Detection of Malicious Remote Shell Sessions

Semester Thesis

Pierre Dumont
pierredumont@student.ethz.ch

supervised by
Dr. Vincent LENDERS (armasuisse)
Roland MEIER (ETH Zürich)
Dr. David GUGELMANN (Exeon Analytics)
Prof. Laurent VANBEVER (ETH Zürich)

October 30, 2017
SA-2017-87
Abstract

In this project, we design, implement, and test a classifier to detect malicious remote shell sessions. Using honeypots and data available on the Internet, we train a classifier to discriminate between malicious and non-malicious commands executed on a Bash terminal. We develop a modular log analyzer framework to support multiple input data formats. This framework is used for differentiating between two sources of sequences of commands: a Cowrie honeypot and Bash history files. Using a k-Nearest Neighbors algorithm, we build classifiers which can operate on single shell commands or on consecutive commands. We evaluate their performances and the feasibility to use our classifiers to detect malicious activity in remote shells. Our evaluation shows that our classifier is capable of reaching a true positive rate of 99.41% and a true negative rate of 99.69% after observing 4 commands.
Contents

1 Introduction 3
   1.1 Related Work .............................................. 3
   1.2 Contributions ............................................. 4
   1.3 Structure .................................................. 4

2 Goals 5

3 Data Sources 6
   3.1 Bash History .............................................. 6
   3.2 Cowrie ..................................................... 6

4 Log Analyzer 8
   4.1 Launcher .................................................. 8
   4.2 SSH Log Analyzer ......................................... 10
      4.2.1 SSH Log Formats ................................... 10
   4.3 SLA Modules .............................................. 16

5 Classifying Sessions 20
   5.1 Features .................................................. 20
   5.2 1-Command Classifier .................................... 21
   5.3 N-Commands Classifier ................................... 21
   5.4 Log Analyzer Procedures ................................ 21
      5.4.1 Selecting Input Data ............................... 23
      5.4.2 Generating the Training Data Features .......... 23
      5.4.3 Creating the k-NN Model ......................... 23
      5.4.4 Generating the Test Data ....................... 23
      5.4.5 Testing the Model ................................ 24

6 Results 25
   6.1 1-Command Classifier .................................... 26
   6.2 2-Commands Classifiers .................................. 31
   6.3 N-Commands Classifiers .................................. 34

7 Conclusions and Future Work 38
   7.1 Conclusions .............................................. 38
   7.2 Future Work ............................................. 38
1 Introduction

With the exponential increase of devices connected directly to the Internet, we are seeing a spike in compromised machines which are often part of a bigger network, a botnet. One of the biggest recent examples is the Mirai botnet which took down DynDNS, one of the most used DNS service providers in the US [1] a year ago. This malware infected over 600,000 IoT devices at its peak, by taking advantage of systems using Telnet and SSH secured with default passwords [2]. Since then, the source code was published on Github [3] and many variants exist in the wild. Security researchers have been using honeypots to analyze malwares by providing an attackable system by hackers.

Providing remote shell access to system administrators greatly helps them to configure and maintain servers which might be physically distant. Managing those systems is a complex task and, by convenience or inadvertence, they might be accessible remotely. Using the Shodan search engine, we notice that over 16 million devices have SSH enabled and are accessible over the Internet [4]. Regarding Telnet the amount of machines allowing to connect is less than 150 thousands according to Shodan [5]. From the established connections we see daily in our honeypots and we can safely assume that most of the Internet-facing SSH servers are under the same attacks that we are. After previous work done on this subject at armasuisse, the bots attacking are not only testing if they connect, but they are also trying to take control over it. If those systems are compromised, the cost for their owners due to data loss, downtime, loss of reputation etc. could be devastating for companies as well as for individuals.

With the recent threats of WannaCry [6], NotPetya [7], etc. this year, we saw leaked 0-days exploits used to compromise networks of big companies that should have been patched. A common phrase used in Information Security is “it is not a question of if you are hacked but when”. When an attacker has breached the defenses and is in the internal network, we need defensive tools to detect intrusions such as Intrusion Detection Systems (IDS). Since SSH is one of the most popular protocol, we can expect hackers to move laterally by connecting via SSH with stolen credentials to compromise more machines. Detecting and blocking rapidly threats is of utmost importance for IT security teams in all companies.

In this project, we aim to build a binary classifier capable of detecting malicious commands being executed on a SSH server as quickly as possible. This system will provide intrusion detection (IDS) capabilities on shells to solve those issues.

1.1 Related Work

Studying SSH to distinguish between users has already been done in [17] as early as 2001. In their work, the team from Berkeley, study the users’ typing pattern to guess the users’ passwords. Since then, many papers analyzed SSH from a network perspective such as SSHCure [18] using Hidden Markov Model [19]. Those papers mainly focus on the threat of SSH brute-forcing.

However, SSH servers will get compromise and several papers [20] [21] have already examined how the attackers operate when they connect to the authors’ honeypots. Nowadays, the threats on the Internet are coming from botnets and most of the interactions with the honeypots are not advanced.

For a broad survey of machine learning classifiers to detect malicious code, the authors of [23] presented various algorithms to classify malicious executables files. Many of those techniques can be found in other papers such using n-grams in [24] or in [25] where the authors used n-grams as features to a Support
Vector Machines (SVM) algorithm to classify XSS (Cross-Site Scripting) and SQL Injections.

One important work that inspired this project and our previous analysis on this subject is the work of this paper [22] where the authors investigated the behavior of operators of a popular Remote Access Trojan (RAT) called DarkComet. They analyzed hundreds of sessions where human attackers decided to use the webcam, explore the system and gather passwords file, installing keyloggers, etc. The authors give us an thorough insight into amateurs’ actions on compromised systems.

To the best of our knowledge, this work is the first that looks into the problem of classifying malicious remote shell commands based on the contents of these commands.

1.2 Contributions

The contributions of this thesis with regards to detecting and classifying remote shell sessions from two sources, one malicious and the other non-malicious, are the following:

- We build a modular and extensible framework which accepts multiple log formats, can analyze various aspects of a shell session and outputs several formats of data (plots, features files, csv files, etc.). We created log formats for logs formatted by Cowrie honeypots, for plain Bash History files and for general csv formatted file type.

- We provide this system with multiple modules to produce the intermediary steps needed to create our command classifier. The modules include: analyzing the source IPs, the time elements of a session (for Cowrie logs), generating different features characterizing each sessions, creating a $k$-Nearest Neighbor classifier model, testing and outputting results of a $k$NN model and many more.

- Finally, we evaluate the performance of remote shell session classifiers using real data from a Cowrie SSH/Telnet honeypot and the bash history of Github developers.

1.3 Structure

The structure of this report is the following: we start by talking about the background and the goals of this project in section 2, and the solutions we designed to solve this problem. Afterwards we present the available data sources that we want to be able to distinguish between with our classifier. Then we dive into a technical description of the log analyzer framework we built in section 4 with its different components. Section 5 is dedicated to the detection of malicious sessions where we describe the 1-Command classifier and the N-Commands binary classifier. In section 6, we analyze the results of the classifiers using formulas such as the true positive rate (TPR) or the Receiver Operating Characteristics (ROC). Finally, we discuss discoveries we made and what still needs to be accomplished to have a fully deployable system.
2 Goals

The main goal of this work is to develop a method to detect as quickly as possible if the operator connected to our SSH server is either an attacker or an administrator. The system we will build whose purpose to differentiate between commands from two available data sources: sessions from a Cowrie SSH honeypot and Bash history files publicly available. Those sources represents our malicious and non-malicious SSH operators. We imagine our system running in the internal network of companies as an IDS, constantly monitoring the SSH servers. Those would send each command for analysis to our detection system.

