Michigan State University

Team Proofpoint

Improved Detonation of Evasive Malware

Project Plan

Fall 2018

Proofpoint Sponsors

Brad Woodberg

Michigan State Capstone Team

Jack Mansueti

Ryan Gallant

Sean Joseph

Ian Murray

Tae Park

Table of Contents

Executive Summary

Proofpoint is a cybersecurity firm focusing on enterprise-level threat tracking, mitigation, and elimination. Although Proofpoint is known for client endpoint protection, they employ an extensive R&D infrastructure for handling new malware. The foundation of this infrastructure is the sandbox: a malware quarantine set in a highly restricted and transparent environment.

For years the sandbox was the premier tool for analyzing unfamiliar malicious code. Combining reverse engineering with system monitoring techniques, cybersecurity experts are able to determine crucial characteristics of malware – including its trigger conditions, payload, intended targets, and origin. However, malware programmers have recently become aware of the advantage sandboxing grants malware researchers and have created a new class of malicious code in response. Such code has been termed evasive malware. This new malware has the ability to recognize quarantine environments and will function entirely different in these scenarios. This is done with the intent of making the malware more difficult to examine, thus removing any discernible advantage of sandboxing. For example, malicious code that finds itself within a sandbox may modify its triggers or payload, or do nothing at all. With this new evasive malware becoming steadily more common, it is now necessary for threat researchers to once again gain the upper hand in the cybersecurity arms race.

Combating evasive malware is a complex and difficult challenge. Proofpoint's approach involves reverse engineering malware files and following a two-step process. First, a piece of malicious code is analyzed to find where exactly where the sandbox-recognizing code lies. Once this is complete, the code must then be modified to negate its recognition abilities. The malware will now run and can be researched as originally intended.

Functional Specifications

Malware authors have recently evolved their craft to include new evasive techniques. This improved malware is able to avoid detection by sandbox analysis software. This causes a multitude of problems. The malware avoids analysis, and its signatures and heuristics remain unknown to cybersecurity experts. With the prevalence of evasive malware on the rise and the overall quantity of malware growing ever larger, manually reverse engineering each sample is no longer feasible. As a response, Proofpoint has proposed a tool aimed at the improved detonation of quarantine-avoiding malware.

Proofpoint's solution is a comprehensive system capable of detecting a sample of evasive malware. The fundamental goal is to programmatically identify malicious code that displays signs of checking for a quarantine environment. The program will then 'flag' these samples as evasive. In doing so, cybersecurity researchers are able to easily conduct additional manual analysis. This comes with the benefit of discovering new malware signatures, gaining awareness of sophisticated advanced persistent threats (APTs), and contributing to databases of known malware samples.

For added functionality, the application will support autonomous malware modification to remove all sandbox-related checks. Reverse engineering will be used to cause the malware to execute in quarantine identical to how it would in any live system. This creates several benefits to malware analysts. The primary advantage is the decreased workload on Proofpoint staff. Instead of reverse engineering samples manually to determine the payload, it can be done automatically in the quarantine environment. Using this functionality, evasive samples require no more additional effort than traditional malware. Engineers still have the ability to conduct manual analysis for additional details, but it is not required.

The program will also create auxiliary data related to multiple-sample analysis. Frequently occurring samples and repeated patterns will be documented for study. A similar record will be taken for suspicious network traffic. The purpose of this large-scale data collection is to allow malware engineers to make trend-based predictions.

All of these functions will be accessible through an intuitive web interface. A dashboard will present a high-level view of the system elements. A series of widgets will illustrate features pertaining to various elements of the project. Through these widgets, users can access the status of individual software components, a queue of samples to be processed, and a results page - containing the derived conclusions from the data processed.

Design Specifications

Overview

This project is a system for the automated analysis of malware, detecting evasion techniques and forcing the program to fully execute. By using the Cuckoo sandboxing framework and Suricata network traffic monitor to analyze incoming samples and custom Python modules to force software to run, the process for detecting smarter malware becomes simpler and more automated. The web UI allows analysts to easily visualize data and manage malware samples through dashboards and classification.

Web Dashboard

The web dashboard displays real time data such as number of samples processed, results of analysis, and top evasion signatures. Data can be filtered and sorted based on a number of factors in order to more efficiently sift through samples. The dashboard also provides analysts with the opportunity to "drill down" on individual malware samples, pulling up a Cuckoo page with additional information about the samples for additional analysis.

Malware Analysis

The malware analysis function uses Cuckoo and Suricata to classify malware based on its behaviors. Both systems look for specific signatures, labeling a given sample based on which and how many behaviors it matches. From there, the program will decide whether to label the malware as non-evasive, an evasive sample that will be modified to fully execute, or an unknown sample that should be escalated to analysts for proper review.

Malware Modification and Detonation

This portion modifies evasive malware in order to force it to fully execute. By reversing checks and modifying sections of the underlying assembly code, the aim is to create situations in which the malware executes in spite of safeguards that aim to prevent programs from detonating in secure or limited environments. This results in more information and data for analysts to follow up on and design solutions for, as they can now see the full range of behaviors in a given malware sample.

Use Cases

1. Analysts will use this software to counteract malware samples that have shown evasive qualities in execution. Without using the software, the Cuckoo sandbox will only report that the sample performed evasive checks. After running samples through the team's

Improved Detonation of Evasive Malware modules, the sample's evasive capacity is negated, and Cuckoo will report the full payload to the analyst.

