SER335 Lab 2 ONLINE: **Host and Network Security Lab**

Objectives:

- Understand authentication and implement password security using password salts.
- Understand authorization and implement Role-Based Access Control (RBAC) into the application**.**
- Understand how sniffing and spoofing exploits work and how they might be defensed.

OVERALL SUBMISSION

- Assemble your submission files for each part together in one zipfile named <asurite>_lab2.zip
- Submission instructions for each part are given after each task. You will submit one zipfile to Canvas.

Part I: Host Security (50 points)

AUTHENTICATION

In Part I we focus on host security, particularly the role of the host in provide filesystem protections. In part I you will modify an existing Java program to control authentication & authorization for filesystem access. You are provided with starting code and some information below to get you started.

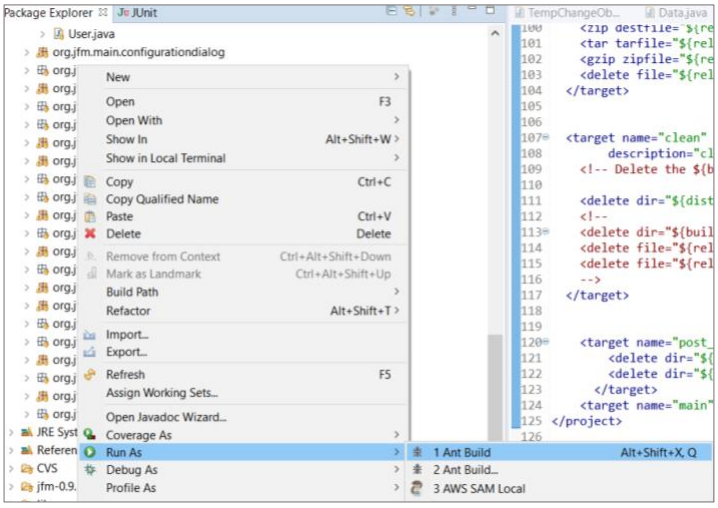
Run the jfm project:

There are two ways to run the project inside IDE:
1) Creating a jar file using Ant build
2) Run as a Java Application.

Creating and running an executable jar file using Ant:

Apache ant is a general-purpose build tool primarily used to build and deploy java projects. Ant uses build.xml to configure the build process. The build.xml file for our project is located under the '*jfm*' directory. The build.xml file has tasks to:

- *compile:* Compile the java code inside the '*src'* folder and copy the compiled code to a build directory
- *copy-dependencies:* Group all dependencies (libs) into a big *dependency-all.jar* file
- *dist:* Use compiled code & *dependency-all.jar* to create *jfm.jar and run.[bat|sh]* executables inside '*jfm-0.9.1*' directory
- *clean and post_clean:* Clean all the temporary folders.



To execute the build file, right-click on *build.xml, select RunAs* and click on the first *Ant build* option. The build file creates a new *jfm-0.9.1* folder and adds the executables (jfm.jar, run.sh, run.bat) to it. Double-click on the appropriate executable to run the application, or right-click and "Run As" → Java application in your IDE, or you may also use ant at the terminal command-line (ant dist), cd to jfm-0.9.1, and run (run.bat or run.sh). *Please understand how this code builds and runs and understand the paths!*

Background for Part 1

We will implement Authentication & Authorization features for a file management tool JFM (Java File Manager). Users must be authenticated to use the tool, and based on a role, given various file permissions. You will implement salted passwords for authentication and use an external configuration file to implement Role-Based Access Control (RBAC). The given code has 3 files: authentication.json, authorization.json, and salts.json located inside the '*Resources'* folder. The files store the following data:

salts.json - stores list of users and their salts(keys).
authentication.json - Stores list of users, their salted passwords, and assigned roles.
authorization.json - Stores list of roles and privileges associated with each role.

In the Java code, these files are loaded by 3 Singleton pattern objects under the edu.asu.ser335.jfm package. The org.jfm.Main main program instantiates these Singletons to force-load these files.

UsersSingleton: Wraps hashtables that store users with their respective passwords and users with their respective roles.
RolesSingleton: Wraps a hashtable that stores roles and privileges associated with each role.
SaltsSingleton: Wraps a hashtable that stores users and a salt associated with each user.