When a threat is detected by our monitoring tool on a SSH server with a high probability, the session would be terminated and all previously typed commands would be sent for analysis. In addition, IT security teams will be alerted and the threat isolated.

The two main challenges are the following:

1. How can we reliably differentiate between a malicious session from a non-malicious one knowing that attackers tend to be unpredictable and commands can be ambivalent in their purpose until enough are captured.

2. How quickly can we decide if a sequence of commands is malicious or benign?

Another important aspect is that the false positives rate must be very low (high true negative rate): in real situations, most of the commands will come from an administrator and classifying administrative actions as malicious will cause troubles to the authorized user for their daily work. Therefore, minimizing false positives is one of our priorities to make the classifier usable.

With this type of classifier, we conjecture that the results of our models will allow us to use the developed system in the future as a real-time remote shell session IDS solution.
3 Data Sources

To build a classifier between malicious and non-malicious interactions, we rely on a supervised machine learning technique which creates a model of typical SSH administrator and malicious commands from historical data. In this section, we describe the sources of the used data for training and testing our model.

3.1 Bash History

Bash history files contain the last commands typed by a user in a bash shell saved in the home folder as a hidden .bash_history file. Since it is unlikely that a user would enter malicious commands on his system, we suppose that the commands in those files are non-malicious (administrative, programming, code testing, maintenance, updates, etc.). Our source of .bash_history files is Github: using the search function, we found thousands of .bash_history files [8]. Note that this search must be done as a logged-in user, otherwise, it will not return any results.

Github provides an API [9] which allows us to search and download code directly. For this purpose, a download script (github_downloader.py) was created. Using several search queries, we managed to download more than the maximum search limit of 1'000 results. This allowed us to downloaded 3'156 .bash_history files, totaling 20.7 MB. After using the module Commands described in section 4.3, we find that out of those 3'156 files, 3'146 are not empty, and contain in total 973,621 commands with 234,063 unique commands, which is quite diverse.

It’s interesting to notice that the top 7 commands are the following common ones:

1. ls, 102187, 10.4956%
2. cd .., 21319, 2.1897%
3. exit, 12996, 1.3348%
4. ll, 12512, 1.2851%
5. clear, 11980, 1.2305%
6. git status, 9862, 1.0129%
7. cd, 7882, 0.8096%

3.2 Cowrie

Our other source of data is Cowrie: Cowrie [10] is a SSH/Telnet medium-interaction honeypot based on the Kippo honeypot. Some of the features are:

- Logs (JSON format) all logins attempts and commands executed.
- Saves the downloaded files (from wget, curl, etc.).
- Provides a fake filesystem resembling Debian 5.
- Possibility of adding fake content.
- Logging of SSH proxying.

We are using a dockerized version of Cowrie [11] which is part of the T-Pot system [12] that was setup several months ago.

The files are located in /data/cowrie/ and the structure of the files is the following:
- `log/`: all the logs files, the interesting ones are the `cowrie.json` files. Note the interesting format of the date (2017.1.14).

- `downloads/`: contains the files downloaded from commands (wget, curl, etc.). The file name is the SHA256 hash of the file’s content.

- `keys/`: contains the host RSA and DSA keys.

- `misc/`: contains `userdb.txt`, list of usernames and passwords accepted.

By default, this honeypot lets an attacker in after a random number of login attempts between 2-5 and caches the last 10 attempts. If the username-password provided is in the cache, the login attempt is successful regardless of the random number of attempts required. We can change the configuration in `cowrie.cfg` to allow only certain logins using the `userdb.txt` file.

After previous work done on the analysis of logs, we concluded that most of the malicious traffic originated from bots using simple malware tools or well-known threats.

**Data Logged**  Cowrie is logging a huge amount of interesting data which we could use to derive features to classify malicious remote shell sessions. Here is an example of a command logged by Cowrie:

```json
{
   "eventid": "cowrie.command.input",
   "timestamp": "2017-08-23T00:01:54.647122Z",
   "session": "9869f0a",
   "message": "CMD: enable",
   "isError": 0,
   "system": "CowrieTelnetTransport,70,xxx.xxx.xxx.xxx",
   "src_ip": "xxx.xxx.xxx.xxx",
   "input": "enable",
   "sensor": "t-pot"
}
```

However, there are plenty more columns, some of the interesting ones are:

- **message**: contains information on what happened, depends on the type of event (present in all events).

- **password**: password tried (only in login.success, failed events).

- **sensor**: will be used for distributing cowrie sensors (t-pot).

- **session**: unique session id (example: a0d8a673) (present in all events).
4 Log Analyzer

One of the main contributions of this work is the Log Analyzer (LA) framework. After previous work done on log analysis, the need for a general, modular and extensible approach was necessary to complete this project and it defined the LA. This format allowed us to quickly add processing modules and log formats during the course of this project when needed.

The Log Analyzer is composed of multiple parts each described in their own section, the entry point to the software is the Launcher (4.1), this decides which type of Logs (4.2), which Log Format (4.2.1) and which Modules (4.3) to use with their respective arguments. In each section a diagram provides a high level description of the software. This project is being developed on Python 3.6.2 by using the latest versions of multiple libraries such as numpy 1.13.3, Pandas 0.20.3, Scikit-Learn 0.19, etc. provided by the anaconda distribution version 3.5 [13].

It is well known that analyzing data can take a lot of time and resources to produce the desired results. For this goal, multi-processing is used as often as possible throughout the different parts of the project as well as using multiple intermediary steps to produce final results.

4.1 Launcher

The Launcher is the main entry point to the core of the program. It is used to specify the behavior of the application. The algorithm can be seen in Figure 1 and all the possible variables are described in the tables below. More information can be found in the Appendix D.

There are three modes available to run the software, specifying one of them is necessary. How to specify the input is explained in table 1.

- input_file: in this mode, any specified input files are imported into one Pandas DataFrame and the processing is applied on it as if it was one file.
- input_file_list: in this mode, the software is called on each input file individually.
- input_folder: in this mode, each file in the folder is processed one-by-one as in input_file_list. However, when the processing is finished, it will monitor the folder and when a new file appears, the Log Analyzer will process it.

Table 1: Available modes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Short</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_file</td>
<td>-i</td>
<td>-i file1.log [file2.log ...]</td>
<td>Use the specified log file as the input.</td>
</tr>
<tr>
<td>input_file_list</td>
<td>-il</td>
<td>-il file1.log [file2.log ...]</td>
<td>For this mode, all input files will be executed &quot;one-by-one&quot;.</td>
</tr>
<tr>
<td>input_folder</td>
<td>-if</td>
<td>-if folder/</td>
<td>This mode will process individually all logs in the folder then monitor it for new files.</td>
</tr>
</tbody>
</table>

Secondly, we need to specify which log type with which format and which modules to use. All of those are explained in table 2. As we seen in diagram 3, only the SSH Log Analyzer (SLA) has been developed currently. The log formats available are: "BashHistory", "Cowrie” and "TextFiles". The first one being for .bash_history files, the second one are log files generated by the honeypot Cowrie.
Figure 1: Launcher of the Log Analyzer
The TextFiles format represents any csv-like file, most of them generated by modules.

