2019 11th International Conference on Cyber Conflict
Silent Battle
T. Minárik, S.Alatalu, S.Biondi,
M.Signoretti, I.Tolga, G.Visky (Eds.)
2019 © NATO CCD COE Publications, Tallinn

Detection of Malicious Remote Shell Sessions

Pierre Dumont
Department of Information Technology and Electrical Engineering
ETH Zürich / Kudelski Security
Zürich / Lausanne, Switzerland
pierre.dumont@kudelskisecurity.com

Roland Meier
Department of Information Technology and Electrical Engineering
ETH Zürich
Zürich, Switzerland
meierrol@ethz.ch

David Gugelmann
Exeon Analytics
Zürich, Switzerland
david.gugelmann@exeon.ch

Vincent Lenders
Science and Technology
armasuisse
Thun, Switzerland
vincent.lenders@armasuisse.ch

Abstract: Remote shell sessions via protocols such as SSH are essential for managing systems, deploying applications, and running experiments. However, combined with weak passwords or flaws in the authentication process, remote shell access becomes a major security risk, as it allows an attacker to run arbitrary commands in the name of an impersonated user or even a system administrator. For example, remote shells of weakly protected systems are often exploited in order to build large botnets, to send spam emails, or to launch distributed denial of service attacks. Also, malicious insiders in organizations often use shell sessions to access and transfer restricted data.

In this work, we tackle the problem of detecting malicious shell sessions based on session logs, i.e., recorded sequences of commands that were executed over time. Our approach is to classify sessions as benign or malicious by analyzing the sequence of commands that the shell users executed. We model such sequences of commands as n-grams and use them as features to train a supervised machine learning classifier.

Our evaluation, based on freely available data and data from our own honeypot infrastructure, shows that the classifier reaches a true positive rate of 99.4% and a true negative rate of 99.7% after observing only four shell commands.

Keywords: *malware, botnets, machine learning, attribution, digital forensics, digital trust, authentication*

1. INTRODUCTION

The rise of cloud and Internet of Things (IoT) infrastructure makes it crucial to access computing services and devices remotely for configuration, maintenance or deployment purposes. Recent numbers show that the secure shell protocol (SSH), which is the state-of-the-art method for remote shell login, is available on over 21 million publicly accessible devices [1]. This number does not include devices that provide SSH access only from the internal network or via VPN, which is the common practice for enterprise and home networks.

Ensuring the security of remote shell sessions is not a trivial task. While the SSH protocol itself is believed to be secure in terms of implemented cryptographic primitives, malicious actors still manage to gain access to Internet-facing SSH servers.

Outdated software, weak or stolen credentials and lack of multi-factor authentication are frequent ways for malicious actors to gain access to a remote device. Recent examples include the Mirai botnet, where attackers gained access to 600,000 devices [2].

Systems that are only available internally (e.g., to employees of a company) are generally better protected because (i) they cannot be attacked from outside the company network; (ii) the administrators can enforce strong authentication schemes; and (iii) the tasks that each user is allowed to do on these systems are usually well-defined. However, even if access is restricted and strong authentication schemes make it impossible to steal credentials, internal systems are not spared from attacks because studies show that many attacks come from insiders [3].

The operator of an infrastructure therefore needs a way to differentiate between legitimate and malicious actions without trusting the identity or authenticity of the logged-in user, because failing to block malicious actors can have severe consequences including downtime, data loss, and reputation loss.

In this paper, we present a system that analyzes commands within shell sessions and classifies them as benign or malicious. We leverage the fact that remote shell sessions leave traces by logging the executed commands.

Problem statement and research questions. We address the following problem statement:

Solely based on the commands that are executed in a shell session, we aim at building a classifier capable of quickly distinguishing between benign and malicious sessions. More specifically, we answer the following research questions:

1. Do individual commands contain enough information for a classification between malicious and benign purposes?
2. How many commands need to be analyzed to accurately identify malicious remote shell sessions?
3. Can we detect attackers who try to obscure their commands?