2. Analysts will use this software to discover evasive malware trends over a scope of multiple samples. The web interface displays broad summary statistics such as top signatures allowing Proofpoint analysts to research past and predict future trends in the evasive malware processed.

User Interface Design

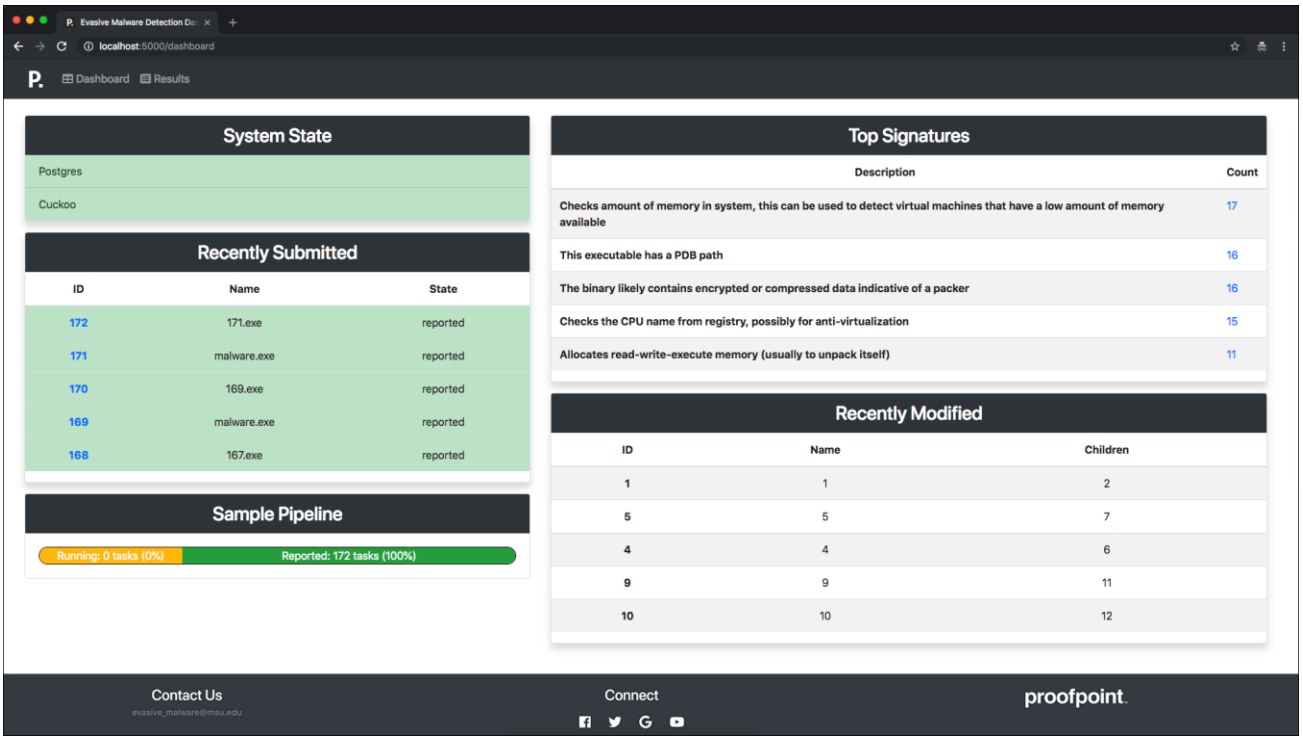

Figure 1: Home Screen Dashboard

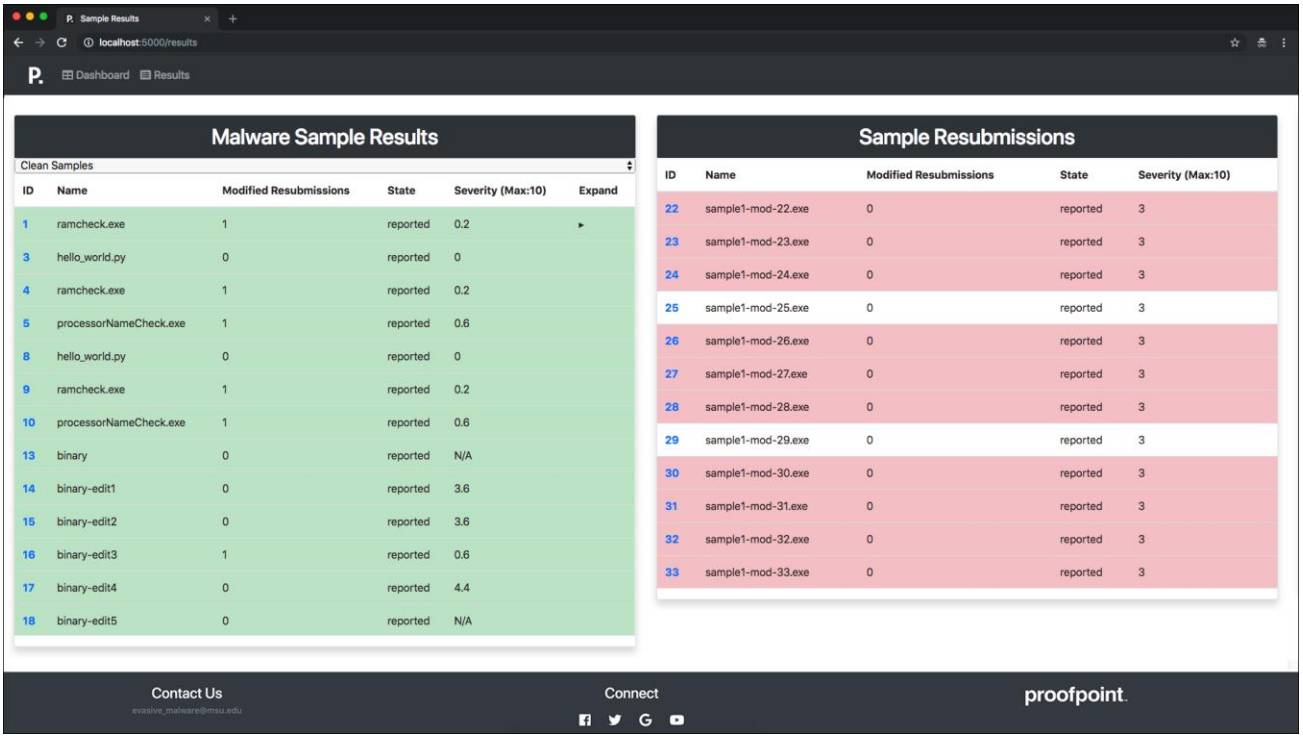

Figure 2: Malware Sample Results

User Interface Components

Seen in Figure 1 are the five widgets presented on the web dashboard. The system state widget illustrates the status of the backend components. The recently submitted widget presents the user with the state of recently processed samples. These samples may be queued, processing, or reported. The sample pipeline widget shows an overarching view of Cuckoo's workload. The top signatures widget presents the user with the most frequently occurring evasive and payload-based signatures. Finally, the recently modified widget narrows the data from total submissions to only those which have been flagged for modification, along with the number of modified children that were created.

Figure 2 is the submission results page. This is an exhaustive list of all samples that have been submitted and modified. Presented the left are the original samples. Illustrated on the right are the resubmissions of samples with their evasive checks negated. This page allows the user to apply numerous filters to narrow the information down to only what is desired by the malware analyst.

Technical Specifications

Overview

The entire system is built on a VMware hypervisor. The web end is run using Bootstrap and jQuery for searching, filtering, and dashboards. The web server is run with Flask, Apache, Python, and SQLAlchemy. It communicates with a database using SQL to receive malware samples, then forwards them to the Cuckoo system for analysis. If Cuckoo and Suricata detect behavior indicative of evasive malware, the sample is sent to a python script, which circumvents the sample's safeguards and forces it to fully detonate. Cuckoo forwards the results of the detonation and disassembly to the web server and they are displayed on the web front end.