The modules are launched in three possible phases, each waiting until the previous one finished. The modules, their arguments and in which phase to use them are described in their section 4.3.

Note that the log analyzer, log format and modules can all have special arguments which are described in the Appendix D.

The remaining arguments are related to the output (table 3). We can specify the output folder which will be created if needed. When the appropriate modules are run, subfolders containing the name of the log analyzer and the log format are created (e.g.: SLACowrie) and in those, the output folders Features/, Logs/, Pickles/, Plots/, Status/ and TextFiles/ are generated.

Most of them are self-explanatory except Status/: in this folder, .status files are generated. Those are used to tell the software if a module was already run on the specified input files. If the flag overwrite_outputs is not set, a module previously executed will not be re-executed. This is useful for example, if the monitor folder mode is stopped then restarted to avoid re-executing modules.

One common problem faced during analyzing BashHistory files is related to the output names: how do we know from which input files does the output files come from? The solution was to use hashes to represent input files. In the folder output/Hashes/, text files are generated which contains pairs of hashes and list of input file names.

Another possibility is to use the -on flag to specify an output nametag which will be part of the output filename. This is the recommended option.

Basic checks are done in the Launcher but most of the argument testing are done in the Log Analyzer.

4.2 SSH Log Analyzer

As the name implies, the SSH Log Analyzer (SLA) is the core class to analyze all types of SSH related logs. The goal is to provide the common code to all log formats in this class which are currently BashHistory, Cowrie and TextFiles.

The main algorithm of the SLA and all log formats is the following and can be found in Figure 2:

1. Setup the used variables
2. Checks the arguments for the Log Analyzer, the Log Formats, the modules and their arguments.
3. Decide which mode to use.
4. Accordingly, start the pipeline on the correct files. The pipeline includes:
   (a) Importing the files
   (b) Cleaning the files
   (c) Starting the modules with the appropriate arguments

4.2.1 SSH Log Formats

Currently, three log formats (LF) have been developed for SLA: BashHistory, Cowrie and TextFiles (Figure 3).

Most of the methods and attributes are declared in the parent class: SLA. The only functions specific for the log formats are the importing and the cleaning.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Short</th>
<th>Required</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_analyzer</td>
<td>-la</td>
<td>False</td>
<td>-la SLA</td>
<td>Specifies which log analyzer to use. Only SLA is available currently.</td>
</tr>
<tr>
<td>log_analyzer_arguments</td>
<td>-laa</td>
<td>False</td>
<td>-laa arg=value</td>
<td>Gives specific arguments to the Log Analyzer. Currently SLA does not take any special arguments.</td>
</tr>
<tr>
<td>log_format</td>
<td>-lf</td>
<td>True</td>
<td>-lf Cowrie</td>
<td>Specifies one of the three currently available log format.</td>
</tr>
<tr>
<td>log_format_arguments</td>
<td>-lfa</td>
<td>False</td>
<td>-lfa arg=value</td>
<td>Gives specific arguments to the Log Analyzer. See in each log format sections for the available arguments.</td>
</tr>
<tr>
<td>modules</td>
<td>-m</td>
<td>True</td>
<td>-m path.module</td>
<td>Specifies which modules to use in phase 1. The argument &quot;all&quot; can be used.</td>
</tr>
<tr>
<td>modules_args</td>
<td>-map</td>
<td>False</td>
<td>-map arg=value</td>
<td>Gives specific arguments to the modules. Note that if multiple modules are used, you must specify path.modules:arg=value</td>
</tr>
<tr>
<td>modules_phase2</td>
<td>-m2</td>
<td>False</td>
<td>-m2 path.module</td>
<td>Same as -m but for phase 2.</td>
</tr>
<tr>
<td>modules_args_phase2</td>
<td>-map2</td>
<td>False</td>
<td>-map2 arg=value</td>
<td>Same as -map but for phase 2.</td>
</tr>
<tr>
<td>modules_phase3</td>
<td>-m3</td>
<td>False</td>
<td>-m3 path.module</td>
<td>Same as -m but for phase 3.</td>
</tr>
<tr>
<td>modules_args_phase3</td>
<td>-map3</td>
<td>False</td>
<td>-map3 arg=value</td>
<td>Same as -map but for phase 3.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Short</td>
<td>Default</td>
<td>Example</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>---------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>output</td>
<td>-o</td>
<td>output/</td>
<td>-o test_output/</td>
<td>Specifies the output folder. Will be generated if it does not exist.</td>
</tr>
<tr>
<td>overwrite_outputs</td>
<td>-ow</td>
<td>False</td>
<td>-ow</td>
<td>Ignores the status files. Modules will be re-run on the specified input and outputs will be overwritten.</td>
</tr>
<tr>
<td>features</td>
<td>-of</td>
<td>True</td>
<td>-of False</td>
<td>Outputs the features or not.</td>
</tr>
<tr>
<td>plotting</td>
<td>-op</td>
<td>True</td>
<td>-op False</td>
<td>Outputs the plots or not.</td>
</tr>
<tr>
<td>text_files</td>
<td>-ot</td>
<td>True</td>
<td>-ot False</td>
<td>Outputs the text files or not.</td>
</tr>
<tr>
<td>output_nametag</td>
<td>-on</td>
<td>None</td>
<td>-on testname</td>
<td>Specifies a user specified nametag that will be used in the output filename.</td>
</tr>
<tr>
<td>verbose</td>
<td>-v</td>
<td>False</td>
<td>-v</td>
<td>Increases the verbosity in the output logs</td>
</tr>
</tbody>
</table>
Figure 2: SSH Log Analyzer (SLA) Core algorithm
Figure 3: Log Formats available for the SSH Log Analyzer (SLA)
**BashHistory** is the format for .bash_history files. As explained in section 3.1, those files come from Github and have no formatting except one line per command. This module accepts special arguments explained in the Appendix D. The following modules with their arguments can be launched at the different phases for this format. Each modules is explained in their respective sections.

- **Phase 1:**
  - CommandsBashHistory, argument: "top".
  - GenerateNGramsBashHistory, arguments: "ngrams_top", "ngrams_num".
  - NPairsCmdsBashHistory, arguments: "n_pairs", "preprocess", "sliding", "only_firsts".
- **Phase 2:**
  - FeaturesNGramsBashHistory, arguments: "ngrams_file", "label", "o_nametag_ngrams".
- **Phase 3:**
  - None.

**Cowrie** this Log Format comes from the json-formatted logs generated by the Cowrie honeypot. The following modules with their arguments can be launched at the different phases for this format. Each modules is explained in their respective sections.

- **Phase 1:**
  - CommandsCowrie, arguments: "top", "preprocess".
  - CountEventTypes, arguments: None.
  - GenerateNGramsCowrie, arguments: "ngrams_top", "ngrams_num", "preprocess".
  - NPairsCmdsCowrie, arguments: "n_pairs", "sliding", "only_firsts", "preprocess".
  - SessionsDurations, argument: None.
  - SessionsStartEnd, argument: "sampling_rate".
  - SourceIPs, argument: "top".
- **Phase 2:**
  - FeaturesNGramsCowrie: "ngrams_file", "label", "o_nametag_ngrams", "preprocess".
- **Phase 3:**
  - None.