Approach. Our approach is to perform the classification using supervised machine learning, trained on logs from real (benign) users and malicious logs from real attackers. As features, we use sequences of commands of variable length (i.e., n-grams built from entries in session logs). This allows us to capture the context in which a command was executed.

Challenges. The main challenges that we face in this research are the following:

- Attackers tend to be unpredictable and commands can be ambivalent in their purpose depending on the context. This requires us to define and extract features that capture the context in which a command was executed.
- Attackers can mix malicious commands with harmless commands in order to obscure their intentions. This requires our features to be meaningful even if the context in which commands are executed is obscured.
- Attackers can cause considerable damage with only a few commands. This requires our classifier to output reliable results after a short time (i.e., after analyzing a few commands).

Contributions. The main contributions of this paper are:

- A fully implemented binary classifier using machine learning algorithms to differentiate between malicious and benign shell commands (Section 4.).
- A thorough evaluation of the results to show the usability of our classifier (Section 5.).
- Case studies to illustrate use-cases for our classifier (Section 6.).

2. RELATED WORK

To the best of our knowledge, this work is the first that addresses the problem of classifying malicious remote shell sessions based on the executed commands. Existing works show how attackers can gain access to SSH-enabled devices and how to identify malicious commands in source code. Below, we reference the most relevant publications in these areas.

In [4], Song et al. study users' typing patterns when logging in on SSH sessions to guess their passwords. Other papers (e.g., [5], [6]) analyze SSH from a network perspective and focused on the threat of SSH brute-forcing. Using honeypots, the authors of [7] and [8] study how attackers operate after they gain access to an SSH-enabled device. In contrast to their work, our approach is agnostic to how attackers manage to gain remote access to a system and thus also works for malicious insiders that have legitimate access to a system.

In [9], Shabtai et al. present a broad survey of machine learning classifiers for detecting malicious code. Many of the techniques employed to classify malicious executable files can be found in other papers (e.g. n-grams [10] [11]). In contrast to identifying malicious instructions in a program, our approach works in real time and does not require the full session (i.e. the entire program). Therefore, our approach is useful for immediately intercepting ongoing remote shell sessions, which is different from analyzing static program code.

3. BACKGROUND AND DATA SOURCES

In this section, we define our attacker model and introduce the data sources that we used to train and evaluate the classifier.

A. Attacker Model

In this paper, we focus on commands executed in the UNIX-like shell of a system that can log executed commands. A server with a shell provides a command-line interface to users either on a physical terminal or on a remote interface. To access the shell remotely, the most commonly used protocol is Secure Shell (SSH). SSH servers run software such as OpenSSH to authenticate users and provide interactive terminals. SSH provides a secure encrypted channel with user authentication over passwords or certificates.

We consider the threat of an attacker who has bypassed the SSH authentication system and is therefore able to login remotely. This could be achieved by exploiting SSH software vulnerabilities, performing a man-in-the-middle attack, or by acquiring the necessary password/certificate by guessing or data theft. Alternatively, the attacker could be a malicious user with legitimate access to the system. The goal of our work is to detect such attackers as quickly as possible by analyzing their behavior.

B. Data Sources

To train and evaluate our classifier, we need data from both benign and malicious users of shell sessions. In this section, we describe how we collected publicly available

logs of benign sessions and how we used our own infrastructure to collect logs of malicious sessions.

1) Public Bash History Logs

By default, the Bash shell (the most widely used shell) and most other shells log the commands that a user executes. In the example of Bash, all executed commands are by default written to a file in the user's home directory. To obtain access to a large amount of such history files, we leverage the fact that many developers, intentionally or not, publish their history files in public GitHub repositories [12]. Using GitHub's API [13], we were able to collect 3,146 non-empty Bash history files containing a total of 973,621 commands and 234,063 unique commands. Since these commands originate from real users and it is unlikely that attackers would publish their files, we labeled the commands contained in these files as benign.

