SER335 Engineering Secure Software Systems Lab 4

Objectives:

1. Conduct a penetration test using Metasploitable
2. Apply secure coding guidelines to identify security-related defects in source code.
3. Apply Static Analysis tools to identify vulnerabilities in software
4. Understanding runtime environment protections using Java's security model

Your submission should be a zipfile with the component activity submission files. Name this <asurite>_lab4.zip

Background Setup: Metasploitable and Kali

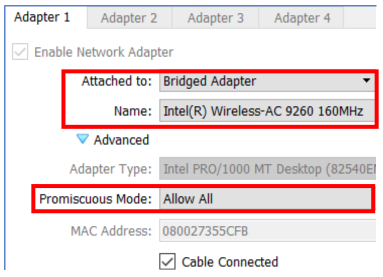
For this activity you will conduct a penetration test using the Metasploitable toolkit and Kali. There are 2 ways to run these frameworks, one as VMs under VirtualBox, the other as docker containers. Keep in mind that Metasploitable is an intentionally vulnerable VM/container! You do not want to run this on any exposed machines/ports!

Running Metasploitable and Kali as Virtual Machines under VirtualBox:

We provide a script below with an accompanying walkthrough video from Dr. Findler posted on Canvas. This takes you through the setup right until the point of conducting the specific attack. Download metasploitable from here: link

Environment Setup for VMs:

1. Setup Network using Bridge Adapter for Kali Linux (Time code 1:10) and Metasploitable.
2. Login to the Kali Linux system with your login credentials and launch the terminal emulator.
3. Switch to superuser using the sudo command:
   Command: *sudo su* (enter the password for Kali Linux)
4. Login to the Metasploitable system using *msfadmin/msfadmin*.
5. Check current network settings of kali (Time code 7.07) and Metasploitable machine using the ifconfig command.
   Command: *ifconfig*



Enable Network Adapter ✓
Attached to: Bridged Adapter
Name: Intel(R) Wireless-AC 9260 160MHz
Adapter Type: Intel PRO/1000 MT Desktop (82540E
Promiscuous Mode: Allow All
MAC Address: 080027355CFB
Cable Connected ✓

Both systems should be on the same network i.e. both have netmask *255.255.255.0* and first, three numbers of the IP address should be the same.

Running Metasploitable and Kali as docker containers (either directly on your host or within the SEED VM):

The setup for Kali and Metasploitable I got from this post (don't worry about parrot only metasploitable) and this YouTube video. I validated these steps on my Windows 10 laptop running Docker Desktop. The respective images you can find here: Kali and Metasploitable.

1. Start up docker.
2. Create a network named "vulnerable" (or whatever you want):

```
$ docker network create vulnerable --attachable --subnet 10.0.0.0/24
```

3. Pull the metasploitable image and then run it interactively

```
$ docker pull tleemcjr/metasploitable2

$ docker run -it --network vulnerable --ip="10.0.0.3" --name metasploitable --
hostname metasploitable2 tleemcjr/metasploitable2 bash
```

4. Inside this metasploitable terminal, start the vulnerable services:

```
root@metasploitable2:/# services.sh
```

5. Check the metasploitable network settings: root@metasploitable2:/# *ifconfig* (should say 10.0.0.3)
6. Pull the kali image and run it:

```
$ docker pull kalilinux/kali-rolling

$ docker run -it --network vulnerable --ip="10.0.0.2" --name kali --hostname kali
kalilinux/kali-rolling bash
```

`$ ipconfig` (should say 10.0.0.2)

7. Kali does not come pre-installed with nmap or the msfconsole like the VM does, we have to install them

```
┌──(root㉿kali)-[/]
└─# apt update

┌──(root㉿kali)-[/]
└─# apt install nmap

┌──(root㉿kali)-[/]
└─# apt install metasploit-framework
```

(note this says metasploit not metasploitable on purpose)

Activity 1: Conduct a Penetration Test Using Metasploitable (25 points)

For this activity your submission is a PDF file with the requested commands you execute and screenshots demonstrating the request steps of the attack. Specifically:

1. Under Penetration Testing below in the script, step #2, *give the command you used to determine what ports are open for potential exploits on the target machine, and provide a screenshot of the result of the command*.
2. This step starts the attack (at the end of the script in orange). Do the following:
   a. Set the RHOSTS and verify it has been set using "show info" and *capture a screenshot of the result*
   b. Execute the attack itself (exploit). (no screenshot required)
   c. Do a "`whoami`" command once you do (*give a screenshot*)
   d. Create a new user named "ser335hacker" using the Unix *`adduser`* command (*give the command*)
   e. Show the contents of the /etc/passwd file on the metasploitable VM/container using the cat command: `cat /etc/passwd`

Name your submission file for this activity "activity1.pdf", and ensure it is part of the overall submission.

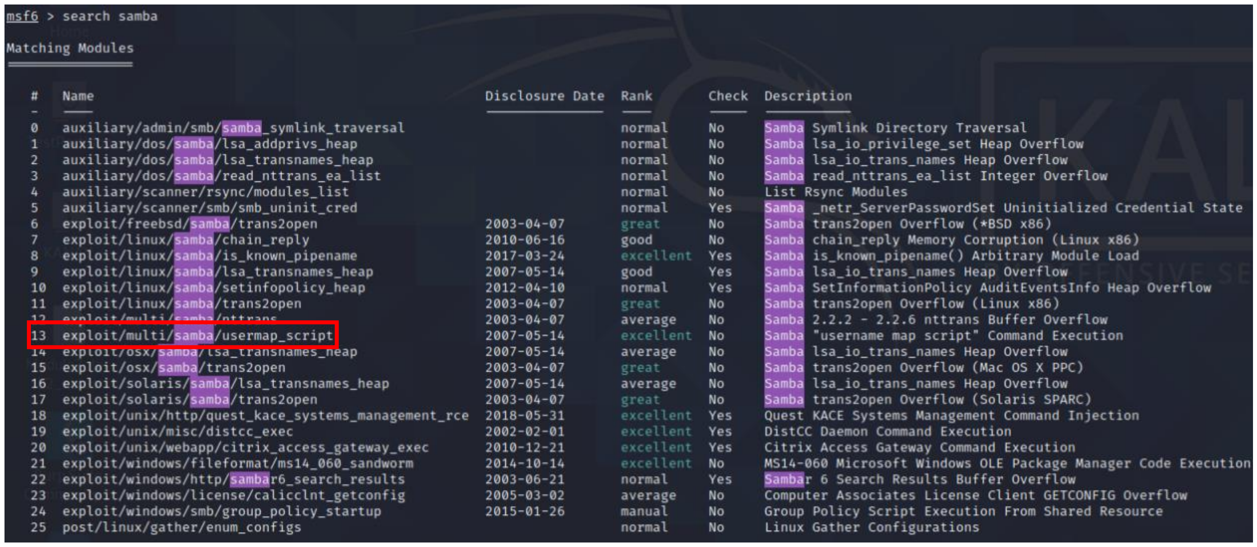
Activity Script:

Penetration Testing:

1. Open a kali terminal with superuser access and run the *msfconsole* command to open the Metasploit console.
2. Check for open ports in the Metasploitable system and find the Samba client-server port. You need to determine the command to use (see specification #1 above).

Preparing to attack Samba

a) Let's check what all attacks we can do on Samba. Search Metasploit app for Samba exploits using the command: *search Samba*