**TextFiles** this Log Format refers to any csv-like formatted file. Most of those files are outputs of other modules. This module accepts special arguments explained in the Appendix D. The following modules with their arguments can be launched at the different phases for this format. Each modules is explained in their respective sections.

- **Phase 1:**
  - MergeFiles, argument: "keep_top"
  - MLkNNTest, arguments: "model", "threshold", "thresholds_steps".
• MLkNNTrain, arguments: "test_file", "knn_val", "up_to_k".

• Phase 2:
  • None.

• Phase 3:
  • None.

4.3 SLA Modules

Many modules were developed for this project (Figure 4). All of them must inherit from the parent class: SLAModule where the core of the main algorithm resides. Since for this project developing the SSH Log Analyzer was the main focus, most of the modules were designed for the SLA. However, modifying the parent class to a more generic Module class should not be too complicated.

The algorithm follows this procedure and can be found in Figure 5:

1. Setup the used variables in the __init__ function and call the run() method
2. The run() method calls:
   (a) pre-processing()
   (b) processing()
   (c) post-processing()
   (d) output() which calls
      i. output_features()
      ii. output_plots()
      iii. output_texts(...raw=False...)
      iv. output_texts(...raw=True...)
      v. output_texts_daily()
      vi. write_status()

Commands Analyzing the commands is one of the most valuable analysis possible. In this module, commands, words, IPs, etc. are processed and analyzed. Outputs range from top words, plotting top 25 commands, looking at IP addresses in commands, etc. Those values are saved in the FeaturesTable for Cowrie sessions.

CountEventTypes Cowrie logs the attackers’ interaction with the honeypot using different event types. Examples: cowrie.session.connect, cowrie.session.closed, cowrie.command.input, cowrie.login.success, cowrie.command.success, etc. This module goes through the log and for each session, counts each event type and total number of events. Those values are saved in the FeaturesTable for Cowrie sessions.

FeaturesNGrams This module uses an n-grams file to generate a NGrams-FeaturesTable. For each commands generate its n-grams and for each of those, check if it is present in the n-grams file, then count the occurrences. This module should be used in phase 2 since it relies on the n-grams file generated by modules.GenerateNGramsCowrie. However, for Cowrie only, if the n-grams file does not exist, this module will try to generate it.
Figure 4: SSH Log Analyzer (SLA) Modules available
Figure 5: SSH Log Analyzer Modules Core algorithm
GenerateNGrams  This module generates the top \( t \) \( n \)-grams for the commands from the Cowrie logs. The \( n \) parameter is used as "up to \( n \)"-grams, meaning that if \( n = 3 \), the \( n \)-grams will be generated for \( n = 1 \), \( n = 2 \) and \( n = 3 \) and for each of those, the top \( t \) are kept. The commands are split into "atomic" commands, e.g.: any commands with semicolons and logical operators are split into multiple commands, currently only done for Cowrie. The tokenizer is splitting by spaces, no punctuation is taken into account to generate the \( n \)-grams.

NPairsCmds  This module generates pairs of length \( n \) of commands. This is used to generate the input to the N-Commands classifier. One option allows to produce sliding pairs or not. Note that this module can also be used for \( n = 1 \) to generate a list of commands.

MergeFiles  It merges any file specified in the input but this is done in the LF TextFiles so this module "does nothing" except outputting the merged df. If -if is used, the content of the folder are merged together. -i and -il are equivalent, files specified are merged together. The input is any "csv" formatted text files. Note: SLATextFiles supports the arguments: "delimiter","header" and "strict", "drop_duplicates", "add_count" like this, default: "fa delimiter=, header=True strict=True drop_duplicates=None add_count=None. See section 4.2.1 and the Appendix D for more information.

MLkNNTrain  This module generates the \( k \)-Nearest Neighbors model using the scikit-learn library [14]. It saves the model which can be used later on with the MLkNNTest module in the Pickles/ folder. If -map test_file=ngrams_features.txt is specified, the testing phase is also applied, as if it would have been done in MLkNNTest. Note that model ranges from a few hundreds of Megabytes to multiple Gigabytes depending on multiple factors. The input is a features table, NGramsFeaturesTable only currently. It needs to contains a column label, columns session and cmd are optional and will be removed from the features. The other columns will be considered as the features.

MLkNNTest  This module needs a \( k \)-NN model specified with the -map model=MLkNN.pickle argument. It tests the input NGramsFeaturesTable on the model and produces output texts and plots. The model file is located in SLATextFiles/Pickles/MLkNN/. Note -map model is optional, using -on name can be used. This will use the default path and filename with the output_nametag specified by -on. One of those two must be used, it is recommended to use -map model.

SessionsDurations  Cowrie writes in the event "cowrie.session.closed" in the column "duration", the duration of each session. An empirical cumulative distributive function plot is generated from this data. The duration is placed as a feature in the FeaturesTable (not the NGramsFeaturesTable) for Cowrie sessions.

SessionsStartEnd  Cowrie logs contains timestamps for each event. The session start at the event "cowrie.session.connect" and stops at the event "cowrie.session.closed". Both events' timestamps are analyzed and plotted in a time series plot and are used as features in the FeaturesTable.

SourceIPs  Cowrie logs contains the source ip for each session. This is analyzed in this module and text files as well as plots are generated. The source ip as well as multiple subnets are saved as features in the FeaturesTable.
5 Classifying Sessions

The goal of this project is to differentiate between malicious and benign sequences of commands. For this purpose, multiple modules were created to help generate the necessary data and were briefly described in section 4.3. In this part, we will describe the procedure to reach our goal. First, we are going to investigate which features might be useful to distinguish between the two sources possible of the commands. Secondlly, we describe the machine learning techniques used in this work of the 1-command classifier. Thirdly, we move from classifying one command to classifying a sequence of commands with the N-command classifier. The last part is dedicated to how to use the Log Analyzer to do this computation.

5.1 Features

Deciding which features to use is a complex task which depends heavily on the data used. Since the type is commands, we are going to use Natural Language Processing (NLP) techniques to create the features. Treating the commands as words, we can think of many interesting features to analyze:

- number of words
- number of letters
- number of special characters used in commands such as semi-colon, dash, etc.
- IPs in commands
- top words
- etc.

We are going to use a Natural Language Processing (NLP) technique called n-grams as features. The 1-gram being a single word, the 2-gram (or bigram) being two contiguous words, etc. With this NLP tool, we will decompose the commands into n-grams and keep a certain amount of the most used ones. For example, the following command decomposes into the following n-grams:

```
ls -lah file.txt file2.txt
```

- For 1-gram: ls, -lah, file.txt, and file2.txt : all individual words
- For 2-grams: ls -lah, -lah file.txt, and file.txt file2.txt : all ordered pairs of length 2
- For 3-grams: ls -lah file.txt and -lah file.txt file2.txt : all ordered pairs of length 3
- For 4-grams: ls -lah file.txt file2.txt : all the words.