A login event (user login) is captured by "actionPerformed" method in org.jfm.main.LoginPannel.java. This panel is the 1st thing you see when the application starts. In the event listener, a user is validated by *validateUser()*, and on successful validation the main application is initialized (*MainFrame frame = new MainFrame()*). In the given code *validateUser()* is hardcoded to return true no matter what (so the user is always successfully logged in under any selected role!).

Task H1: Implement login functionality (18 points)

Write your code inside the *validateUser()* method in the *LoginPannel.java* file. Implement the following functionality:

1. (3) For each login event, check that the user, role, and user-role mapping exists in the *userRoleMapping* table from UsersSingleton.
2. (3) If the user, role, or user-role mapping does not exist, then the method will return a '*false*'.
3. (9) If the user, role, and user-role mapping exists:
   a. Generate a salted password using the '*siphash*' library.
   b. Verify the generated salt password against the salted password stored in the appropriate Singleton. A user should be successfully validated only if their username, salted password, and role match. The function will return '*true*' on successful validation.

Siphash-2.0.0.jar is available in 'libs' folder

The following login credentials (which use the salts) can be used to validate the user:

User:	adam	Role: admin	Password: 123456
User:	zampa	Role: developer	Password: 123456
User:	tim	Role: user	Password: 123456

AUTHORIZATION (Role-Based Access Control)

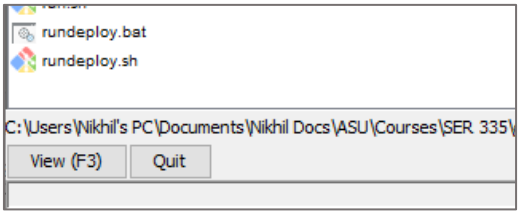
Background for this Task

On successful login *MainFrame()* (in org.jfm.main.MainFrame.java) method is invoked which calls the *MainPanel()* constructor (MainPanel.java). Furthermore, the MainPanel.java file contains code to display various action buttons like copy, mkdir, move a file or directory, etc. The buttons are added to the panel in '**buttons.addElement**' statements.

```
buttons.addElement(f5Button); //Copy
buttons.addElement(f6Button); //Move
buttons.addElement(f7Button); //Mkdir
```

Task H2: Implement RBAC for the JFM tool (15 points)

A user should perform only those actions permitted to his/her role. The activity button should only be added to the panel(displayed) if their role has the privileges. For example, in the authorization.json file if the admin role has '*view*' and '*quit*' privileges, then the application should display only those two buttons. You can view the privileges granted to a role in the authorization.json file.



Steps to follow:

1) Get all the privileges assigned for the *userRole* from the proper *Singleton(s)*. (3 points)
2) Based on privileges display the actions. (12 points)

Note: Partial code has been implemented for this task for the "Add User" button for admin to ease your understanding. Check out how the '*addNewUser*' action has been implemented in *MainPanel.java*

Background for task H3:

In JFM only an admin can add a new user. When you click on the '*adduser*' button, an admin panel pops up and has fields to enter a new username, password and select a role. Once the information is filled and submitted, the application (**AddUserPanel.java**) checks if the user already exists. If the user exits, the application notifies the user about it. If the user does not exist, a new salted password is created, and user details are added via the Singletons.

Task H3: Implement change password functionality in the JFM application. (17 points)

Only a user with role admin can change the user's password.

1. You need to add a button (called "changePwdButton") to the MainPanel. This button should only be visible to admin users, so append the keyword '*ChangePassword*' to the admin role in the authorization.json file
2. When that button is clicked, the event should be handled in *ChangePasswordAction*
3. *ChangePasswordAction* should show a new dialog box, *ChangePasswordPannel*
4. When the dialog box is filled out (username, password, role) and the *Submit* button clicked, the action event listener should start the Change Password functionality (see the TODO comment):
    a. If any of the dialog fields are empty, catch this and display an appropriate GUI message (a JOptionPane is the quickest way to do this, several examples throughout the code).
    b. The username must be an existing user
    c. The role must be that username's present role
    d. If either b or c is not true, catch this and display an appropriate GUI message
5. Otherwise update the password, including making a new salt. To do this you should refactor *UsersSingleton,* adding a new method patterned after *createPasswordMapping* (if you think about the commonalities and differences between the "create" and "update" cases, you can do a smart simple refactoring).