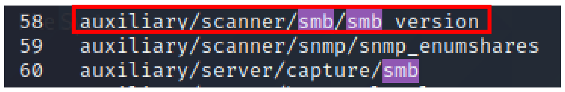


```
msf6 > search samba

Matching Modules
================

   #   Name                                                 Disclosure Date  Rank       Check  Description
   -   ----                                                 ---------------  ----       -----  -----------
   0   auxiliary/admin/smb/samba_symlink_traversal                           normal     No     Samba Symlink Directory Traversal
   1   auxiliary/dos/samba/lsa_addprivs_heap                                 normal     No     Samba lsa_io_privilege_set Heap Overflow
   2   auxiliary/dos/samba/lsa_transnames_heap                               normal     No     Samba lsa_io_trans_names Heap Overflow
   3   auxiliary/dos/samba/read_nttrans_ea_list                              normal     No     Samba read_nttrans_ea_list Integer Overflow
   4   auxiliary/scanner/rsync/modules_list                                  normal     No     List Rsync Modules
   5   auxiliary/scanner/smb/smb_uninit_cred                                 normal     Yes    Samba _netr_ServerPasswordSet Uninitialized Credential State
   6   exploit/freebsd/samba/trans2open                     2003-04-07       great      No     Samba trans2open Overflow (*BSD x86)
   7   exploit/linux/samba/chain_reply                      2010-06-16       good       No     Samba chain_reply Memory Corruption (Linux x86)
   8   exploit/linux/samba/is_known_pipename                2017-03-24       excellent  Yes    Samba is_known_pipename() Arbitrary Module Load
   9   exploit/linux/samba/lsa_transnames_heap              2007-05-14       good       Yes    Samba lsa_io_trans_names Heap Overflow
   10  exploit/linux/samba/setinfopolicy_heap               2012-04-10       normal     Yes    Samba SetInformationPolicy AuditEventsInfo Heap Overflow
   11  exploit/linux/samba/trans2open                       2003-04-07       great      No     Samba trans2open Overflow (Linux x86)
   12  exploit/multi/samba/nttrans                          2003-04-07       average    No     Samba 2.2.2 - 2.2.6 nttrans Buffer Overflow
   13  exploit/multi/samba/usermap_script                   2007-05-14       excellent  No     Samba "username map script" Command Execution
   14  exploit/osx/samba/lsa_transnames_heap                2007-05-14       average    No     Samba lsa_io_trans_names Heap Overflow
   15  exploit/osx/samba/trans2open                         2003-04-07       great      No     Samba trans2open Overflow (Mac OS X PPC)
   16  exploit/solaris/samba/lsa_transnames_heap            2007-05-14       average    No     Samba lsa_io_trans_names Heap Overflow
   17  exploit/solaris/samba/trans2open                     2003-04-07       great      No     Samba trans2open Overflow (Solaris SPARC)
   18  exploit/unix/http/quest_kace_systems_management_rce  2018-05-31       excellent  Yes    Quest KACE Systems Management Command Injection
   19  exploit/unix/misc/distcc_exec                        2002-02-01       excellent  Yes    DistCC Daemon Command Execution
   20  exploit/unix/webapp/citrix_access_gateway_exec       2010-12-21       excellent  Yes    Citrix Access Gateway Command Execution
   21  exploit/windows/fileformat/ms14_060_sandworm         2014-10-14       excellent  No     MS14-060 Microsoft Windows OLE Package Manager Code Execution
   22  exploit/windows/http/sambar6_search_results          2003-06-21       normal     Yes    Sambar 6 Search Results Buffer Overflow
   23  exploit/windows/license/calicclnt_getconfig          2005-03-02       average    No     Computer Associates License Client GETCONFIG Overflow
   24  exploit/windows/smb/group_policy_startup             2015-01-26       manual     No     Group Policy Script Execution From Shared Resource
   25  post/linux/gather/enum_configs                                        normal     No     Linux Gather Configurations
```

There are many exploits that can be performed on Samba. We will use the *usermap_script* as our exploit. It allows us to log into the target system as admin with admin privileges.

b) Follow the below steps to determine the version of Samba that the host system is using

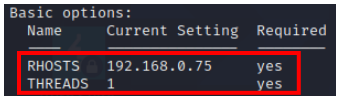
   i) Search for smb options
      Command: *search smb*



```
58   auxiliary/scanner/smb/smb_version
59   auxiliary/scanner/snmp/snmp_enumshares
60   auxiliary/server/capture/smb
```

   ii) Use the 'auxiliary/scanner/smb/smb_version' option.
      Command: *use auxiliary/scanner/smb/smb_version*

   iii) Set the Remote Host RHOST to the target IP address.
      Command: *set RHOSTS 192.168.0.75 (IP address of Metasploitable)*
      // Note this is 10.0.0.3 if using docker



```
Basic options:
  Name     Current Setting  Required
  ----     ---------------  --------
  RHOSTS   192.168.0.75     yes
  THREADS  1                yes
```

   iv) Use *'show info'* to verify RHOST is set correctly.