We decided to use the n-grams as features for our commands since they are representing at the same time, the most commonly used words, commands, parameters, IPs, etc. and also preserves the order of the commands. In real situations, deciding if one command is malicious or not is a hard task since most of the commands could be used for administrating a machine and compromising it. However, using multiple commands from a sequence, gives us a lot more information about the maliciousness or not of the user. We conjecture that with a few commands, we are able to differentiate correctly between both sources. Multiple papers using n-grams to detect malicious code confirms this [24][25][26] as well as our results in section 6.
Generating the top most common \( n \)-grams is done by the module: GenerateNGrams in section D.3.3, where we can decide the top and the \( n \) parameter. The \( n \) parameter decide up to how many \( n \)-grams we will use.

Each of those \( n \)-grams will be a feature for all the commands. To generate the feature table, we use the FeaturesNGrams module (section D.3.3) to which we must give the \( n \)-grams table generated. Each of the command to test, is split into \( n \)-grams which are compared to the features. If the \( n \)-grams is present in the columns, its occurrence is written down in a row.

5.2 1-Command Classifier

The 1-Command classifier works on a single command. We expect the 1-Command classifier to be limited in performance since it is hard to detect an attacker on a single command. Considering the application of the real-time intrusion detection, the goal is to detect an intrusion as fast as possible. Creating the model is done with the MLkNNTrain module for multiple values of \( k \) and testing the performance in the MLkNNTest module.

One of the algorithm used for classification is the \( k \)-nearest neighbor (\( k \)-NN) algorithm. In \( k \)-NN, a command is assigned to the label (BashHistory or Cowrie) of the \( k \) nearest commands. Scikit’s implementation "simply stores instances of the training data" [16] and the predict function searches for the closest features vector of the testing data features vector. This is why the model’s size can reach multiple Gigabytes. Note that the documentation warns us : "if two neighbors, neighbor \( k + 1 \) and \( k \), have identical distances but different labels, the results will depend on the ordering of the training data" [16] which might induce "rounding" errors.

This algorithm needs two sets of data: the training data and the testing data. Both needs to be different as we will train the model on one set and test the model with the other.

5.3 N-Commands Classifier

Moving from a 1-command classifier, we start by testing a 2-commands classifier with the same pipeline as before. We reuse the pipeline because we believe this approach will provide better results as the order of the commands is of the utmost importance to classify if a sequence of commands is malicious or not.

Using the module NPairsCmds will be used to split sequences of commands into pairs of length \( N \) of commands. Afterwards, we can use the same techniques as for the 1-Command classifier generate \( n \)-grams from the training set, then create the features table for both sets, and finally generate and test the model. Note that the module NPairsCmds create pairs of sliding commands, e.g: command 1 and 2, 2 and 3, 3 and 4, etc. for pairs of length \( N = 2 \). This increases the number of commands and the processing time.

The same approach can be used to generate the model for any sequences of \( N \) commands. We are going to use the module NPairsCmds to split sequences of commands into pairs of length \( N \) then use the pipeline as for the 1-Command classifier.

5.4 Log Analyzer Procedures

This section gives an overview of how to use the SSH Log Analyzer to perform the Machine Learning algorithm. In Appendix B, our pipeline tool can be found for the 1-Command classifier and the N-Commands classifier. Each steps calls the Log Analyzer with different modules, arguments, inputs and outputs. The diagram in Figure 6 gives an overview of the full pipeline.
Figure 6: 1- and N-Commands Classifier Pipeline
5.4.1 Selecting Input Data

The first step is to select appropriately data from our two sources. To produce good quality features, we select at least as much BashHistory unique commands as from Cowrie. Each source will contribute to approximatively 50% of features. To select an appropriate amount of BashHistory files, we need to know how many commands from Cowrie logs we are using. For this purpose, the module Commands was developed and it outputs the number of commands, the number of unique commands and more.

By experience, selecting thirty days of Cowrie logs gives us approx. 3’000 unique commands. Afterwards, we select an equivalent (or more) amount of BashHistory logs with the following modules:

1. Using the module CommandsBashHistory on logs and by trial and error.
2. Using the script split_folder.py which can split a folder of bash history files into folders containing a defined (approx.) amount of commands.
3. Another option is to run the module CommandsBashHistory on all the available bash history files individually, then use the MergeFiles module with the appropriate parameters to keep a certain amount of commands.

For the N-Commands classifier, we generate the pairs of length $N$ beforehand with the appropriate NPairsCmds module.

5.4.2 Generating the Training Data Features

The next step is to generate the top $n$-grams with GenerateNGrams module for both data sources: BashHistory and Cowrie, then merge both files (module MergeFiles). This will give us the columns for the features table.

Afterwards, we run the FeaturesNGrams module with the $n$-grams file on both training data. This step is one of the most time consuming steps since there are thousands of commands to run through the following algorithm with hundreds, if not thousands of features:

1. For each command, if it was not processed earlier:
   2. Create a dictionary from the features $n$-grams.
   3. Decompose the command into $n$-grams.
   4. For each of those $n$-grams, check the dictionary with all the features $n$-grams, if it is present, increase the count.
   5. Add the $n$-grams dictionary to the features table. Each rows represent one command with the columns being the features $n$-grams.

5.4.3 Creating the $k$-NN Model

Now that the features table is created, we run the module MLkNNTrain on the merged feature table to create the model. We run it for multiple values of $k$.

5.4.4 Generating the Test Data

As in section 5.4.2, we generate the features table on the test data by using the $n$-grams features generated on the training data and then, merge both tables.
5.4.5 Testing the Model

Using the model generated with the training data, we can now test it with the features table from the testing data. Results will vary depending on the $k$ value, how many top $t$ $n$-grams were selected, the size of $n$ for the $n$-grams, and the threshold to decide to which label round the predicted value in the module MLkNNTest (when $k > 1$).
6 Results

In this section, we describe the results we found by using both classifiers on the following data:

- **Training:**
  - Cowrie: August 2017 logs from our honeypot (31 days, 2'774 unique commands out of 837'176 commands in total).
  - BashHistory: randomly selected files with 11'036 unique commands.

- **Testing:**
  - Cowrie: September 2017 logs from our honeypot (30 days, 2'760 unique commands out of 1'408'905 commands in total).
  - BashHistory: randomly selected files with 11'525 unique commands.

The ratio between the number of Cowrie and Bash History was selected because in real scenarios, many more benign commands would be used than malicious ones. Due to the high variability of the Bash History commands, we want to make sure our classifier is capable of differentiating them by using a high number. The number of unique commands is specified for Cowrie to show the low variability in the commands seen on the honeypot. The performance of the pipeline is another reason for this choice of data sets sizes: for a given top \( t \), \( n \)-grams and \( N \) commands, it takes multiple hours for a twelve cores virtual server with 16 Gigabytes of RAM to process all data.

We will denote data originating from Cowrie as 1 (malicious) and from BashHistory as 0 (benign). So a true positive (TP) result is when the classifier outputted a 1 for a Cowrie command and for a true negative (TN), a 0 for a BashHistory command.

Respectively, a false positive (FP) is when we classified incorrectly a BashHistory command as a Cowrie command (Type I error) and for a false negative (FN), the opposite (Type II error). Type I errors are annoying for normal users who will be classified as malicious while a Type II error will let an attacker do its bidding on the SSH server.

Several parameters are interesting to denote the performance of binary classifiers:

- **True Positive Rate:** also known as recall or sensitivity. \( TPR = R = \frac{TP}{TP+FN} \), denotes the proportion of correctly identified positives.