2) Honeypot Logs

Honeypots are systems that appear to be vulnerable to some attacks. They are typically deployed by security researchers to trick attackers into revealing their malicious code, allowing the defenders to learn about their latest methods of operation.

For our purposes, we run Cowrie, a medium-interaction SSH/Telnet honeypot [14]. After running our honeypot for 5 months (June – October 2017), we collected data from over 320,000 remote shell sessions and a total of 2.35 million executed commands (5,316 unique commands). Because no benign user would log in to an unknown device, it is safe to assume that all these sessions were established with malicious intent by automated scripts and bots. Thus, we labeled the commands in these sessions as malicious.

4. CLASSIFYING REMOTE SHELL SESSIONS

In this section, we describe how we reached our goal of differentiating between malicious and benign sequences of commands. After explaining how we identified useful features, we describe two variations of our classifier: we start with a one-command classifier that classifies one command at a time, and generalize it to an N-command classifier that classifies N subsequent commands at a time.

A. Feature Selection

Whether a command is malicious or not depends on three factors:

- The program that is executed: e.g. "rm" to delete files or directories

- The parameters: e.g. “-rf ./*” to recursively delete everything within the current directory
- The context: e.g. the current directory or previously executed commands.

For example, executing “rm -rf ./*” in a user’s “downloads” directory is likely not malicious but rather used to clean up old files. On the other hand, executing “rm -rf ./*” in a server’s root directory would make the server unusable and is probably done with malicious intent (or by accident). In the following, we describe how we built features for machine learning based on this information. With these features, we wanted to cover the first two factors (program and parameters). For the third factor (context), we used sequences of commands to feed the classifier (cf. Section 4. B.3).

To compute features that contain programs and parameters, there are two extreme approaches: taking the entire command, program and parameters: e.g. “rm -rf ./*”, as one feature or treating the program and parameters as independent features, e.g. “rm”, “-rf”, “./*”. However, both extremes are unpromising because they either fail to recognize similar commands – “rm -rf ./*” has similar effects as “rm -rf ./” – or fail to capture dependencies between the program and its parameters: “rm -rf ./*” is very different from “ls -rf ./*”. Therefore, we followed a middle course and used so-called n-grams as features. N-grams are sub-sequences of n items from a sequence of items. N-grams are widely used in natural language processing and existing work shows that they are also useful for classifying malicious executables [15].

We generate the n-grams by splitting a command at each whitespace, e.g. “rm -rf ./*” decomposes into the sequence [“rm”, “-rf”, “./*”]. This sequence then has 1-grams [“rm”, “-rf”, “./*”],2-grams [“rm -rf”, “-rf ./*”] and 3-gram “rm -rf ./*”.

In Section 4. B.1) we explain how we optimized the value of n for our classifier.

B. Classifier

In this section, we explain how our classifier works. We start by explaining the motivation behind our choice of k-NN as the classification algorithm to use and continue by describing two versions of our classifier: the first classifies one command at a time while the second works on multiple commands simultaneously.

1) Model Selection

For our classifier, we need an algorithm that fulfills the following requirements:

- It is suitable for clustering high-dimensional data
- It is fast enough to classify commands in real time

- It produces explainable results (e.g. to allow manual inspection by a security analyst)
- It provides high accuracy.

There are a large number of binary classification algorithms (e.g. Support Vector Machines (SVM), k-Nearest Neighbors (k-NN), or random forests) that are used in the security domain [16] [17]. After initial experiments with various algorithms, we decided to use the k-Nearest Neighbor (k-NN) algorithm, which achieves good, explainable, and quick results (cf. Section 5.) with a simple model. One major downside of k-NN is its (comparably) high memory requirement because the trained model must contain a large number of data points.