Be sure to check that the proper json files have been updated.

<u>Submission for Part 1:</u>

- Make a zipfile named <asurite>_lab1_part1.zip from the root of your source directory. *Make sure you run 'ant clean' and 'ant post-clean' to remove all build artifacts before creating your zipfile.*
- Your source tree should have all code for tasks H1, H2, and H3.
- Please indicate where in the source files you wrote code for each task by prefixing the change with a comment "Task H1, "Task H2", or "Task H3" as appropriate. You may also include the step number.
- If your code is not stable do NOT submit it! You are responsible for fully testing your solution, and creating a proper submission package! Get something working fully before moving ahead to the next (part of a) task. Use git locally (you don't need github really) to manage your work so you do not lose changes!
- Late submissions are not accepted. You may however, submit as many times as you like before the submission deadline, we always grade the latest submission.
- Please put a readme.txt or readme.md files in the root of your source tree to inform us of anything you think we need to know to effectively assess your submission.
- Your code must build and run for us on our systems. Therefore, do not use any hardcoded paths! A common mistake is to hardcode where the json files are in your src tree – do not do this!

Part II: Network Security : Implementing a Sniffer and Spoofer (50 points)

Background: So, what does Sniffing and Spoofing a network mean?

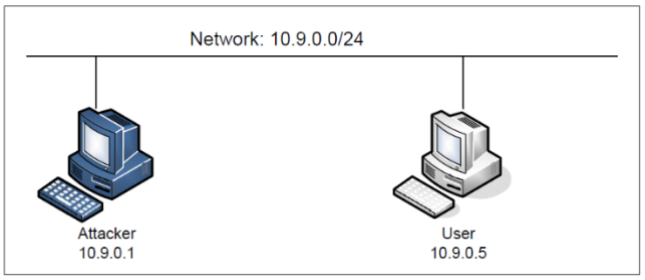
Sniffing is a process of monitoring and capturing all data packets passing through a given network. Attackers use sniffers to capture data packets containing sensitive information such as passwords, account information, etc.

Spoofing is the act of disguising a communication from an unknown source as being from a known, trusted source. Spoofing can be used to gain access to a target's personal information, spread malware through infected links or attachments, bypass network access controls, or redistribute traffic to conduct a denial-of-service attack.

Task N1: Provide one example of sniffing and one separate example of spoofing attacks from the real world (in the news) or via popular culture (movies). How could these attacks have been prevented in the first place? (6 points)

Environment Setup for Task N2 using Container

In this task, we will use two machines (docker containers) that are connected to the same LAN. The figure below depicts the environment setup using containers. We will perform all the attacks on the *attacker* container while using the other container as the *user* machine.



Container Setup

Please download the provided *'Labsetup.zip'* file to your VM, unzip it, enter the '*Labsetup'* folder, and use the *docker-compose.yml* file to set up the lab environment. Start the attacker and user containers using 'docker-compose up'. If you look at the Docker Compose file, you will see that the attacker container is configured differently from the other container.

Here are the differences:

Shared folder: We use the attacker container to launch attacks. The attacking code is put inside the attacker container. Code editing is more convenient inside the VM than inside the container because we can use our favorite editors. Thus, if you look at the Docker Compose file, you see that a 'volumes' folder is shared between the VM and the attacker container. (Remember your 'docker volume ls' command)

Host mode: The attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. When the container is in the host networking mode, it can see all the traffic and even has the same IP addresses as the host. (Remember 'docker network ls' and 'docker network inspect <id>' for docker networking information)