c) Type in '*run'* to find out the version of Samba on the target system. Note the version does fall within the usermap_script exploit vulnerable versions.

d) To leave the smb_version command you can type in the *'exit'* command.

3. Samba Attack!!
   a) Open a *msfconsole* and type in '*search samba'* to find the desired exploit.
   b) Type in *use exploit/multi/samba/usermap_script*

When you get to this point you need to complete the exploit specified in the above at the top of the activity starting with step 2.

Submission: At the top of the activity for steps 1 and 2 I highlighted in yellow italics the items we need for your Activity 1 submission. Please copy these items into a file named lab4_activity1.docx. Also, we recognize there can be some sensitivity in setting up the VMs or containers for this activity; if you feel there is something you need to tell us regarding your lab setup, please put it at the top of this document in a section labelled "Readme".

Activity 2: Secure Code Reviewing Using Coding Guidelines (30 points)

Overview:
In this task, you will review 3 code examples: two in Java and one in C. Using the SEI CERT secure coding guidelines, search for a related topic in the code guideline and implement a compliant version of the code so that it meets best practice standards. No compiler is necessary for this exercise, but you may choose to perform the code modifications and execute the programs in a compilation environment to ensure they work. 10 points each.

Your submission will be 3 revised code snippets and an explanation of the changes made with direct references to the SEI CERT coding guidelines. Please include your code revisions in a zipfile named lab4_activity2.zip to be included with the overall assignment submission. Note these are snippets so by itself the code won't compile or run, though you may consider writing a main program to test it out (we don't require it, but if we think your code won't run that is what we will do to verify it!).

Task 1: Consider the following Java code example:

```
public class EqualityTest {
  public static void main(String[] args) {
     char[] chars1 = {'a', 'b', 'a'};
     char[] chars2 = {'a', 'b', 'a'};
     System.out.println(chars1.equals(chars2)); // Prints false
  }
}
```

This code violates best practices for how to do certain Expressions (EXP) in Java. Use the CERT Secure Coding Guidelines for Java and find a corresponding best practice rule that addresses how to correct or avoid such a problem. Indicate the recommended mitigation approach, and rewrite the code so that it is compliant.

Task 2: Consider the following Java code example:

```
public class FileTest {
  public static void main(String[] args) {
    showFile(args);
  }
  public static void showFile(String[] args) throws IOException {
    try {
        FileInputStream inputFileStream = new FileInputStream(args[0]);
        readAndShowFile(inputFileStream); // not shown; assume it displays file and closes stream
    } catch (FileNotFoundException exc) {
       System.out.println("No such file: " + args[0] + ", exception " + exc);
    }
  }
}
```

This code violates best practices for how to do handle Input and Output (FIO) in Java. Use the CERT Secure Coding Guidelines for Java and find a corresponding best practice rule that addresses how to correct or avoid such a problem. Indicate the recommended mitigation approach, and rewrite the code so that it is compliant.

Task 3: Consider the following C code example, which does a simple math operation. In fact, math operations are some of the most common expressions found in programs, and most program code can easily generate exceptional conditions that a programmer may not easily envision.

```
#include <stdio.h>

void process(unsigned int x, unsigned int y) {
  unsigned int sum = x + y;
  return sum;
}
int main() {
   unsigned int a = 40918;
   unsigned int b = 21293;
   unsigned int c = process(a,b);
   printf("Sum of %u and %u: %u", a, b, c); // Depends...
   return 0;
}
```

This code violates best practices for how to process integers in C. Use the CERT Secure Coding Guidelines for C and find a corresponding best practice rule that addresses how to correct or avoid such a problem. Indicate the vulnerability and recommended mitigation approach, and rewrite the code so that it is compliant.