Frontend User Interface

Bootstrap, jQuery, HTML5, and CSS3 are used to effectively present users with appropriate data from the malware detonation system. The web application is implemented with Python 3.6 and Flask micro web framework. Apache is used to reliably route front end web traffic as well as malware sample jobs. Our data persistence needs are met by a PostgreSQL Relational Database and SQLAlchemy to provide Object Relational Mapping for our Flask application.

Backend Malware Analysis

The technologies used for backend malware analysis are Cuckoo and Python2.7. The backend system is responsible for evasive malware detection, disassembly and modification, and forced detonation. When a malware sample is received, it is run through Cuckoo, an automated malware analysis system. Cuckoo analyzes the sample and reports on behaviors and characteristics. These behaviors are: execution steps, network traffic, and memory usage. The sample will then be assigned tags based on its unique behaviors.

If the sample demonstrates one or more evasive behaviors, it is passed along to the disassembly/modification stage. During this stage, the sample is disassembled and analyzed by a Python2.7 utility. The utility will find lines in the binary (assembly language) where the sample diverges under specific conditions. It will then modify the binary in order to change the steps of execution of the malware sample. A single sample can be modified multiple times, and output multiple modified executables.

Once the sample is modified, it is run through the Cuckoo sandboxing system once again. At this stage, a previously evasive malware sample should detonate properly, regardless of its environment. This allows the malicious behaviors to be noted by the web UI.