Getting the network interface name: When we use the provided Compose file to create containers for this lab, a new network is created to connect the VM and the containers. The IP prefix for this network is 10.9.0.0/24, which is specified in the docker-compose.yml file. The IP address assigned to our VM is 10.9.0.1. We need to find the name of the corresponding network interface on our VM because we need to use it in our programs. The interface name is the concatenation of "br-" and the ID of the network created by Docker. Use the ifconfig command to list network interfaces. Look for the IP address 10.9.0.1. Again, since we are using the host networking driver, our container is effectively sharing our VM's network interface.

```
[01/23/21] seed@VM:~$ ifconfig
br-46fb5e9db317: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:f5:c1:96:a3  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Using Scapy to Sniff and Spoof Packets

Scapy is a tool used to do sniffing and spoofing. Moreover, it can also be used to construct other sniffing and spoofing tools i.e., we can integrate the Scapy functionalities into our own program. To use Scapy, we write a Python program and import all Scapy's modules. **Note**: To spoof the packet, the program should be run using the root privileges.

```
seed@VM:~$ chmod a+x mycode.py
seed@VM:~$
seed@VM:~$ mycode.py
```

Make mycode.py executable (another way to run python programs)

SNIFFING: Sniffing Packets using Scapy

The sample code to sniff the packets is provided below:

```
#!/usr/bin/env python3
from scapy.all import *
import sys
filterCriteria= sys.argv[1]
def print_pkt(pkt):
      pkt.show()
pkt = sniff(iface='br-46fb5e9db317', filter= filterCriteria, prn=print_pkt)
```

Create a new python file '*sniffer.py*' inside the *attacker* container and add the above code. Change the interface ID to the interface ID of your system. Run the *sniffer.py* program with root privileges and provide filtering criteria as a command-line argument. To capture ICMP packets the command will be: *python3 sniffer.py 'icmp'* .

Now, open a new command prompt inside the *user* container and ping the attacker container.
ping 10.9.0.1 (IP of attacker container)

```
root@c9e41e01ebab:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.113 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.094 ms
```

```
root@VM:/volumes# python3 sniffer.py 'icmp'
###[ Ethernet ]###
  dst       = 02:42:be:04:ac:f2
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 31153
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xace0
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xe640
        id        = 0x12
        seq       = 0x1
###[ Raw ]###
           load      = '\x9eg\x15`\x00\x00\x0
e\x1f !"#$%&\'()*+,-./01234567'
```

```
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:be:04:ac:f2
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 59891
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7c9e
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xee40
        id        = 0x12
        seq       = 0x1
###[ Raw ]###
           load      = '\x9eg\x15`\x00
e\x1f !"#$%&\'()*+,-./01234567'
```

The above sniffer program captures ICMP packets on the 'br-46fb5e9db317' interface and for each packet captured callback function print_pkt() will be invoked; this function prints out the packet's information. Moreover, for each packet, an **echo-request** and its corresponding **echo-response** are generated.

Scapy filters are expressed in a format called BPF (Berkeley Packet Filter), you can see the syntax here.

Task N2: (12 points) For the following cases write filtering criteria as command-line arguments and provide screenshots (that shows the command-line and the results) that demonstrate you can capture the packets for the following cases:

a) (4) Capture only the ICMP packet
b) (4) Capture any TCP packet that comes from a '10.9.0.5' IP and with a destination port number 23 (telnet port)
c) (4) Capture packets that come from or go through 173.194.208.0/24 subnet.

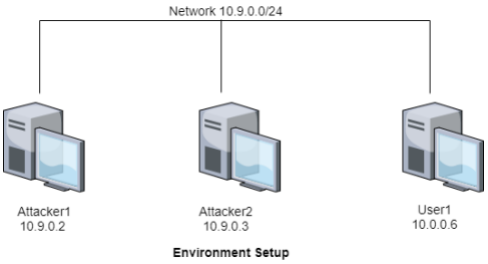
Note:

1) To test task N2.b, you need to telnet from user container to attacker container. *Command: telnet 10.9.0.1*
2) To test task N2.c, you can ping 173.194.208.103 from the user container. *Command: ping 173.194.208.103*

Task 3 Setup for SPOOFING: Spoofing ICMP Packets

Environment Setup for spoofing

We will use three machines(containers) that are connected to the same LAN. The figure below depicts the lab environment setup using containers. We will perform spoofing attacks from attacker1 and attacker2 containers while using the user1 container as the user machine.



Environment Setup

Container Setup

Make sure none of the containers are running. Stop any running containers from the previous task using *'docker-compose down'*.

Please enter the '*Spoofing'* folder inside the '*Labsetup'* folder and use the *docker-compose.yml* file to set up the spoofing environment. Start the attackers and user containers using '*docker-compose up'*

Background

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. For this task, we will spoof ICMP echo request packets in *attacker* containers and send them to the *user1* container. We will use Wireshark to observe whether our request is accepted by the receiver. If accepted, an echo reply packet will be sent to the spoofed IP address.