- **True Negative Rate:** also known as specificity. \( TNR = \frac{TN}{TN+FP} \), measures the proportion of correctly identified negatives.

- **False Positive Rate**, \( FPR = 1 - TNR \) represents the proportion of incorrectly identified positives.

- **False Negative Rate**, \( FNR = 1 - TPR \) measures the proportion of incorrectly identified negatives.

- **Precision**: \( PPV = \frac{TP}{TP+FP} \) or the positive predictive value

- \( F_1 = 2 \cdot \frac{PPV \cdot R}{PPV+R} \). The \( F_1 \) score tells us about the accuracy of the test taking into account the precision (\( PPV \)) and the recall (\( R \)). \( F_1 \) is also known as the harmonic mean of precision and recall.

- \( F_{0.5} = 1.25 \cdot \frac{PPV \cdot R}{0.25 \cdot PPV + R} \) which increases the weight of recall over precision.

- \( F_2 = 5 \cdot \frac{PPV \cdot R}{4 \cdot PPV + R} \) which reduces the influence of false negatives.
• Accuracy: \( acc = \frac{TP + TN}{TP + FP + TN + FN} \).

The classifier returns a value between 0 and 1. When the value is not an integer, a threshold is used to decide to which label round the value. This threshold with the \( k \) value for the \( k \)-nearest neighbors, the number of top \( t \) \( n \)-grams to use, and up to how many \( n \) for the \( n \)-grams are the four variables we can adapt to optimize the results.

We can expect to find an optimum value for both top \( t \) and \( n \) for the \( n \)-grams. This will be highly dependent on the training and testing data. Remember that those parameters will decide how many features are used. Choosing as many as possible might not be the best idea as it will increase the computational time without necessarily improving the performance of the classifier.

We will generate the results for each of the following values for all variables when possible:

- **threshold** \( thr = [0, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8, 0.9, 1] \) where \( t = 0 \) we round up all values to 1 (malicious) and for \( t = 1 \) we round down all values to 0 (non-malicious).
- **top \( t \) \( n \)-grams**: \( t = [50, 100, 175, 250, 500, 750, 1000] \) where \( t = 50 \) means that we keep the top 50 grams for each \( n \) value of the \( n \)-grams.
- **\( n \)-grams** \( n = [1, 2, 3, 4, 5] \) where \( n = 2 \) means that we use the 1-grams and the 2-grams.
- **\( k \)-Nearest Neighbors**: \( k = [1, 2, 3, 4, 5, 6, 7, 8, 9] \) where \( k = 2 \) means that we take the two closest neighbors as the label. This also means that we can have commands with values at 0.5 when the closest two neighbors are different. The threshold will decide to which class (BashHistory: 0 or Cowrie: 1) it will be rounded. An odd number is recommended.

### 6.1 1-Command Classifier

Classifying correctly one command between two different sources is not an easy task since common commands are used indiscriminately both in Cowrie and BashHistory logs. However, classifying non-common commands is easier since they only occur in one of the two sources. The script containing all steps to get those results can be found in Appendix E.

Using the four variables described above, 3'276 evaluation runs are generated by the module MLkNNTest for the 1-Command Classifier. We are first going to investigate with a threshold value fixed at 0.5 which leaves us with 252 files. Out of those runs, we start by fixing \( k \) to compare values of accuracy between the different top \( t \) and \( n \)-grams value.

First, we are going to investigate the following parameters: how do the top \( t \) \( n \)-grams and \( n \)-grams influence the results for different values of \( k \). In Figure 7a below, we notice that most of the Cowrie commands were classified as malicious for most of the \( k \) values. However, on the other side, in Figure 7b, we notice that except for high values of top \( t \) and \( n \)-grams, the true negative rate is under 60%.

Those results are expected and are confirmed in Figure 8. The low variability of the Cowrie logs and the high variability of the Bash history commands could be an explanation for a much higher true positive rate than the true negative rate. We can see that increasing the number of features improves the quality of the classifier for the true negative rate. However, this has an optimum, afterwards, we are over-fitting our model with too many features.
(a) True Positive Ratio for multiple $k$ values for the 1-Command Classifier, $threshold = 0.5$

(b) True Negative Ratio for multiple $k$ values for the 1-Command Classifier, $threshold = 0.5$

Figure 7: TPR vs TNR scores
Figure 8: Accuracy score for multiple $k$ values for the 1-Command Classifier, threshold $= 0.5$

We show in the table 4 below a comparison between various best values averaged by the $k$ values.

Table 4: Scores comparison for higher values of top $t$ $n$-grams, sorted by accuracy, averaged by $k$ value

<table>
<thead>
<tr>
<th>Label</th>
<th>TNR</th>
<th>TPR</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = 1000$, $n = 1$</td>
<td>0.741515</td>
<td>0.987192</td>
<td>0.864354</td>
</tr>
<tr>
<td>$t = 750$, $n = 1$</td>
<td>0.739202</td>
<td>0.987046</td>
<td>0.863124</td>
</tr>
<tr>
<td>$t = 500$, $n = 1$</td>
<td>0.737063</td>
<td>0.986378</td>
<td>0.861720</td>
</tr>
<tr>
<td>$t = 500$, $n = 5$</td>
<td>0.740339</td>
<td>0.978919</td>
<td>0.859629</td>
</tr>
<tr>
<td>$t = 500$, $n = 2$</td>
<td>0.723870</td>
<td>0.985960</td>
<td>0.854915</td>
</tr>
<tr>
<td>$t = 500$, $n = 4$</td>
<td>0.722540</td>
<td>0.985981</td>
<td>0.854260</td>
</tr>
<tr>
<td>$t = 500$, $n = 3$</td>
<td>0.722579</td>
<td>0.985604</td>
<td>0.854092</td>
</tr>
<tr>
<td>$t = 750$, $n = 3$</td>
<td>0.739934</td>
<td>0.986315</td>
<td>0.817703</td>
</tr>
</tbody>
</table>

Without averaging on the $k$ values, the top results for the True Negative Rate (TNR) can be found in the table 5 below.
Table 5: Scores comparison the top True Negative Rate values for various \( k \), top \( t \) and \( n \)-grams

<table>
<thead>
<tr>
<th>Rank</th>
<th>Label</th>
<th>TNR</th>
<th>TPR</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( k = 7, t = 750, n = 3 )</td>
<td>0.813877</td>
<td>0.983076</td>
<td>0.867284</td>
</tr>
<tr>
<td>2</td>
<td>( k = 7, t = 1000, n = 1 )</td>
<td>0.813617</td>
<td>0.983829</td>
<td>0.898723</td>
</tr>
<tr>
<td>3</td>
<td>( k = 7, t = 750, n = 1 )</td>
<td>0.811535</td>
<td>0.983640</td>
<td>0.897588</td>
</tr>
<tr>
<td>4</td>
<td>( k = 9, t = 750, n = 1 )</td>
<td>0.809280</td>
<td>0.982324</td>
<td>0.895802</td>
</tr>
<tr>
<td>5</td>
<td>( k = 9, t = 750, n = 3 )</td>
<td>0.809107</td>
<td>0.981196</td>
<td>0.863426</td>
</tr>
<tr>
<td>10</td>
<td>( k = 5, t = 1000, n = 1 )</td>
<td>0.804857</td>
<td>0.984393</td>
<td>0.894625</td>
</tr>
<tr>
<td>11</td>
<td>( k = 7, t = 500, n = 1 )</td>
<td>0.802775</td>
<td>0.982136</td>
<td>0.892456</td>
</tr>
<tr>
<td>13</td>
<td>( k = 7, t = 500, n = 4 )</td>
<td>0.801648</td>
<td>0.982324</td>
<td>0.891986</td>
</tr>
<tr>
<td>14</td>
<td>( k = 7, t = 500, n = 5 )</td>
<td>0.801474</td>
<td>0.982324</td>
<td>0.891899</td>
</tr>
<tr>
<td>15</td>
<td>( k = 7, t = 500, n = 2 )</td>
<td>0.801388</td>
<td>0.981760</td>
<td>0.891574</td>
</tr>
</tbody>
</table>

We notice that optimum values for \( k \) are \( k = 7 \) and \( k = 9 \) for which the true negative ratio is around 80% for the top \( t = 500 \) \( n = 2, n = 3, n = 4, n = 5 \) \( n \)-grams. We notice that adding more top \( t \) \( n \)-grams do not impact greatly those results.