System Architecture Diagram

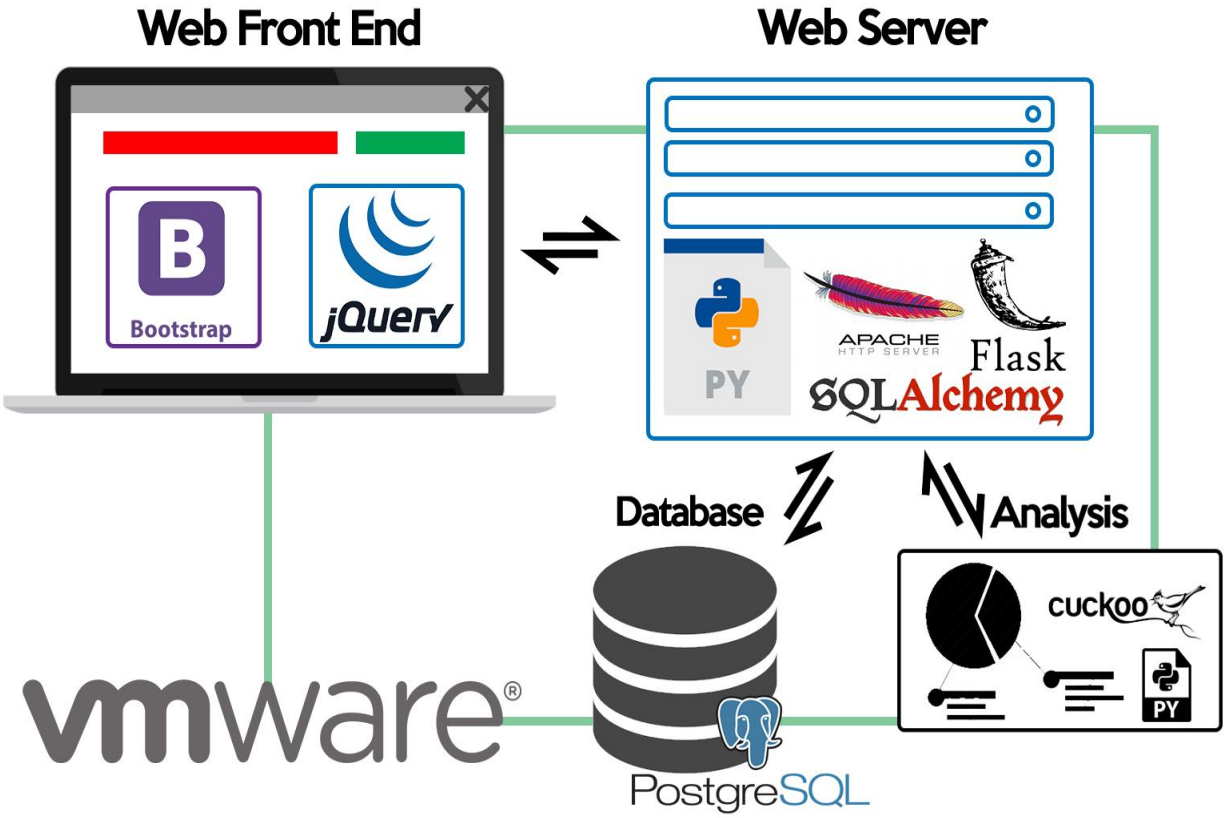

Figure 3: System Architecture

System Architecture

The web front end is developed using Bootstrap and jQuery and communicated with the web server, which utilizes Python, Apache, SQLAlchemy, and Flask. The web server communicates with the Cuckoo sandbox, Suricata network monitor, and PostgreSQL to gather raw data. All of these services are hosted on Proofpoint's infrastructure using a VMware virtual machine.