Activity 3: Static Analysis for Security (SAST) (45 points)

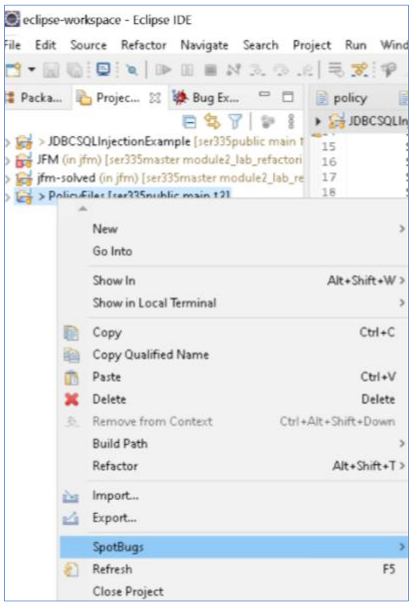
Overview

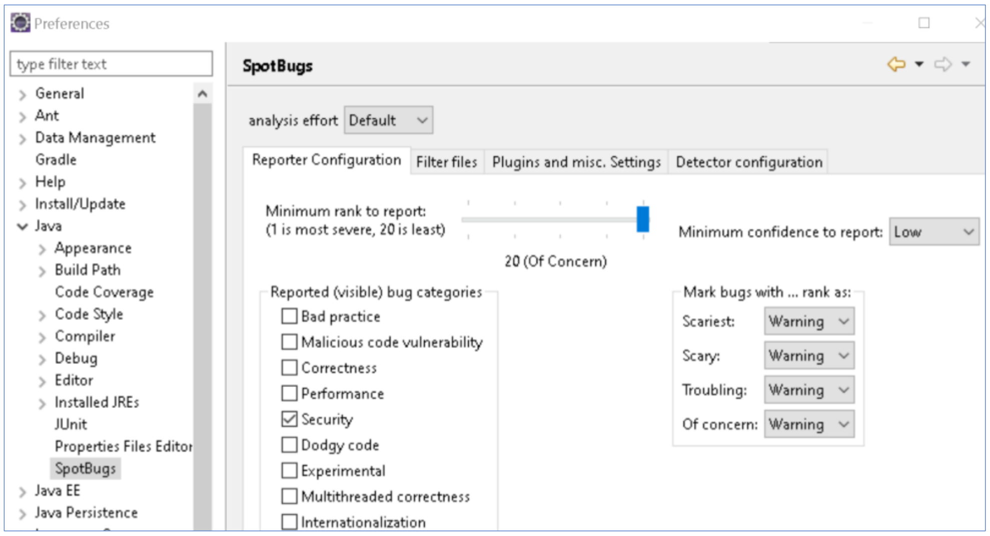
Static analysis tools for security (SAST) probe pre-compiled or compiled source code for security-related defects. There are a variety of categories of such tools, one of which you will experiment with for this lab activity. *SpotBugs* is the successor to the popular *FindBugs* from the University of Maryland, and includes a category of heuristic rules for security. Further, SpotBugs supports its own plugins, one of which we will use called find-sec-bugs. Then, we will examine ways to resolve the most severe security-related defects through code and Java's security manager.

Static Analysis Using SpotBugs:

Environment Setup

- This lab requires use of the Eclipse IDE.
- In Eclipse, go to the Eclipse Marketplace (under the Help menu). Search for "SpotBugs". When it is found please install the plugin.
- You should restart Eclipse. Before going any further, check that SpotBugs was properly installed:
  - Load any Java project (any code you have, or start a new one, or start from the given code).
  - Right-click on the project in the Package or Project Explorer views in the left pane of the Java perspective in Eclipse. You should see a context menu that says SpotBugs (see figure to the right)
  - You should also be able to go to *Preferences* (under the Windows menu on Windows, or the Eclipse menu on Mac) and under "Java" see "SpotBugs"
  - Make sure only "Security" is checked under bug categories, and the minimum rank is set high (20) and confidence is set to Low. This enables the broadest scope of catching security-related defects. See image below:


eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Wind
Packa... Projec... Bug Ex... policy
JDBCSQLInjectionExample [ser335public main
JFM (in jfm) [ser335master module2_lab_refactori
jfm-solved (in jfm) [ser335master module2_lab_re
New
Go Into
Show In Alt+Shift+W
Show in Local Terminal
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Refactor Alt+Shift+T
Import...
Export...
SpotBugs
Refresh F5
Close Project