Since the cost in performance (processing and storage) is high for a minimal increase, we will keep evaluating for the top \( t = 500 \) and \( n = 5 \) at most. However, we hope that by adding more commands for the N-Commands classifier, we can reduce that number. Those values are dependent on the quantity of training and testing data we choose.

Deciding on a threshold is not an obvious task as it depends on the \( k \) value and is a trade-off between false positives and false negatives. We can see below results for various thresholds in a receiver operating characteristic curve or (ROC curve) which plots the true positive rate (TPR) against the false positive rate (FPR), for various threshold values. Remember that which thresholds make sense depends on the value of \( k \). For example, for \( k = 2 \), the thresholds that make sense are 0, 0.5 and 1, where 0 means that all values are rounded to 1 (Cowrie) and for a threshold of 1, rounded to 0 (BashHistory). In Figure 9, we plot the ROC with the thresholds marked on the lines for the top \( t = 500 \) \( n = 4,5 \)-grams and \( k = 7 \)-Nearest Neighbors.
We compare several threshold values in the table 6 by taking the highest accuracy score for each thresholds and see that optimum values are $thr = 0.8$ and $thr = 0.9$. Notice that the value of $k$-NN, top $t$ and $n$-grams change according to the threshold. We remark that using a higher threshold highly increases the value for the True Negative Rate and lowers slightly the True Positive Rate.
Table 6: Scores comparison for various thresholds, top 1 accuracy score for each threshold

<table>
<thead>
<tr>
<th>Label</th>
<th>TNR</th>
<th>TPR</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>thr = 0.25, k = 9, t = 750, n = 1</td>
<td>0.781873</td>
<td>0.984957</td>
<td>0.883415</td>
</tr>
<tr>
<td>thr = 0.4, k = 8, t = 1000, n = 1</td>
<td>0.809020</td>
<td>0.983829</td>
<td>0.896424</td>
</tr>
<tr>
<td>thr = 0.5, k = 7, t = 1000, n = 1</td>
<td>0.813617</td>
<td>0.983829</td>
<td>0.898723</td>
</tr>
<tr>
<td>thr = 0.75, k = 3, t = 500, n = 5</td>
<td>0.964007</td>
<td>0.910117</td>
<td>0.937062</td>
</tr>
<tr>
<td>thr = 0.8, k = 4, t = 500, n = 5</td>
<td>0.976236</td>
<td>0.903347</td>
<td>0.939792</td>
</tr>
<tr>
<td>thr = 0.9, k = 4, t = 500, n = 5</td>
<td>0.976236</td>
<td>0.903347</td>
<td>0.939792</td>
</tr>
</tbody>
</table>

One interesting value is the area under the curve (AUC) which denotes the ability of correctly classifying where 1 is a perfect result, and 0.5 is as good as a random guess. For the highest ROC AUC (top t = 500 and n = 5-grams), we plot its ROC plot at various thresholds for each k in the Figure 10. This plot confirms again the good performance of our system for higher values of k. We notice that the classifying as BashHistory (0) get a lower score on general. The results can be explained due to the high variability between the training and testing data for BashHistory while for Cowrie, it might not change too much from one month to the next one.

![ROC Plot](image)

Figure 10: Receiver Operating Curves for top t = 500 and n = 5 n-grams

### 6.2 2-Commands Classifiers

Moving from the 1-Command Classifier, we start by testing the 2-Command Classifier, we expect that adding commands will reduce the need for more features as a sequence of commands will decompose into more ngrams which will be part of the features, creating a more detailed features vector. Remember that generating pairs of two commands are done in a sliding way: we will produce pairs of command 1 and 2, 2 and 3, etc. this will greatly increase the number of commands. As we can imagine, adding commands increases both the accuracy in Figure 11 and the true negative rate in Figure 12.
Figure 11: Accuracy score for multiple $k$ values for the 2-Command Classifier, $\text{threshold} = 0.5$

The accuracy score is quite good for the top $t = 250$ $n = 5$-grams, however, it provides still a quite low result for the true negative ratio (0.867 for the average of the $k$’s.).
Results of the averages of the scores for the top $t = 500$ ngrams with $n = 1, 2, 3, 4, 5$ for the 1-Command and the 2-Commands classifiers can be found in the table 7 below. We notice improvement on all fronts, especially the true negative rate which is one of our most important parameter to optimize.

Table 7: Scores comparison between the 1-Command and the 2-Command Classifier

<table>
<thead>
<tr>
<th>Score</th>
<th>1-Command</th>
<th>2-Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.856923</td>
<td>0.945346</td>
</tr>
<tr>
<td>TPR</td>
<td>0.984568</td>
<td>0.974466</td>
</tr>
<tr>
<td>TNR</td>
<td>0.729278</td>
<td>0.913655</td>
</tr>
</tbody>
</table>

Another interesting remark is that the best values for $k$ are $k = 7, 9, 5$ for the 1-Command and $k = 9, 3, 1$ for the 2-Command classifier. This and the results in the plots above, tells us that the values of $k$-Nearest Neighbors do not influence that much, the only recommendation is to use a odd number as this helps the classifier to label the session correctly.

Regarding the thresholds, we investigated multiple values in the table 8 and in opposite to the 1-Command classifier, there are not significant changes between thresholds. We conclude that keeping a threshold value $thr = 0.5$ is good enough.
Table 8: Scores comparison for various thresholds, top 1 accuracy score for each threshold

<table>
<thead>
<tr>
<th>Label</th>
<th>TNR</th>
<th>TPR</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$thr = 0.25, k = 1, t = 500, n = 3$</td>
<td>0.906561</td>
<td>0.978379</td>
<td>0.942470</td>
</tr>
<tr>
<td>$thr = 0.4, k = 3, t = 500, n = 3$</td>
<td>0.907255</td>
<td>0.976840</td>
<td>0.942047</td>
</tr>
<tr>
<td>$thr = 0.5, k = 9, t = 500, n = 5$</td>
<td>0.991679</td>
<td>0.953533</td>
<td>0.971010</td>
</tr>
<tr>
<td>$thr = 0.75, k = 9, t = 500, n = 3$</td>
<td>0.995146</td>
<td>0.946057</td>
<td>0.970601</td>
</tr>
<tr>
<td>$thr = 0.8, k = 6, t = 500, n = 3$</td>
<td>0.994799</td>
<td>0.951774</td>
<td>0.973287</td>
</tr>
<tr>
<td>$thr = 0.9, k = 9, t = 500, n = 5$</td>
<td>0.999133</td>
<td>0.928540</td>
<td>0.960883</td>
</tr>
</tbody>
</table>

6.3 N-Commands Classifiers

In this section, we will compare results for various values of $N$. As we want to optimize the true negative ratio, this is the first result we will analyze in Figure 13a with the top $t = 250$ and $n = 3$-grams for $N = 1, \ldots, 10$. 
We immediately notice the increase going from 2 commands to 4 commands
and starts to stagnate around 6 commands. Looking at the averages in the table 9 below, we conclude that an optimal number of commands needed to differentiate between both types of sessions is 6 but 5 commands should be enough since the difference is not too big.