Process Flow Diagram

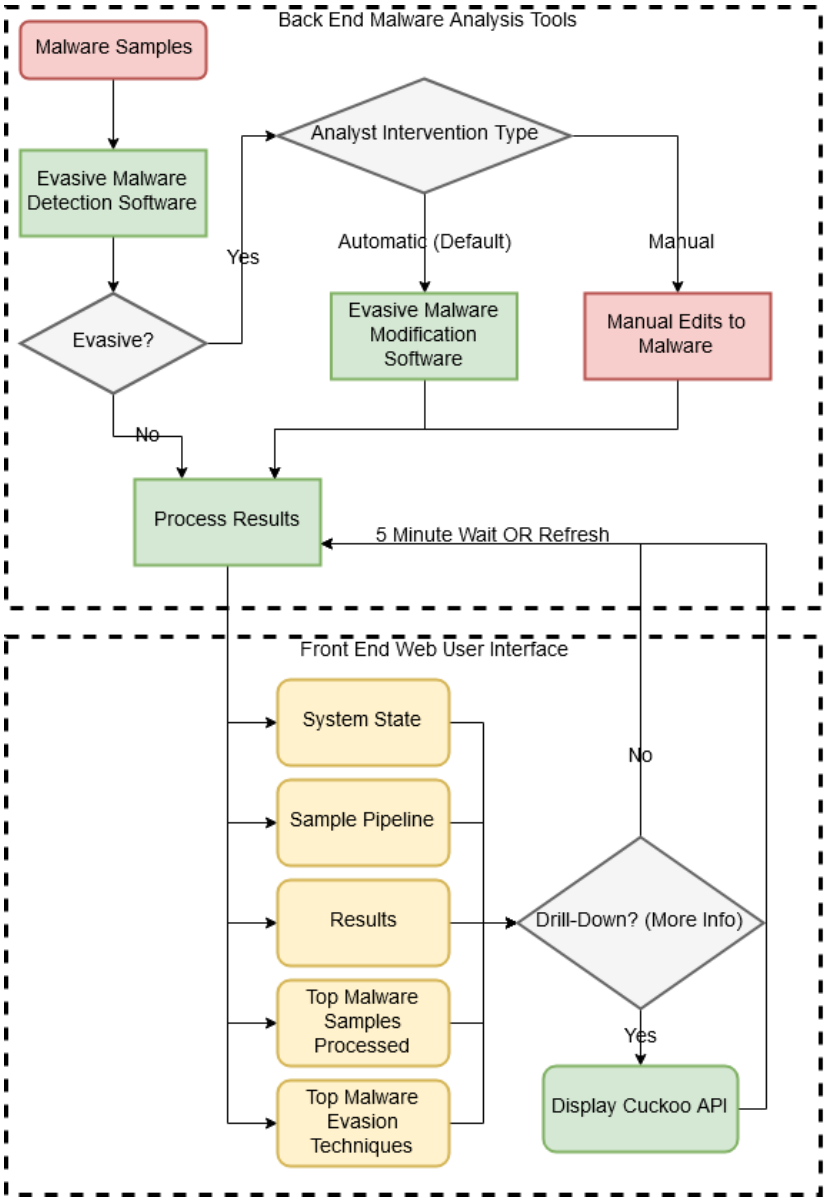

Figure 4: Data Process Flow

Process Flow

Malware samples are classified as evasive or benign based on the output of the detection module. If the sample is evasive, it is sent to the modification module for detonation. Otherwise, it is sent for processing. Once modified, evasive samples are sent for processing. Once processed, the results are sent to the web user interface for user analysis. This interface periodically checks for updates and will fetch new data on a five-minute basis.

Executable Analysis and Modification

The process of analyzing and modifying malicious executable files can be a difficult process to do, even manually. This can be compounded by malware authors efforts to obfuscate their code. Below is an example of modifying the execution of a simple, benign executable. The first step in the process is to run the executable. Figure 5 shows the example executable is a number guessing game. The initial guess is 22, which yields an incorrect result.

```
>>>./msucapstone
***Welcome to the MSU Capstone Lottery Game!***
Please guess this week's magic number 1-100:22

Sorry that is not the magic number, you lose
>>>
```

Figure 5: Executing an unknown binary

Tools such as objdump can also help analysts to understand the behavior of a binary. The utility objdump disassembles binaries of many different file formats. Figure 6 shows the first few lines of the objdump output. The unknown binary has a file format elf64-x86-64. ELF or the Executable and Linkable File format is commonly used by Linux distributions.

```
>>>objdump -d msucapstone

msucapstone:     file format elf64-x86-64
```

Figure 6: Executing objdump to disassemble the guessing game

Examining the main section of the executable is a good place to start. Programmers will be familiar with the concept of the main function being the entry point for their program.

Figure 7 shows the main section objdump output for the unknown binary. The far-left column shows the location of each instruction. The second column displays the bytes that make up that particular instruction. The far-right column shows the x86-64 assembly that correspond to the byte instructions.

The unknown binary's main section shows a few calls to functions like puts, printf, and scanf. The functions puts and printf are both ways of printing output to the screen. While scanf is a function for taking input. Immediately following the call to scanf is a cmp instruction that compares the content of register %eax to a the hex number 0x4f. The decimal representation of 0x4f is 79.

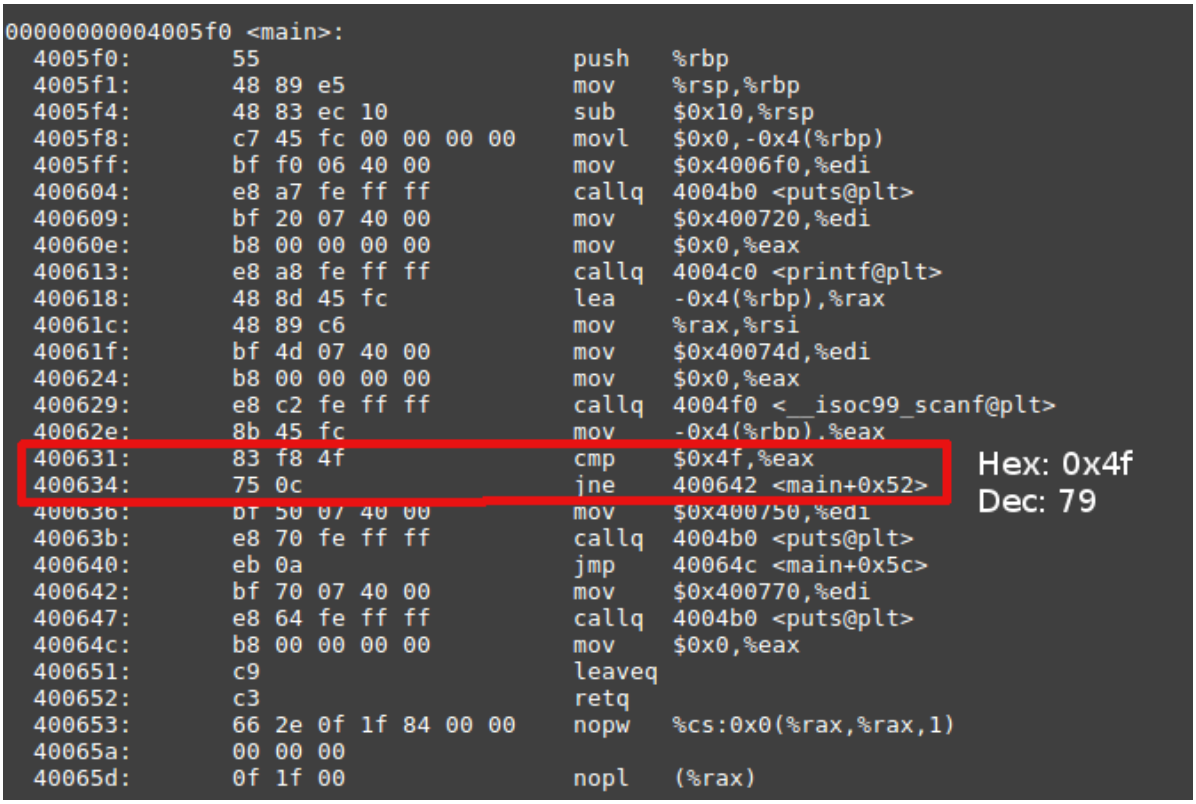

Figure 7: Learn the magic number by examining the main section

Figure 8 shows the output of the unknown binary when the number 79 is used as input. 79 is the correct answer.