As explained in Section 4. A, we use n-grams consisting of program names and parameters as features for the classification. More precisely, we define a parameter, M and simultaneously use n-grams up to length M. That is, for M = 3, we use 1-grams, 2-grams, and 3-grams as features for the machine learning algorithm.

For each n-gram length, we only consider a fixed number of the most frequent n-grams. This is done to reduce the complexity of the model and avoid infrequent commands that might cause overfitting.

Each n-gram is represented as one dimension in k-NN classification, thus the higher this number is, the larger the space for classification.

2) 1-Command Classifier

The 1-command classifier works on a single command. This comes with the advantages that it is fast and easy to run but – because whether a command is malicious or benign often depends on the commands that were executed before and after – it achieves limited accuracy.

As stated above, we use the k-Nearest Neighbor (k-NN, with the default parameters specified by sklearn[1]) method to classify a command and thus a command is assigned to a label (malicious or benign) of the k-nearest commands.

3) N-Command Classifier

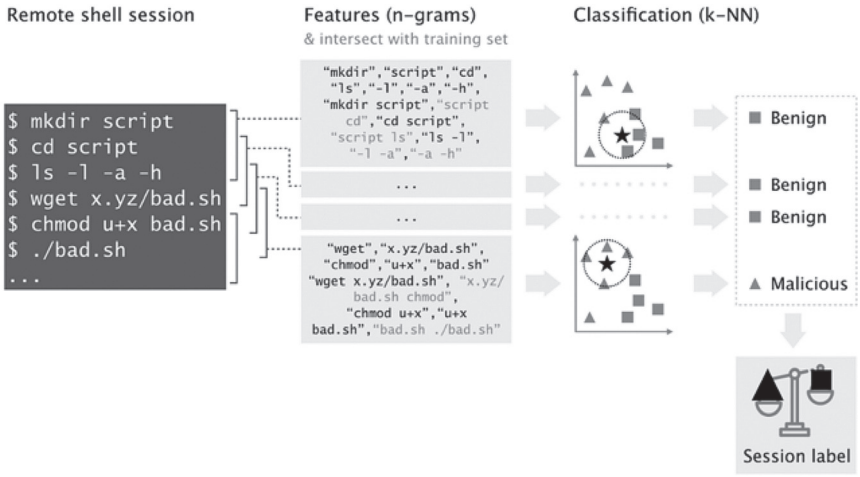
While the 1-command classifier is useful for quickly classifying individual commands, it lacks the ability to capture the context in which a command was executed (i.e. the preceding and succeeding commands); hence, we generalize it to an N-command classifier.

1 https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

As the name suggests, the N-command classifier analyzes N subsequent commands together. In contrast to the 1-command classifier (which builds the n-grams for each command individually), it uses a sliding window over the entire shell session and computes the top n-grams for each sequence of N subsequent commands. Similar to the 1-command classifier, it subsequently uses k-NN to predict whether a sequence of commands is malicious or not.

Figure 1. illustrates the N-command classifier with an example.

FIGURE 1. N-COMMAND CLASSIFIER SYSTEM OVERVIEW. FROM SESSION LOGS, WE PARSE N SUBSEQUENT COMMANDS (EXAMPLE: N = 3), COMPUTE THE N-GRAMS FOR N = 1...M (M = 2), REMOVE N-GRAMS THAT DO NOT EXIST IN THE TRAINING DATA, AND APPLY K-NN (K = 4) TO OBTAIN THE PREDICTED LABEL.



4) Classifying Sessions

To label a session, an operator could specify an arbitrary policy that uses a mix of predictions of individual commands and sequences of variable lengths to determine an action. This would allow an operator to implement custom policies such as to report, terminate or manually inspect sessions depending on the types of commands that they contain.

Since k-NN produces explainable models, the operator is even able to manually inspect the closest commands or sequences that were used to classify ongoing sessions. This would allow for a short feedback loop as the operator is capable of quickly dismissing false positives.