SpotBugs

analysis effort: Default

Minimum rank to report: (1 is most severe, 20 is least) 20 (Of Concern)

Minimum confidence to report: Low

Reported (visible) bug categories
- [ ] Bad practice
- [ ] Malicious code vulnerability
- [ ] Correctness
- [ ] Performance
- [x] Security
- [ ] Dodgy code
- [ ] Experimental
- [ ] Multithreaded correctness
- [ ] Internationalization

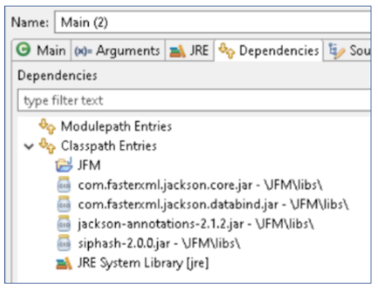
Mark bugs with ... rank as:

Scariest:	Warning
Scary:	Warning
Troubling:	Warning
Of concern:	Warning

- Next, download find-sec-bugs from here: https://find-sec-bugs.github.io/
  - There are installation instructions for Eclipse that I followed with some slight modifications. I've included what you need to do here, it is straightforward.
  - It says it supports IntelliJ, but this is out-of-date. Unfortunately, recent versions of IntelliJ are not compatible with this plugin so you are stuck using Eclipse ☹
    - First, download the find-sec-bugs jarfile and put it in a safe spot on your local hard drive.
    - Next, return to the SpotBugs configuration screen as shown above. Note the *Plugins & misc. Settings* tab.
      - On that tab is an option to "Add" on the right. Add your find-sec-bugs jarfile and restart Eclipse.

TASK: Find security-related defects using SpotBugs and find-sec-bugs (15 points)

Using the configuration you just setup, create a new Java project (if you have not already), using the given code for the lab. You should recognize it – it is the same JFM program given to you in lab 2!



Name: Main (2)

Main | Arguments | JRE | Dependencies | Sou

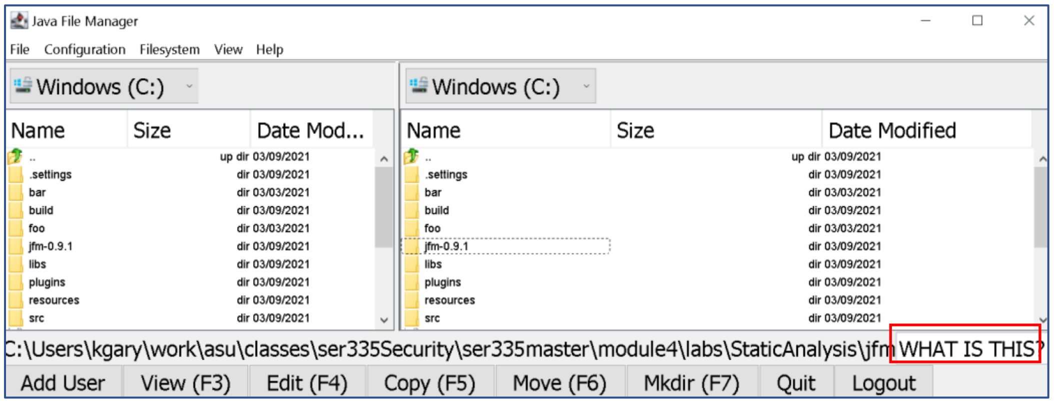
Dependencies

type filter text

```
Modulepath Entries
Classpath Entries
    JFM
    com.fasterxml.jackson.core.jar - \JFM\libs\
    com.fasterxml.jackson.databind.jar - \JFM\libs\
    jackson-annotations-2.1.2.jar - \JFM\libs\
    siphash-2.0.0.jar - \JFM\libs\
    JRE System Library [jre]
```