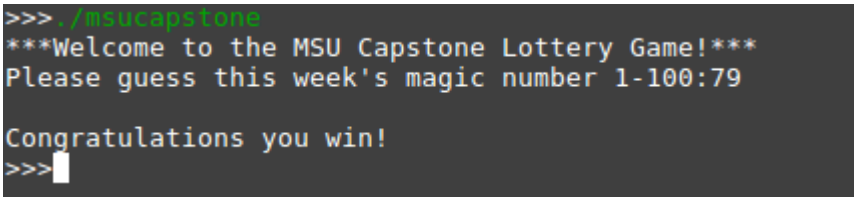

Figure 8: Run the binary again and input the magic number

Figure 9 shows the area of interest in the unknown binary using a hex editor. The column along the left marks every 16 bytes. By taking note of the instruction positions from Figure 7, the compare and jump instructions are located.

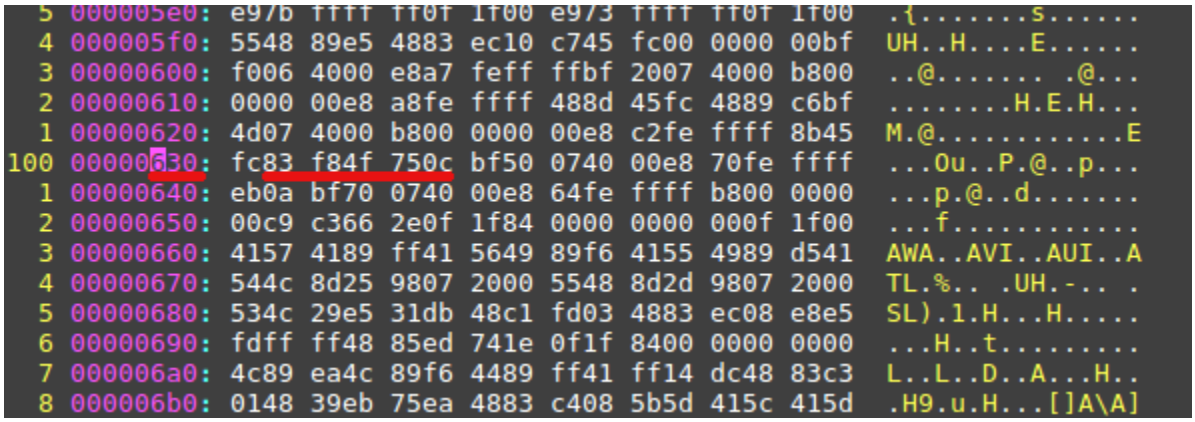

Figure 9: Locate the compare and jump instruction in a hex editor

Modifying the jump instruction from 750c (jump not equal) to 740c (jump equal) will yield behavior that is opposite to what the unknown binary was displaying previously. Figure 10 shows the edit of the instruction.