5. EVALUATION

In this section, our classifier is evaluated based on real-world data. After explaining our evaluation methodology, we present the resulting performance of our classifier. We show that the 1-command classifier can quickly identify malicious commands and that the N-command classifier improves the results even for small values of N. Finally, we evaluate the impact of an attacker who tries to obscure their intent.

A. Methodology

In this section, we provide details about the datasets, parameters, and performance metrics used.

1) Datasets

The data used in this paper originates from two data sources: publicly available shell session logs and our honeypot logs, both described in Section 3. B. In Table I, we summarize key information about the datasets used for the following experiments.

TABLE I: INFORMATION ABOUT THE DATASETS USED FOR THE EVALUATION

	Training	**Testing**
Malicious dataset	Honeypot logs (Aug 2017)	Honeypot logs (Sept 2017)
Total commands	837,176	1,408,905
Unique commands	2,774	2,760
Benign dataset	Random sample from public Bash history files	Random sample from public Bash history files
Total commands	29,710	39,749
Unique commands	11,051	11,539

2) Parameters and Metrics

In Table II, we list all the parameters that influence the performance of our algorithm, explain their meaning and specify the evaluated values.

TABLE II: PARAMETERS USED FOR THE EVALUATION

Parameter	**Explanation**	**Evaluated values**
M	Maximum n-gram length (the features are n-grams for n=1, …, M)	1, …, 5
N	Number of subsequent commands analyzed together	1, …, 10
T	Number of n-grams to keep for each n-gram length n	[50, 100, 250, 500, 750]
k	Number of neighbors (for k-NN)	[1, 3, 5, 8]

We interpret commands that are predicted to be malicious as "positive" and commands that are labelled benign as "negative". We then use standard statistical terms and metrics for binary classification:

- True Positive (TP): a malicious command was classified as malicious
- True Negative (TN): a benign command was classified as benign
- False Positive (FP): a benign command was classified as malicious
- False Negative (FN): a malicious command was classified as benign.

In the experiments, we typically report the True Positive Rate (TPR = TP / (TP + FN)), also known as recall, which denotes the proportion of correctly identified malicious commands, and the True Negative Rate (TNR = TN / (TN + FP)), also known as specificity, which denotes the proportion of correctly identified benign commands. The False Positive Rate (FPR = 1-TPR) and the False Negative Rate (FNR = 1-TNR) can be derived from these values.

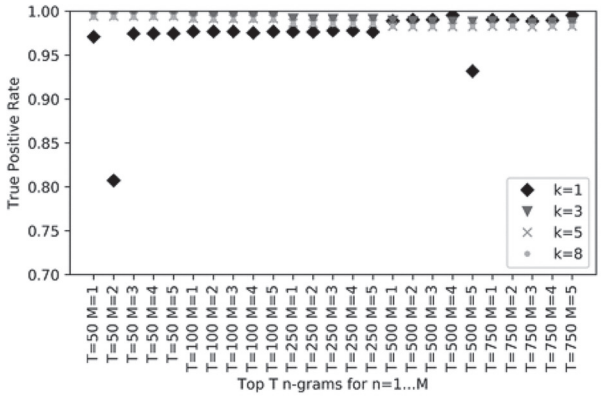
The accuracy of the classifier computes to acc= $\frac{TP+TN}{TP+TN+FP+FN}$.

B. Classification with the 1-Command Classifier

In this experiment, we evaluate the performance of the 1-command classifier, a special case of the N-command classifier with N = 1. This classifier is useful for fast classification, but has limited performance because it does not consider the context in which a command was executed.

We use the datasets described in Table I to train and test the classifier with different values for parameters T, M, and k (cf. Table II). In Figures 2, 3, and 4, we plot the resulting TPR, TNR, and accuracy, respectively.

FIGURE 2. TRUE POSITIVE RATE FOR DIFFERENT VALUES OF T, M, AND K FOR THE 1-COMMAND CLASSIFIER.

