ML Based Malware Analysis of Executable Files and Source Code

Team Name - Provaxin

Team Members -

Priydarshi Singh
Aman Agrawal
Devanshu Singla
Niket Jain
Shubhankar Gambhir
Varenya Srivastava

The Problem

- Malware attacks are increasingly becoming more common and frequent.
- There is an ever-increasing need to build tools to identify and restrict malwares.
- Usually antivirus tools detect and stop **malicious executables**.

But…

CVE-2022-23812: This affects the package node-ipc from 10.1.1 and before 10.1.3. This package contains malicious code, that targets users with IP located in Russia or Belarus, and overwrites their files with a heart emoji.

CVSS Base Score: 9.8

The Problem

- Developers often (ie, almost always) use popular open-source packages without checking their code.
- It is trusted that very popular packages will definitely not be malicious.
- The package node-ipc used to have >1M downloads every week.

- This incident has changed open source software as we know it.
- There are demands from the open source community to build tools for checking malwares in packages downloaded by package managers like npm and pip.

Solution Overview

We use both ML based and Rule based techniques to perform static as well as dynamic analyses of executable files and source code.

Malware Analysis of PE Executables using Machine Learning - 1st Model

- We have used Ember dataset which consists of feature set of 1.1 million PE executable binary files. The dataset is divided in 400K malicious, 400K benign and 300K unlabeled files.
- The dataset comprises of 2.3K features like entropy of binary, headers related data, etc.
- Due to computation limit we have to train our model on 0.2 million samples and test on another 0.2 million samples.
- We tried many ML classifiers to classify the data some of them being AdaBoost-SAMME algorithm, Multi-layer Perceptron Classifier, etc.
- Best accuracy of 86.5% was achieved using Random Forest Classifier after hyperparameter tuning.

Malware Analysis of PE Executables using Machine Learning - 2nd Model

- Since our first model was a generic one, we decided to make our model more “malware class specific”, training on a particular class of malware .
- Our training dataset had two classes: Benign and Malicious
- Sourcing the dataset:
- For malicious files:
    - We wrote a mail to virusshare.com to get access(invite only) to the malware dataset
- For benign files:
    - Same as the previous model

Results and Observations for 2nd Model

- From the previous set of ~2000 features, we selected around 54 features from the feature set, based on a paper by Dr Viorel, et al (Reference added in the docs)
- The accuracy obtained was >0.98 for our model, which was based on the K-Nearest Neighbour Approach.
- Limitation:
  - On a different class of malware corpus, the model sometimes miss classifies a malware file as a benign one. (Will be shown in the demo)

Malware analysis of source code

- Python and JavaScript are most popular languages whose programs are shared as source code (ie., they can't be compiled into an executable)
- We aimed to build the following tools:
  - `propip`: A wrapper for pip (official package manager for Python).
  - `pronpm`: A wrapper for npm (official package manager for NodeJS).
- The tools work exactly as the original pip and npm.
- After the packages have been downloaded, the tools perform malware analysis on the downloaded code.

JavaScript Malware Analysis

- We perform a **dynamic analysis** of JavaScript code.
- The JavaScript code is run in a **sandboxed environment**.
  - The sandbox is created by redefining powerful functions (like `eval`, `fetch`, etc.) in pre-existing objects and modules (like `window`, `console`, etc.) to log the parameters and return a default value.
  - We use an open source tool (`malware-jail`) for creating the sandbox.
- The state of the sandbox is analysed for potentially malicious actions.
- Usually, a package has its code spread over multiple files, which doesn't work properly in the sandbox. We use `browserify` to combine all the source code into one single file.

JavaScript Malware Analysis

- We ran 39,000+ Javascript malware samples* in the sandbox and used the results to create rules for detecting a malware.
- Examples of rules where a file is flagged as a malware:
  - If the code makes a request to a known malicious website (as determined using IP Quality Score API).
  - If the code downloads some executable files that are detected as malware by our ML models.
  - If it uses `eval` to try to execute obfuscated code .

*Source: github.com/HynekPetrak/javascript-malware-collection

Python Malware Analysis

- We perform a **static analysis** of python code.
- The python code is checked for the potentially vulnerable calls/functions and imports using rule-based approach.
- The imported files are further checked recursively for dangerous code.
- Report is generated after analysis which gives count of dangers of different threat level.
- References
  - https://docs.openstack.org/bandit/1.4.0/blacklists/blacklist_imports.html
  - https://docs.openstack.org/bandit/1.4.0/blacklists/blacklist_calls.html

Python Malware Analysis

- We have also build Malicious URLs classifier using Machine Learning techniques.
- We performed feature extraction on a dataset consisting of 400k URL samples and used ML algorithms - Logistic Regression and Linear Support Vector Classification for the task.
- We were able to achieve 96.2% and 98% accuracy for the respective models.

Limitations of Source Code Analysis

- For `pronpm`: some packages don’t run any code upon just importing them, we need to call some functions. In the current form, the analyser cannot call functions in the package.
- For `propip`: since this is a static analysis, we cannot correctly determine obfuscated calls to eval, or measure how many times a function is called.
- Rules are not exhaustive.

Remaining Tasks (to be completed by 29th)

- Complete the rules for `propip.`
- Combine the separate commands to create final wrappers (`pronpm` and `propip`)
- Find examples of real malicious packages that are flagged by `pronpm` and `propip`.

Thanks.