- Make sure you can run this code within Eclipse as opposed to using ant and the command-line as in Lab 2.
  - Right-click on the project and select *Run As… Java Application*
  - This should work, but if it does not:
    - Check your Run Configuration (available under the *Run As*… context menu)
    - It should show a main class of org.jfm.main.Main on the Main tab
    - It should show the jar files in the lib directory under the Dependencies tab

Depending on how observant you were when you ran this code for lab 2, you may or may not have noticed the unlabeled TextField in the lower right corner of the main UI. ***WHAT IS THIS?***



Java File Manager

File Configuration Filesystem View Help

Windows (C:) | Windows (C:)

```
Name         Size       Date Mod...    | Name           Size          Date Modified
..                up dir 03/09/2021    | ..                    up dir 03/09/2021
.settings            dir 03/09/2021    | .settings                    dir 03/09/2021
bar                  dir 03/03/2021    | bar                          dir 03/03/2021
build                dir 03/09/2021    | build                        dir 03/09/2021
foo                  dir 03/03/2021    | foo                          dir 03/03/2021
jfm-0.9.1            dir 03/09/2021    | jfm-0.9.1                    dir 03/09/2021
libs                 dir 03/09/2021    | libs                         dir 03/09/2021
plugins              dir 03/09/2021    | plugins                      dir 03/09/2021
resources            dir 03/09/2021    | resources                    dir 03/09/2021
src                  dir 03/09/2021    | src                          dir 03/09/2021
```

C:\Users\kgary\work\asu\classes\ser335Security\ser335master\module4\labs\StaticAnalysis\jfm WHAT IS THIS?

Add User | View (F3) | Edit (F4) | Copy (F5) | Move (F6) | Mkdir (F7) | Quit | Logout

Let's see. Type "notepad" into it if you are on Windows, or /System/Applications/TextEdit.app/Contents/MacOS/TextEdit if you are on a Mac. What happens? Should it happen?

Task 3.1: Provide a screenshot of JFM running with the respective command and what shows on the screen.

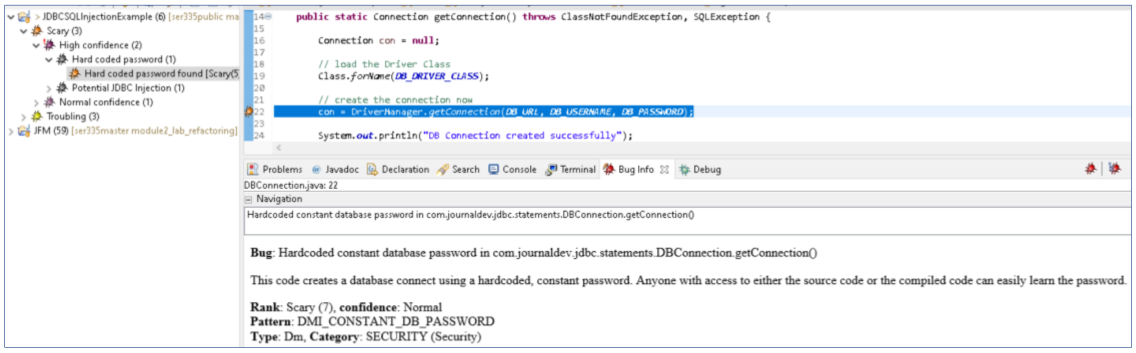
Let's have SpotBugs help us find the problem. Close the application, return to Eclipse, and run SpotBugs → Findbugs from the context menu when you right-click on the project. This will run SpotBugs in the background (you can see it processing in the status bar in the lower right). It doesn't take too long, wait about 30 seconds to be sure.

