final java.lang.String PRODUCT;
	public static **final** java.lang.String LANGUAGE;
	public static **final** int VERSION;
	public static **final** int RELEASE;
	public static **final** int MAINTENANCE;
	public static **final** int DEVELOPMENT;
	public static **final** java.lang.String S_VERSION;
	}

The final modifier prohibits modification of a variable after initial value was set.
Initially they only used it to protect their product name ☺

AP5: Non-Final Static Variables: How to audit ?

1. Via a built-in **findbugs** detector find candidate classes by searching for `public` classes
2. For these classes find
   I. Primitive Data fields and Strings are declared as `public static`, non-`final`
   II. Object Type Data fields, Arrays and Containers are declared as `public static`
   III. Methods that allow access on non-public instances of (I + II)

AP5: Conclusion and Suggestions

- **Conclusion**
  - Non-final static final fields allow to establish covert channels between protection domains and bypass restrictions such as the applet sandbox .

- **Suggestion** [http://java.sun.com/security/seccodeguide.html]
  - To the extent possible, refrain from using non-final public static variables because there is no way to check whether the code that changes such variables has appropriate permissions.
  - In general, be careful with any mutable static states that can cause unintended interactions between supposedly independent subsystems.

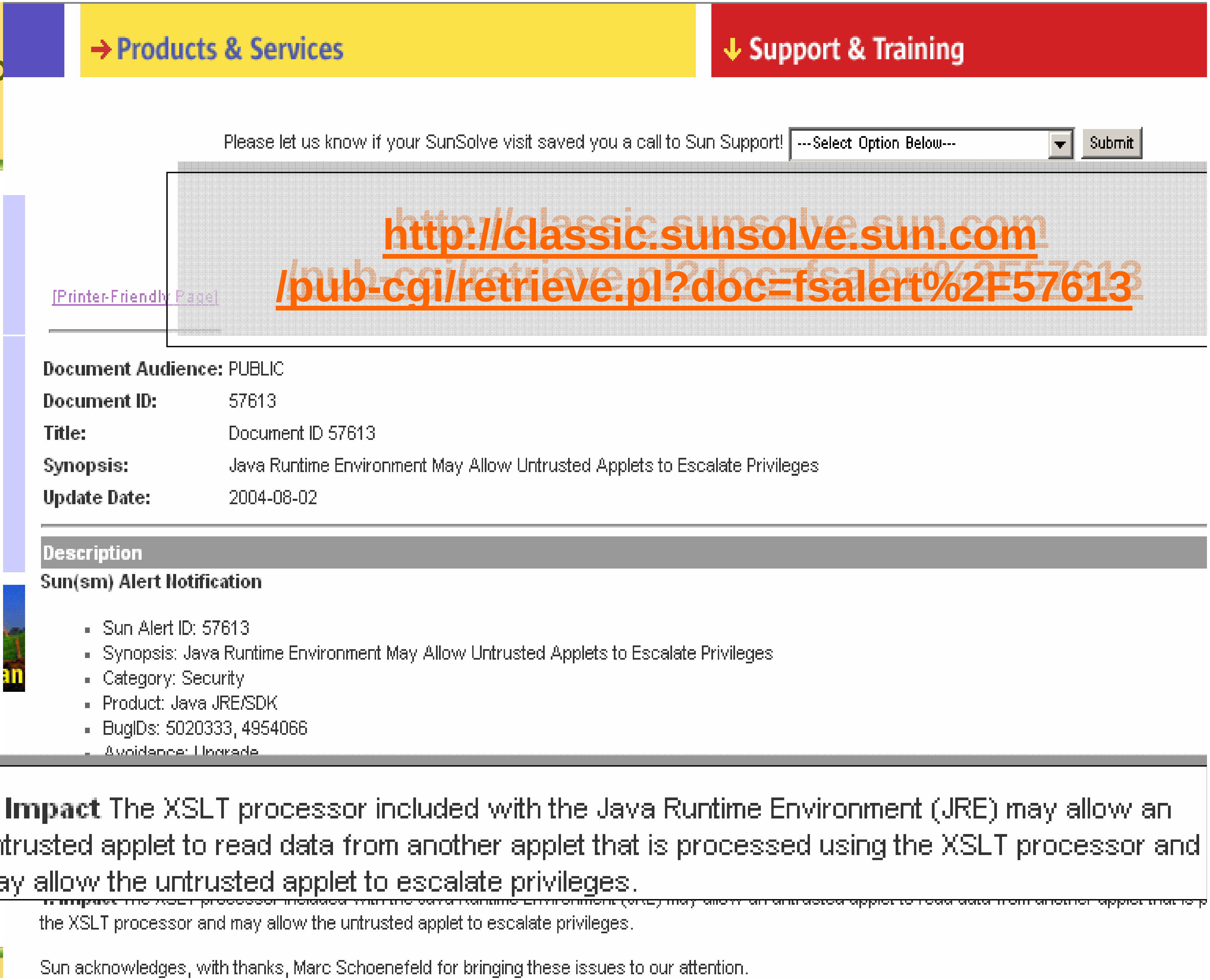
Antipattern 6: Insecure component reuse

- „Distributed component-structured applications can consist of software components which are supplied by different vendors. Therefore one has to distinguish between application owners and software component vendors and there is a needs for corresponding protection“: [Hermann, Krumm]
- 3rd – party components might be built with a functionality based programmer intend, whereas the control of the confined execution models of the JDK require a security based programmer intend.
- JDK as a component-structured middleware application uses a lot of XML functionality from the Apache foundation. Is there enough protection against vulnerabilities of these 3rd-party components embedded in JDK ?

AP6: Insecure component reuse, Risk & Extent

- **Risk**
  - The XSLT parser embedded in JDK is directly taken from a previous apache **XALAN** standalone version, downloadable from http://xml.apache.org
  - It is highly configurable, especially it allows to customize the functions that may be employed during **XSLT** (extensible stylesheet language transformations)
  - **Non-final static arrays in trusted libraries may contain objects that are allowed to process** data throughout the entire JVM
  - We will show that the Antipattern **insecure component reuse** allows malicious code to exploit **visibilities granted to trusted code by inserting malicious callbacks**

AP