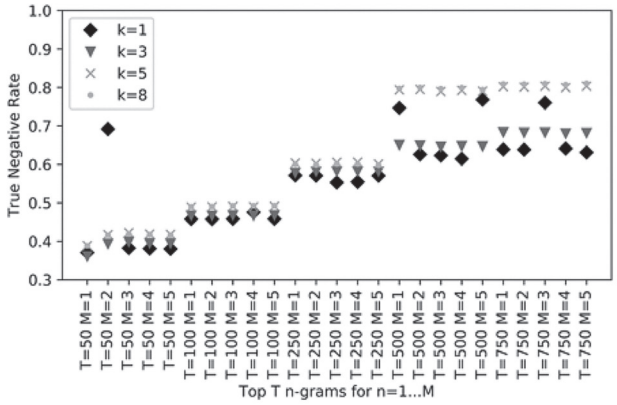


As Figure 2 shows, T and M have little impact on the identification of malicious commands and the 1-command classifier achieves a TPR of more than 0.98 in most configurations. An explanation for the small impact of T and M is the fact that most of the attacks against our honeypot are automated bots using scripts with low variability (i.e. they often run the same commands). Adding too many features by increasing T and M likely overfits the data, which explains the downward trend as both values increase.

Except for k = 1, which is expected to be unstable as it relies only on the closest neighbor, the number of chosen neighbors has little impact on the TPR. The observation that the results are good and stable for small values of k (e.g. k = 3) shows again the low variability; however, as we show below, a higher value for k is required for a good TNR.

In Figure 3, we analyze the True Negative Rate and show that the 1-command classifier obtains a TNR of up to 0.81 for the parameters used. The results for TNR show the opposite trend compared to the TPR: the higher T and M are, the better the performance of the 1-command classifier. This is due to the high variability of benign commands as they were taken randomly from many different users. For the best TNR, k should be chosen between 5 and 8.

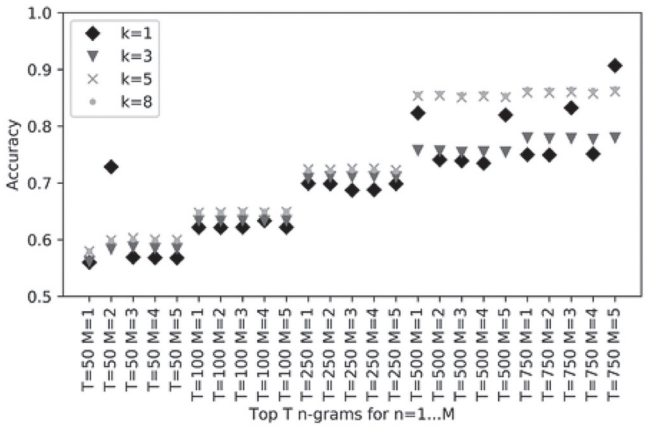
FIGURE 3. TRUE NEGATIVE RATE FOR DIFFERENT VALUES OF T, M, AND K FOR THE 1-COMMAND CLASSIFIER.



Combining the results for TPR and TNR shows that the best results (in terms of accuracy as seen in Figure 4) are achieved for T = 750, M = 5, and k = 8 (TPR = 0.983, TNR = 0.800, accuracy = 0.860). However, because these parameter values result in high resource requirements, we use the slightly sub-optimal parameter values T = 500 and M = 3 for evaluating the N-command classifier in the next section. Note

that the 1-command classifier using these parameters still achieves TPR = 0.982, TNR = 0.796, accuracy = 0.855 and is therefore only about 0.5% worse than the optimal configuration (w.r.t. accuracy for k = 8) while requiring approximately 50% less computation time (15 instead of 30 minutes to generate the testing features table), and 50% less storage (750 MB instead of 1.5 GB for the k-NN model for a specific k).

FIGURE 4. ACCURACY FOR DIFFERENT VALUES OF T, M, AND K FOR THE 1-COMMAND CLASSIFIER.



