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 full 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. You need to complete part 1 of this lab first because some of Joern’s functionality is explained in part 1 and is not re-explained here.

# The Lab:

1. If you do not already have it, download the dataset that accompanies this lab. This dataset contains 4 directories. Each directory contains samples of code. In each directory all of the samples exhibit the same code flaw. The 4 directories correspond to 4 vulnerabilities: command injection, heap based overflows, stack based overflows, and memory leaks.
2. Use the joern-parse script to extract CPG for each directory. This process is the exact same as the previous lab, simply point the joern-parse script to the specified directory and specify an output file. You should end up with 4 CPG files, one per directory.
3. As seen in the previous section of the lab, Joern has fully featured scripting language for traversing and querying the CPG database files. This is outside of the scope of this lab, so to extract all of the function data we will be using a pre-written script that is included in the Joern project. It has been provided in your lab files under joern-scripts. You are looking for “cfg-for-funcs-dump.sc”. This is a Scala script that extracts all of the function data from each CFG file and writes it to a JSON file. To run the script, launch joern (Run “./joern” from where it is installed). Load your desired CPG file (you will have to do this for each CPG file). Then run the following: ‘cpg.runScript(“absolute\_path\_to\_script”)’. You must put the full path to the provided script, otherwise it must be in the same directory as the joern executable. Relative paths (using ../ or ./ will not work). Upon completion of the script a “cfg-for-funcs-dump.json” file will be in the directory with the Joern executable and it will contain all of your training data. It is recommended that you rename this so you can track which JSON file corresponds to each CPG file.
4. This step is a checkmark, ensure that you have 4 unique JSON files, one for each vulnerability directory in the provided dataset.
5. Extract the necessary training data from the json files, for this example in the lab we will only need the function names, however its worth recognizing that in a real world (non-lab) setting, you would want to train on much more than just function names. We are only able to do this due to the nature of our dataset.
6. Truncate the length of each function name to a consistent length so that the training data is all equal length and then set aside roughly 10% of the data for testing the model afterwards.
7. You should end up with 4 arrays once all the data preprocessing is complete, the first two should be the training data and a second array of equal size that contains the label for each bit of data. The second two arrays should be the same thing, but with the 10% of training data set aside for testing later.
8. Using the Keras/Tensorflow API as a reference point, build a sequential model with at least 5 layers and 1 output layer. As a hint, the output layer should be a Dense layer with as many units as possible categories of classification (since our dataset has 4 types of vulnerabilities, the dense layer should have 4 units).
9. Once you have architected your model use the Model.compile() and Model.fit() functions to train your model.
10. Finally, add the following code after you have fit your model to see the accuracy (ensure that “model” is the name of your model object and “x\_test” and “y\_test” correspond to the 10% testing data arrays you specified before):

score = model.evaluate(x\_test, y\_test, verbose=0)

print('Test loss:', score[0])

print('Test accuracy:', score[1])

1. It is completely reasonable to expect >85% accuracy on this model since we are using clearly defined function names. In writing this lab, the answer key model has 99% accuracy, but 85% will be all that is required in your submission.

## 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.