- The BugExplorer should come up when it is done. If it does not come up automatically, then navigate to the Window → Show View → Bug Explorer choice on the Eclipse menu. Record the following for your submission:
  - How many total security defects does SpotBugs report (again, make sure you previously configured SpotBugs with find-sec-bugs correctly using the exact select options from above)?
  - What are the different categories of severity? How many types of *confidence* are reported for each? Under each confidence level what types of defects are reported.
  - You may provide a screenshot of the expanded menu under the BugExplorer.

Task 3.2: Record and provide a screenshot of the BugExplorer in Eclipse reporting categories and confidences of security defects in the JFM project.

- Highlight the defect with High Confidence. Expand the report to find where the defect is in the code, and double-click on the item to have Eclipse jump to that file and line number in the code. Further, right-click on the defect and select *Show Bug Info* from the context menu. This should display an informative window of information about the nature of the defect.
  - Provide a screenshot of your full Eclipse window. It should show the expanded Bug Explorer on the left, the code window with the highlighted line of code where the security defect is, and the Bug Info in the bottom pane. The image below shows this for a different example.
  - SpotBugs/find-sec-bugs will also, if available, connect the nature of the defect to OWASP and/or the Common Weaknesses Enumeration (CWE)> for the defect in question, what is the CWE entry? Use the information in the links provided to describe the defect and why it is *Scary* and reported with *High Confidence*.

Task 3.3: Provide a screenshot of Eclipse reporting on the High Confidence defect in the JFM code, and describe the defect using the resources provided.



```java
    public static Connection getConnection() throws ClassNotFoundException, SQLException {

        Connection con = null;

        // load the Driver Class
        Class.forName(DB_DRIVER_CLASS);

        // create the connection now
        con = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);

        System.out.println("DB Connection created successfully");
```

DBConnection.java: 22

Hardcoded constant database password in com.journaldev.jdbc.statements.DBConnection.getConnection()

Bug: Hardcoded constant database password in com.journaldev.jdbc.statements.DBConnection.getConnection()

This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.

Rank: Scary (7), **confidence**: Normal
Pattern: DMI_CONSTANT_DB_PASSWORD
Type: Dm, **Category**: SECURITY (Security)

This is an example screenshot for the SQL Injection sample code in the class public GitHub repository. Your screenshot for the task should look somewhat like this.

Task 3.4: Refactor the Java code to create a new & improved JFMSecure! (30 points):
Refactor the JFM code to address the High Confidence vulnerability using a more secure approach with 2 techniques:

1. *(10) Use the latest recommended facilities provided by the language*. The High Confidence defect focuses on a specific way of doing a specific thing in Java. But since Java 5 there has been a better way of accomplishing the same functionality using *ProcessBuilder*. Refactor this code to use this functionality.
2. *(20) Use whitelisting to specify exactly what is allowed*. Whitelisting is a pessimistic technique in that by default it denies everything except for those things explicitly allowed. Apply whitelisting to resolve this vulnerability by reading a whitelist.json file that has in it the roles and the set of whitelist terms allowed for that role. This file should be located with the other json files, and have a similar structure to authorization.json:

```
[
        {
                "role": "admin",
                "whitelist": ["TERM1", "TERM2", …]
        },
        {
                "role": "developer",
                "whitelist": ["TERM3", "TERM4", …]
        },
        {
                "role": "user",
                "whitelist": ["TERM5", "TERM6", …]
        }
]
```

When you understand the vulnerability then you will understand what TERMX should be.

NOTE: This is asking you to implement whitelist functionality in the JFM code, NOT to just provide a whitelist.json.

Submission for Activity 3: This activity asks for screenshots, explanations and code. Paste the screenshots into a Word doc named lab4_activity3.docx with whatever text you feel is necessary (label the screenshots please using the numbers in the callout boxes above). Provide the modified source tree (the source files and any supporting files, like the json files) that we should be able to build and run your code ourselves (please do not include compiled code and libraries, or the entire IDE project as this makes your submission too large) in lab4_activity3.zip. Both files go into the overall submission zipfile.

As mentioned at the top of the lab, your overall submission for the assignment should be a zipfile that includes the individual documents and zipfiles from the respective individual activities.