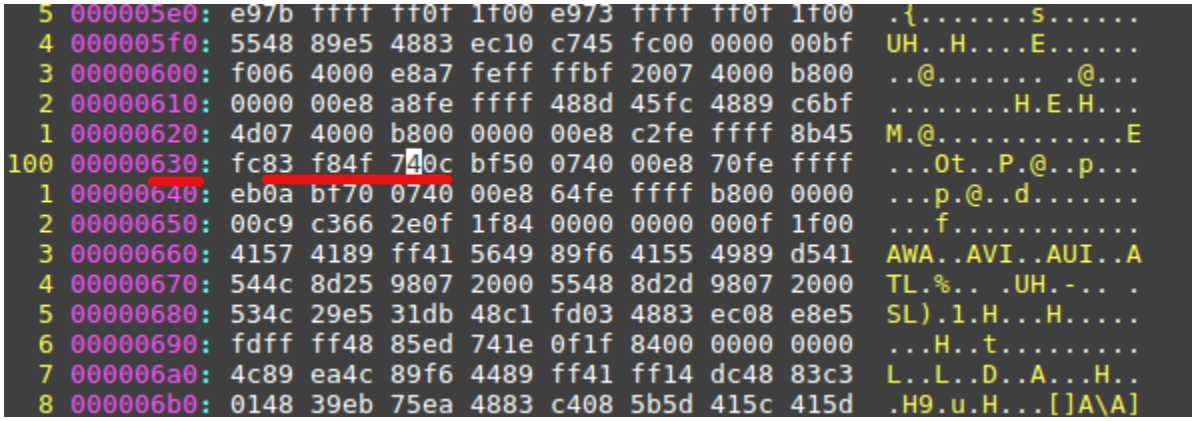

Figure 10: Modify the jump instruction to do the opposite

Running the unknown binary through objdump again shows that the instruction has changed from jne (jump not equal) to je (jump equal). Figure 11 shows the main section from the objdump output of the modified binary.

```
00000000004005f0 <main>:
  4005f0:       55                      push   %rbp
  4005f1:       48 89 e5                mov    %rsp,%rbp
  4005f4:       48 83 ec 10             sub    $0x10,%rsp
  4005f8:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  4005ff:       bf f0 06 40 00          mov    $0x4006f0,%edi
  400604:       e8 a7 fe ff ff          callq  4004b0 <puts@plt>
  400609:       bf 20 07 40 00          mov    $0x400720,%edi
  40060e:       b8 00 00 00 00          mov    $0x0,%eax
  400613:       e8 a8 fe ff ff          callq  4004c0 <printf@plt>
  400618:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  40061c:       48 89 c6                mov    %rax,%rsi
  40061f:       bf 4d 07 40 00          mov    $0x40074d,%edi
  400624:       b8 00 00 00 00          mov    $0x0,%eax
  400629:       e8 c2 fe ff ff          callq  4004f0 <__isoc99_scanf@plt>
  40062e:       8b 45 fc                mov    -0x4(%rbp),%eax
  400631:       83 f8 4f                cmp    $0x4f,%eax
  400634:       74 0c                   je     400642 <main+0x52>
  400636:       bf 50 07 40 00          mov    $0x400750,%edi
  40063b:       e8 70 fe ff ff          callq  4004b0 <puts@plt>
  400640:       eb 0a                   jmp    40064c <main+0x5c>
  400642:       bf 70 07 40 00          mov    $0x400770,%edi
  400647:       e8 64 fe ff ff          callq  4004b0 <puts@plt>
  40064c:       b8 00 00 00 00          mov    $0x0,%eax
  400651:       c9                      leaveq
  400652:       c3                      retq
  400653:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  40065a:       00 00 00
  40065d:       0f 1f 00                nopl   (%rax)
```

Figure 11: View the modified instruction with objdump

Figures 12 and 13 show that the modified binary is in fact doing the opposite of the original. 79 is the only incorrect answer, while 22 is one of the numerous correct answers.

```
>>>./msucapstone
***Welcome to the MSU Capstone Lottery Game!***
Please guess this week's magic number 1-100:22

Congratulations you win!
>>>
```

Figure 12: The previously incorrect 22 is now correct

```
>>>./msucapstone
***Welcome to the MSU Capstone Lottery Game!***
Please guess this week's magic number 1-100:79

Sorry that is not the magic number, you lose
>>>
```

Figure 13: The previously correct 79 is now incorrect

Software Technologies

Front End

- Python 3.6
- HMTL5 & CSS3
- Bootstrap CSS
- Cuckoo API
- Flask
- JQuery

Back End

- Python 2.7
- Cuckoo
- Suricata
- PostgresSQL
- SQLAlchemy
- Apache
- VMware

Development Environments

The Flask and Python 3.6 web application will be developed in various text editors such as Notepad++ and Vim. Flask's development environment will also be leveraged to develop the web application and see the changes in real time. The malware evaluation extension to Cuckoo will be written in Python 2.7 to easily interact with Cuckoo and will also be written in various text editors.

The sandbox detection and program modification software will use the Geany text editor and objdump disassembly program. For more complicated samples, a manual run-through will be performed using Interactive Disassembler (IDA Pro) These tools allow for intuitive deconstruction of executable files and are provided in Proofpoint's lab environment.

Database Schema

The solution requires the preservation of data belonging to malware samples. The Cuckoo sandbox system contains a database that stores a subset of the total set of data pertaining to malware samples. Certain data is required when displaying system/individual statistics on the dashboard.

Malware Sample Relevant Data		
Name	**Data Type**	**Description**
timeReceived	DateTime	Timestamp when sample received by system
fileHash	Blob	SHA256 of file submitted
processingState	Int	Status (pending, in progress, complete)
processingTime	DateTime	Time spent “in progress”
malwareFamily	VarChar	Malware family (if known)
isInteresting	Bool	If sample was evasive or not
numExecutionMods	Int	Number of times the sample was modified to get different results
cuckooSignatureHits	Int	Cuckoo triggers hit
suricataRuleHits	Int	Suricata triggers hit
isAnalyzed	Bool	Yes if sample checked by human
networkFlowCount	VarChar	Network information received from Suricata

Figure 14: Required Data Fields

Risk Analysis

Reverse Engineering Difficulty

Difficulty: Moderate

Importance: High

Problem: Malware samples are rarely available as readable code. In a majority of cases, they are presented as executable files. These files are of no research value and must be reverse engineered into code in order to analyze and modify them. Reverse engineering is unfamiliar to the team and is frequently known to be an unreliable process.

Mitigation: There are a variety of available tools designed for executable decomposition. Each tool has various benefits and drawbacks and there is benefit in using a multitude of tools in parallel. Proofpoint recommends the use of the interactive disassembler (IDA Pro), objdump, and/or strings. Despite not having access to Proofpoint's infrastructure until the week of 9/24, the team can download these free tools and become proficient in advance.

