Machine Learning Spam Detection Lab

# Required Materials:

1. Computer Running Linux/MacOS (Linux is ideal)
2. Python 3.7+
3. Java 8 JDK
4. Joern CLI
5. Git
6. Source-Highlight (optional, but nice to have)

# Covered Topics:

1. Static Analysis
2. Machine Learning

# Background:

Static code analysis is the process of analyzing a program’s code to try and identify patterns, vulnerabilities, call graphs, etc. It is the process of extracting information about a program from its source code. In this lab we will run static code analysis on a variety of programs including a known vulnerable version of the Linux Kernel. This lab will also include

# Pre-Lab Preparations:

1. Install Python 3.7 or later.
2. Install the Java 8 JDK.
3. Once you have Python and Java installed, you will need to download the static analysis software that we will be using in this lab. We will be relying on a program called Joern. You can get Joern from the following link: <https://github.com/ShiftLeftSecurity/joern/releases/latest/download/joern-cli.zip>. Once downloaded, unzip the downloaded directory and open a terminal. Run `cd path/to/joern-cli` to move into the joern-cli directory that you just downloaded (you will need to specify the path).
4. Test your download (and ensure that you have all required dependencies by running: `./joern`. You should see the program compiling, the name of the program will print in ASCII, and then you will be in an interactive shell. Type exit to leave this shell.

# Static Analysis with Joern

## Background:

In this portion of the lab, we will become familiar with how to use Joern to extract data about C and C++ codebases. We will then use Joern’s interactive shell and python module to explore the previously extracted data.

## The Lab:

1. We will need to get code to statically analyze. We will be statically analyzing 3 codebases. The first is Microsoft’s REST CPP API. It has no known vulnerabilities, but is a large codebase and is a good example of analyzing C++ code. Run `git clone [https://github.com/microsoft/cpprestsdk.git`](https://github.com/microsoft/cpprestsdk.git%60) to download yourself a copy and note where it is downloaded.

From within the previously downloaded joern-cli directory run the following command: ` ./joern-parse [path/to/cpprestsdk] --out [path/to/output/graph]`. Let’s briefly go over that command. The `./joern-parse` calls the script that will extract the graphs of the program. The `[path/to/cpprestsdk]` should be the **relative** **path** to the Microsoft code you downloaded earlier. Finally the `--out [path/to/output/]` tells the program where to place the final call graph binary. The output path should end in `cpprestsdk\_cpg.bin`. This is not a required name, but it should be a .bin file and using an easy name to remember will help when we have multiple output files.

One you click enter on that command it will take a while to run, but once complete, you should see a file named `cpprestsdk\_cpg.bin` or whatever you chose to name the file in the directory you specified.

1. Now we want to explore the code using Joern’s built in tool. Run `./joern`. The tool will recompile and then enter an interactive shell.

Run: loadCpg(“cpprestsdk\_cpg.bin”). Note this should be the path to the previously extracted file. If you moved it or changed its name, this command will be different.

1. Now that you have loaded the code, we can begin exploring it. First let’s extract all the methods in the C++ REST SDK. To do this run: `cpg.method.name.l`. This will print all the methods found in the code. To save this output to a file, run: `cpg.method.name.l |> methods.txt`.
2. Let’s explore some of the method calls in a little bit more detail. First, let’s see every time that memcpy (a C function for copying memory) is called.

Run: ` browse(cpg.method.name("memcpy").callIn.code.l)`. This will List every time that the function memcpy is called. Please play around with this. Take any function from the saved list of functions we made before and replace “memcpy” with that function name in the previous command and see what comes up.

1. Now let’s get a little bit more information. Run: ` cpg.method.name("memcpy").callIn.dump`. This will give you the code surrounding the memcpy function call. This will be much easier to read if you [install source-highlight](https://www.gnu.org/software/src-highlite/), but it is not a requirement. You can also write this information to a file using the same notation as before: ` cpg.method.name("memcpy").callIn.dump |> function\_dump.txt`.

Please do this with 3 function names aside from memcpy and include those in your lab submission.

### Submission Instructions:

Please zip your extracted bin files as well as your text files (generated through Joern) and then submit the zip file. Please name each bin file and text file so that it is easy to tell what each file contains. If you fail to do so, your lab might be graded improperly.

# Machine Learning on Static Analysis:

# Background:

Manually analyzing large codebases can be time consuming and outright impossible. Additionally, using standard methods of static analysis can often miss vulnerabilities that can be devastating to a system. Because of this, applying a machine learning model to identify malicious code through static analysis is a sound strategy. In this lab, we will be taking the previously generated bin files (code graphs) and training a machine learning model on them.

This part of the lab is based dataset from the following paper:

μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection by Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin, Fellow, IEEE

<https://arxiv.org/pdf/2001.02334.pdf>

# The Lab:

## Submission Instructions:

Submit a zipped directory containing your model code, data files (bin files containing call graphs). Whoever is grading your model should be able to just run a single python script within your unzipped directory and have the model retrain itself and print the accuracy.