```
#!/usr/bin/python3

from scapy.all import *
a=IP()                          ①
a.src = "1.2.3.4"               ②
b = ICMP()                      ③
p = a/b                         ④
send(p)                         ⑤
```

The code in the box shows an example of how to spoof ICMP packets. Line ① creates an IP object from the IP class; a class attribute is defined for each IP header field. Line ② shows how to set the source IP address field. Line ③ creates an ICMP object. The default type is echo requests. In-Line ④, we stack a and b together. The / operator means adding 'b' as the payload field of 'a' and modifying the fields of 'a' accordingly. We can now send out this packet using send() in Line ⑤.

Note: Use ls(a) or ls(IP) to see the attribute names/values of a packet. We can also use a.show() or IP.show() for the same.

Task N3: (10 points) Run the '*SpoofingDriverProgram.py*' program in the '*Spoofing'* folder from the VM command-line. The program executes '*Attacker1_ICMPPacket_Spoofer.py'* and '*Attacker2_ICMPPacket_Spoofer.py'* inside *attacker1* and *attacker2* containers respectively. These programs send spoofed packets from *attacker1* and *attacker2* containers to *user1* container. Use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, you may see an echo reply packet sent to the spoofed IP address (you may also get [No response seen]). Provide a screenshot of the Wireshark application demonstrating packet spoofing. Explain your understanding of spoofing and how the above program spoofed the packets, noting where in Wireshark you can see ICMP packets from each attacking container.

Note: you may need to do the "chmod +x SpoofingDriverProgram.py" as shown above to ensure it is executable.

Task N4 Setup for SNIFFING AND THEN SPOOFING

Container Setup

Make sure none of the containers are running. Stop any running containers using the 'docker-compose down' command.

Just as in Task N2, please use the '*Labsetup'* folder and use the *docker-compose.yml* file to set up the lab environment. Start the attacker and user containers using the 'docker-compose up' command.

Task N4: (22 points) In this task, combine the sniffing and spoofing techniques learned above to implement a sniff-and then-spoof program. You need two machines on the same LAN: the *attacker* and the *user* container. From the *user* container, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the *attacker*, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task.

a) Submit your 'sniff-and-then-spoof' code. Name the file taskn4.py (3)
b) Provide a screenshot of the Wireshark application demonstrating you can sniff and spoof the packet. (4)
c) Ping the following 3 IP addresses from the user container. Report your observation and explain the results. (9)

```
ping 1.2.3.4      # a non-existing host on the Internet
ping 10.9.0.99    # a non-existing host on the LAN
ping 8.8.8.8      # an existing host on the Internet
```

Notes:

1. We are suggesting you use python since this short program is mostly a combination of the previous sniffing and snooping programs. However, you may also use C, Java, or shell scripting to accomplish the same outcome.
2. We will provide credit if you show the ICMP reply (response) in Wireshark. But to get full credit you need to create the full ICMP response so that the reply is shown in the terminal window to the ping program. A website that explains what fields you need to set in your ICMP response is here: https://ivanitlearning.wordpress.com/2019/05/20/icmp-redirect-attacks-with-scapy/comment-page-1/

Submission for Part II:

For submission, create a single zip file named <asurite>_part4.zip that has within it:

- A file named task_n.[docx|odt|txt] that provides complete answers as directed tasks N1, N2, and N3, and the parts of task N4 with your screenshots and ping observations and results (parts b and c above)
- Your file named taskn4.py (part a above) or an alternate non-python solution named taskn4.[ext]
- Do NOT give us a format other than .zip. No .rar, .7z, or .tgz formats.
- You may give us a README.txt (or .md) file that explains to us anything you think we need to know and how to run your code.