Multiple Language Proficiency

Difficulty: Moderate-Low

Importance: Moderate

Problem: Malware is not native to any one niche of developers; it is created with all programming languages. Some of these languages may be obscure or may be old enough to have faded away from modern language guidelines. Because there are at least several hundred programming languages in existence, it is infeasible to conduct behavior analysis on all of the samples the team receives.

Mitigation 1: This is a key area to focus on avoiding feature creep. The evasive malware solution cannot be an umbrella software - the scope is too large. To stay within reasonable bounds, the team must limit analysis to a subset of the greater universe of languages. Furthermore, handling the more obscure languages must start with smaller, simpler samples that only execute a few actions.

Mitigation 2: Despite the malware source code being in numerous languages, this will be a non-issue if the team uses the correct tools. Regardless of original language, these malware samples will share one syntax when broken down into assembly code. This is also the case for analyzing the binary itself.

Navigating Proofpoint's Lab

Difficulty: Low

Importance: High

Problem: Much of the development environment is provided by Proofpoint, with the infrastructure located on their campus. Access is granted through a remote login system. It is unknown how customizable this environment is, considering any necessarily changes to the project. Proposed changes may be declined, or may result in significant delays to the project timeline.

Mitigation 1: Brad has created a Secureshare service and will run any samples the team uploads through the lab during the period where access is still being provided. This will give the team results necessary to build other portions of the project, but will not help in establishing dependencies and becoming familiar with the environment.

Mitigation 2: Yash can be another invaluable asset regarding Proofpoint's environment as he experienced this exact risk less than a year ago. The environment used is a continuation of that used in Spring 2018, so Yash will be helpful in navigating Proofpoint's Cuckoo and VM configurations.

Difficulty of Finding Evasive Checks in Large Malware Samples

Difficulty: Very High

Importance: High

Problem: The assembly files of malware samples often grows into the tens of Megabytes, with tens of thousands of lines of code. Somewhere in this code is one or more crucial "jump" statements, where the evasive malware will determine whether or not to execute. Since these accompany thousands of benign jumps, finding the right ones is challenging.

Mitigation: Many of the samples available have been previously identified by Proofpoint as evasive. Starting with these samples allows the team to avoid trying to identify evasive code in non-evasive samples. The team can also use visual disassembler such as IDA Pro to find the syntax of evasive checks. Once the relevant code is located, it will be much easier to recognize similar actions in other, more complicated samples.

Schedule

Week 1: **8/29-9/2**

- Initial Team Assignment
- First Team Meeting
- Established Methods of Communication (Team & Corporate Contact, Git Repo)

Week 2: 9/3-9/9

- First Client Meeting
- Research & Establish Relevant Technologies (Cuckoo, Suricata, IDA Pro)
- Proofpoint Technical Demonstration

Week 3: 9/10-9/16

- Status Report Document & Presentation
- Project Plan Rough Draft Completed
- Basic Web Page Completed
- Client's Pseudo-Malware Programs Created
- Client's Initial IDA Assignment Completed

Week 4: 9/17-9/23

- Combined Pseudo-Malware with IDA Pro Disassembler
- Project Plan Final Draft Completed
- Web UI Mockups Completed
- Design Day Booklet First Revision
- Cuckoo REST API Skeleton Program (Front-end)

Week 5: 9/24-9/30

- Project Plan Presentation
- Design Day Booklet Second Revision
- Evasive Check Dictionary Created (e.g. jne, jg, je & Respective Hex Codes)
- Set up Proofpoint Environment & Cuckoo Configuration
- Actual Web UI Design Completed
- First Live Website Deployed
- Informal Proofpoint Malware Sample Analysis

Week 6: 10/1-10/7

- Analyze First Evasive Sample(s)
- Web Interface Compatible with hard-coded input
- Formal Proofpoint Malware Sample Analysis
- Python Script to Modify Basic Malware Completed
- Python Script to Detect Evasive Malware Prototype
- Alpha Presentation Preparation
- Cuckoo Server (REST API) Fully Integrated with Web Server

Week 7: **10/8-10/14**

- Malware Modification Module Improvements & Refactoring
- Malware Detection Module Completed
- Advanced Modification Algorithms Proposed & Feasibility Testing
- SQLALchemy Implemented for Web Application
- Web UI Refactoring

Week 8: 10/15-10/21

- Alpha Presentation
- Finalized Decision of Advanced Modification Algorithm
- Postgres Sever Online
- Top Signatures Widget Online

Week 9: 10/22-10/28

- Progress on Implemented Malware Modification Module Advanced Algorithm
- System State Widget Online
- Recent Modifications Widget Online
- Recent Submission Widget Online
- Web UI Improvements
- Advanced Malware Processing Beginning

Week 10: 10/29-11/4

- Finalize Malware Modification Module with Advanced Algorithm
- Project Plan Revisions
- Web UI All Widgets Online
- Camtasia Familiarization & Research

Week 11: 11/5-11/11

- Video Storyboarding
- Start Video Work
- Stress Test Project with Advanced Samples
- Web Interface Finalization & QoL Edits
- Beta Presentation Preparation

Week 12: 11/12-11/18

- Beta Presentation
- Detonator Module Complete & Refactored
- Video Filming

Week 13: 11/19-11/25

- Video Filming and Editing

Week 14: 11/26-12/2

- Project Video Finished

Week 15: 12/3-12/9

- Design Day Setup
- Deliverables Due
- Design Day