Table 9: Scores comparison for 3 – 9-Commands Classifier for top $t = 250$ and $n = 3$-grams

<table>
<thead>
<tr>
<th>Score</th>
<th>3-CC</th>
<th>4-CC</th>
<th>5-CC</th>
<th>6-CC</th>
<th>7-CC</th>
<th>8-CC</th>
<th>9-CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.955025</td>
<td>0.985616</td>
<td>0.983056</td>
<td>0.991193</td>
<td>0.989983</td>
<td>0.990173</td>
<td>0.987459</td>
</tr>
<tr>
<td>TPR</td>
<td>0.982272</td>
<td>0.996101</td>
<td>0.995318</td>
<td>0.995728</td>
<td>0.996350</td>
<td>0.996597</td>
<td>0.996586</td>
</tr>
<tr>
<td>TNR</td>
<td>0.927778</td>
<td>0.975131</td>
<td>0.983056</td>
<td>0.986658</td>
<td>0.978574</td>
<td>0.979294</td>
<td>0.972904</td>
</tr>
</tbody>
</table>

However, we might want to classify commands earlier if it is possible. To answer, this question, we generate our model again but with a top $t = 500$ and $n = 3$-grams and we observe the results in the table 10.

Table 10: Scores comparison for 1 – 4-Commands Classifier for top $t = 500$ and $n = 3$-grams

<table>
<thead>
<tr>
<th>Score</th>
<th>1-CC</th>
<th>2-CC</th>
<th>3-CC</th>
<th>4-CC</th>
<th>5-CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.854092</td>
<td>0.944264</td>
<td>0.973192</td>
<td>0.995506</td>
<td>0.995364</td>
</tr>
<tr>
<td>TPR</td>
<td>0.985604</td>
<td>0.974348</td>
<td>0.980950</td>
<td>0.994143</td>
<td>0.993769</td>
</tr>
<tr>
<td>TNR</td>
<td>0.722579</td>
<td>0.914179</td>
<td>0.965435</td>
<td>0.996870</td>
<td>0.998141</td>
</tr>
</tbody>
</table>

The results from the table can be seen in the Figure 14a below. For the 3-Command classifier, we can accurately classify a command as non-malicious with a probability of 96.54% with a true positive rate 97.86%. Remember that this result is obtained by averaging the results of the different $k$-NN values with a threshold of 0.5. Reaching 4 commands, we manage to obtain a 99.41% of accurately predict a Cowrie session and 99.69% (rounded) to correctly label a Bash History session. With less than 1% false positives and negatives, we show that the classifier is capable of labeling sessions correctly and quickly.
(a) True Positive Ratio for multiple $k$ values for the $1-5$-Command Classifier for top $t = 500$ and $n = 3$-grams, threshold $= 0.5$.

(b) True Negative Ratio for multiple $k$ values for the $1-5$-Command Classifier for top $t = 500$ and $n = 3$-grams, threshold $= 0.5$.

Figure 14: TPR vs TNR scores
7 Conclusions and Future Work

After analyzing the results found in the previous section and reviewing the performance of the pipeline, we discuss some future improvements to our tool and approach in the next sections.

7.1 Conclusions

The results in section 6 show that our system is capable of differentiating quite quickly between both sources of data. After 4 commands, we are capable to classify correctly non-malicious commands with a probability of 99.69\% (TNR) and malicious command with a probability of 99.41\% (TPR) using the top $t = 500$ $n = 3$-grams model. For 5 commands with the model using the top $t = 250$ and $n = 3$-grams, we reach a success rate of 99.53\% and 98.3\% for the true positive rate and true negative rate respectively. The results are great from the true negative perspective: the training and testing data set for Bash History was randomly selected and with high variability since coming from different bash history files. This means that regardless of who the user is, we were capable of labeling them as non-malicious with good accuracy. However, the processing time as well as the storage needed increases rapidly as we increase the top $t$ $n$-grams, the number of $N$-Commands to test, the number of $k$ value for the $k$-NN to use and the size of the training and testing data which forced us to make compromises for the testing.

Since the Bash History files contain commands with a high variety, we can conclude that this source of data is good enough to represent most users profiles in a work environment. For the commands captured by our Cowrie honeypot, due to the fact that most of the attacks are unsophisticated, we do not suppose that it will perform as well against sophisticated attacks and highly-skilled attackers. Improving the data sources is discussed in the final section.

Classifying one command as malicious or benign got acceptable scores due to the limits of our training and testing sets, however, we do not expect to classify sessions after a single command in a real-world scenario. Moving to the $N$-classifier, we would likely be able to detect threats in a real-world scenario. However, due to the consequences of misclassifying false positives, we need to improve our model by increasing its training data and parameters such that we are able to classify a session correctly with a even higher probability. This improvement only depends on the quality of the data and the performance of our machines available.

7.2 Future Work

The results we found with our tool is highly depended on the quality of the data: our next step is to diversify our sources using Bash history files from the Data Science Lab at armasuisse. Since those files contain timestamps, we could add other features related to timings and keystrokes. To be able to detect more advanced attacks, we are in discussion with the organizer of the world’s largest cyber-defence exercise: Locked Shields [27] to gather data from the red team attacks during their next exercise. This would provide us with malicious data coming from professional pentesters which would allow us to more accurately detect advanced persistent threats, obfuscated attacks and exfiltration of data.

Our next improvement on our $k$NN model are the following:

1. Increasing the size of our model by being able to incrementally add training data.
2. Reducing the steps necessary to generate and test the model by improving the chaining between the inputs and outputs of the modules of the Log Analyzer.
3. Improve overall performance.

After choosing an appropriate top $t$ $n$-grams in respect to performance and score for bigger sets of data, we will be able to generate multiple models for each $N$-Commands classifiers which can be continuously improved with more data. Next, we can focus on developing the real-time monitoring system which receives the commands executed on the SSH servers and classify them using the generated models.

More work is needed on the structure of the modules to better accommodate the various log format of the Log Analyzer and afterwards, improving the analysis of logs originated from honeypots with daily statistics to be able to generate reports which details trends over time. We are in discussion with researchers from the NATO Cooperative Cyber Defence Centre of Excellence (CCDCOE) to correlate our information with other honeypots that are geographically distributed which would provide data about botnets and newer malwares. With the help of Exeon Analytics, our Cowrie data sources will be improved by using data from their personalized high-interaction SSH honeypots.

Another improvement on the classification side would be to implement more analyzing modules such as other classifier algorithms and compare their results with our $k$-Nearest Neighbors approach.
References