Please let us know if your SunSolve visit saved you a call to Sun Support!

http://classic.sunsolve.sun.com /pub-cgi/retrieve.pl?doc=fsalert%2F57613

Document Audience:	PUBLIC
Document ID:	57613
Title:	Document ID 57613
Synopsis:	Java Runtime Environment May Allow Untrusted Applets to Escalate Privileges
Update Date:	2004-08-02

Description

Sun(sm) Alert Notification

- Sun Alert ID: 57613
- Synopsis: Java Runtime Environment May Allow Untrusted Applets to Escalate Privileges
- Category: Security
- Product: Java JRE/SDK
- BugIDs: 5020333, 4954066
- Avoidance: Upgrade

1. Impact The XSLT processor included with the Java Runtime Environment (JRE) may allow an untrusted applet to read data from another applet that is processed using the XSLT processor and may allow the untrusted applet to escalate privileges.

the XSLT processor and may allow the untrusted applet to escalate privileges.

Sun acknowledges, with thanks, Marc Schoenefeld for bringing these issues to our attention.

AP6: Insecure component reuse: Refactoring

Before (JDK1.42_05)	After (JDK1.42_06)

public class org.apache.xpath.compiler.FunctionTable {
public static org.apache.xpath.compiler.FuncLoader[] m_functions;
[...]
}
	public class org.apache.xpath.compiler.FunctionTable {
	private static org.apache.xpath.compiler.FuncLoader[] m_functions;
	[...]
	}

This refactoring is **adjusting** the enhanced functionality of the component to the level needed **for running the component securely** in confined execution models. Technically the refactoring cures an antipattern 4 and an antipattern 5.

The **private** modifier prohibits malicious code to modify the table consisting the built-in functions of the XSLT parser.

AP6: Insecure component reuse: How to audit ?

1. 3rd-party components may include all types of antipatterns, from our experience check at least for the antipatterns presented here
    1. Check for Integer Overflow
    2. Check for proper Serialisation, watch for side effects
    3. Check for defensive use of privileged code, especially when using privileged or “AllPermission” protection domains
    4. Adjust inappropriate scope to the level needed and add security checks to public available fields and functionality
    5. Close covert channels in static non-final fields and static mutable container types (also indirect uses)

AP6: Conclusion and Suggestions

- **Conclusion**
  - Even if your own code is secure, 3rd – party components may ruin your security concept
- **Suggestion**
  - Ask the vendor of the components you reuse , whether they check their components with findbugs or similar tools
  - Ask for a findbugs report before buying, this may increase your trust to component
  - A lot of open source projects already include such a report, but some closed source guys still have to learn

finally{}

Q & A

Contact	**Send me an eMail** marc@marc-schoenefeld.com
Detectors presented	**Download at** www.illegalaccess.org