Java/JEE Vulnerabilities explained

Marc Schönefeld

University of Bamberg

Confidence 09, Krakow, PL
Introduction

The Java-Architecture

Vulnerabilities

And now: Real-Life Vulnerabilities

Finalizing
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)

The diagram shows the cost per defect at different stages of the software development process: Requirements (Req's), Design, Coding, Testing, and Maintenance. The cost increases significantly from Requirements to Maintenance.
The Java-Architecture

Overview

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 dysfunctional software development 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 necessary. **Without this wisdom, AntiPatterns will continue to persist.**
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 Top 10

<table>
<thead>
<tr>
<th>Number</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Cross Site Scripting (XSS)</td>
</tr>
<tr>
<td>A2</td>
<td>Broken Access Control</td>
</tr>
<tr>
<td>A3</td>
<td>Malicious File Execution</td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object Reference</td>
</tr>
<tr>
<td>A5</td>
<td>Cross Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A6</td>
<td>Information Leakage and Improper Error Handling</td>
</tr>
<tr>
<td>A7</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A8</td>
<td>Insecure Cryptographic Storage</td>
</tr>
<tr>
<td>A9</td>
<td>Insecure Communications</td>
</tr>
<tr>
<td>A10</td>
<td>Failure to Restrict URL Access</td>
</tr>
</tbody>
</table>
# CWE Top 25, Pt 1, Insecure component interaction

<table>
<thead>
<tr>
<th>CWE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE-20</td>
<td>Improper Input Validation</td>
</tr>
<tr>
<td>CWE-116</td>
<td>Improper Encoding or Escaping of Output</td>
</tr>
<tr>
<td>CWE-89</td>
<td>Failure to Preserve SQL Query Structure (aka ’SQL Injection’)</td>
</tr>
<tr>
<td>CWE-79</td>
<td>Failure to Preserve Web Page Structure (aka ’Cross-site Scripting’)</td>
</tr>
<tr>
<td>CWE-78</td>
<td>Failure to Preserve OS Command Structure (aka ’OS Command Injection’)</td>
</tr>
<tr>
<td>CWE-319</td>
<td>Cleartext Transmission of Sensitive Information</td>
</tr>
<tr>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>CWE-362</td>
<td>Race Condition</td>
</tr>
<tr>
<td>CWE-209</td>
<td>Error Message Information Leak</td>
</tr>
</tbody>
</table>
## CWE Top 25, Pt2, Risky Usage of system resources

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

<table>
<thead>
<tr>
<th>Critical</th>
<th>Worm-Able Code, no or minor user-interaction required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important</td>
<td>Significant Loss of Availability, Integrity or Confidentiality</td>
</tr>
<tr>
<td>Moderate</td>
<td>Difficult Exploitation, Non-default settings required</td>
</tr>
<tr>
<td>Low</td>
<td>Low Probability, low effect</td>
</tr>
</tbody>
</table>

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)**

```java
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.
And now: Real-Life Vulnerabilities

CWE-682: Wrong Calculations

Refactoring: Integer-Overflow

java.util.zip.CRC32 in Java 1.4.1_02

```java
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.
And now: Real-Life Vulnerabilities

CWE-682: Wrong Calculations

History repeats itself: CVE-2009-0794

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

```java
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.length
    + " 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 (\texttt{writeObject})
- The object is transferred to the receiving JVM (File, Socket, JNDI, . . .)
- The receiving JVM reconstructs the object from the flattened representation (\texttt{readObject})
Serialisation in a JEE environment

1. Sender-JVM Marshalls Object to Bytes
2. Send Bytes in POST request (or RMI/JMS etc.)
3. Receiver-JVM demarshalls Bytes into Object
4. Object may propagate
And now: Real-Life Vulnerabilities

Example: CWE-20 Insufficient Input validation

Problems with Serialization

Typical serialization code, read Object from Socket

```java
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 ...
And now: Real-Life Vulnerabilities

Example: CWE-20 Insufficient Input validation

The bytecode shows it all

**Bytecode within the `run()` method**

```java
public void run();
0:   aload_0
1:   getfield #3; //Field ois:Ljava/io/ObjectInputStream;
4:   invokevirtual #7; //Method ObjectInputStream.readObject():Ljava/lang/Object;
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) |
| $t = 2$ | The server casts the object |
|        | A) Cast is valid: continue |
|        | B) 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 bytecode 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:

- `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

```java
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 generates strings 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.
And now: Real-Life Vulnerabilities

Example: CWE-20 Insufficient Input validation

Timing behavior

Timing behavior to create pattern objects in JDK 1.4/1.5

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()` method 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**

```java
/**
 * 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);
}
```
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
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](https://docs.oracle.com/javase/8/docs/api/java/awt/Conditional.html)) were referenced (Schönefeld, 2006b)

And now: Real-Life Vulnerabilities

Example: CWE-20 Insufficient Input validation

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 6f6e616c0014 6a61 .Conditional..ja
0000050: 76612e6176612e6177742e436f6e6469 va.awt.Condition
0000060: 6a6176612e6177742e436f6e646974696f6e al..java.awt.Condition
0000150: 616c0014 6a6176612e6177742e436f6e64 onal..java.awt.Con...
015ffe0: 6a6176612e6177742e436f6e646974696f6e java.awt.Condition
015fff0: 616c0014 6a6176612e6177742e436f6e64 onal..java.awt.Con...
0160000: 6f6e646974696f6e48616e646c65723b7870 ."Handler;xp
```

Marc Schöenefeld (University of Bamberg) Java Vulnerabilities explained Confidence 09, Krakow, PL 32 / 42
The generation program recreates the artificial object representation, which makes the native call fail to `defineClass` 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

```java
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

Propagations of an attack object

- **Attacker Called Servlet**
- **getMeAnObject**
- **read serialized Object**
  - **ObjectInputStream**
  - **readObject**
  - **GetObjectFromPostData**
  - **J2SE**
    - **vulnerable class**
    - **JVM.DLL**
    - **other.DLL**
  - **J2EE**
    - **JMX**
    - **JMXInvokerServlet**
    - **Servlet mapping**
    - **HTTP**
    - **Adapter**
    - **HTTPInvoker**
    - **Port 8080**
  - **OS**
  - **Crash/Excessive CPU usage**

Marc Schönefeld (University of Bamberg)
And now: Real-Life Vulnerabilities

CWE-250: unnecessary permissions

Antipattern: Code injection

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

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

Aliasing a JAVA function with SQL in database/script

```sql
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);
```

Marc Schönefeld (University of Bamberg) Java Vulnerabilities explained Confidence 09, Krakow, PL 36 / 42
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.
And now: Real-Life Vulnerabilities

CWE-732: Permissions and resources

Antipattern: Malicious usage of API functions

Misuse of font functions to DoS a system with an applet

```java
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;
    }
}

class FontCreatorFullDiskApplet extends Applet {
    static { try {
        byte abyte0[] = new byte[0];
        Font font = Font.createFont(0, new MIS());
    } catch(Exception exception) { }
    }
}
```
And now: Real-Life Vulnerabilities

CWE-732: Permissions and resources

Refactoring

Misuse of font functions to DoS a system with an applet

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

drwxrwxrwxrwt 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 familiar with the security concepts of the platform
- Try to learn from CVE descriptions
- Reconstruct exploit idea from inverting the patch concept
- Practice, practice, ...
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.

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.

Deutsch, L. Peter.

E. Bruneton, R. Lenglet, and T. Coupaye.
“ASM: a code manipulation tool to implement adaptable systems.”

Eckert, Claudia.
Sichere, verteilte Systeme – Konzepte, Modelle und Systemarchitekturen.
References

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.
URL http://citeseer.ist.psu.edu/gosling00java.html.

Hovemeyer, David, and William Pugh.
“Finding Bugs is Easy.”
In OOPSLA. 2004.

McGraw, Gary.
Software Security- Building Security In.
Addison-Wesley, 2006.

OpenJDK project.

Schönefeld, Marc.
“Hunting Flaws in JDK.”

_____.
“Pentesting J2EE.”

_____.
“Pentesting Java/J2EE, Finding remote holes.”
Kuala Lumpur, Malaysia: HackInTheBox, 2006b.

Thomas, Mark.