C. Classification with the N-Command Classifier

In this experiment, we evaluated the performance of the N-command classifier for different values of N and k. In contrast to the 1-command classifier evaluated above, the N-command classifier considers the context in which a command was executed by analyzing sequences of N commands (cf. Section 4. B.3).

Figures 5, 6, and 7 show plots of the TPR, TNR, and accuracy, respectively for the N-command classifier depending on N and k. As motivated in the previous section, we fix the values for T and M (T = 500, M = 3). The results show that the TPR as well as the TNR and the accuracy are significantly better for N > 3 compared to the 1-command classifier (note that the results for N = 1 correspond to the 1-command classifier). This confirms our intuition that considering the context in which a command was executed is essential for an accurate classification. We further notice that the value of k has little impact on the TPR and TNR for N > 3.

FIGURE 5. TRUE POSITIVE RATE OF THE N-COMMAND CLASSIFIER AS A FUNCTION OF N (FOR T = 500 AND M = 3). N > 3 SIGNIFICANTLY IMPROVES THE TPR COMPARED TO THE 1-COMMAND CLASSIFIER.

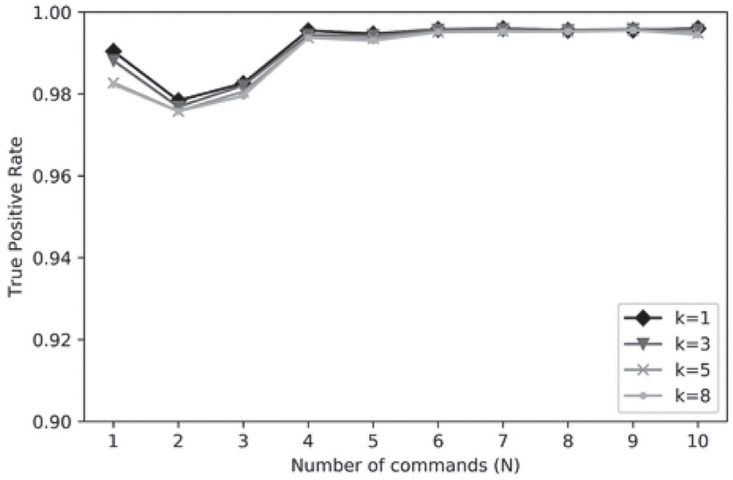


FIGURE 6. TRUE NEGATIVE RATE OF THE N-COMMAND CLASSIFIER AS A FUNCTION OF N (FOR T = 500 AND M = 3). N > 3 SIGNIFICANTLY IMPROVES THE TNR COMPARED TO THE 1-COMMAND CLASSIFIER.

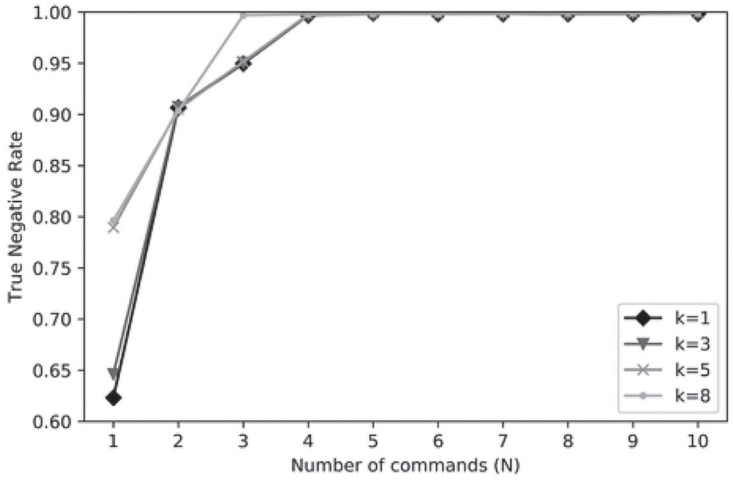
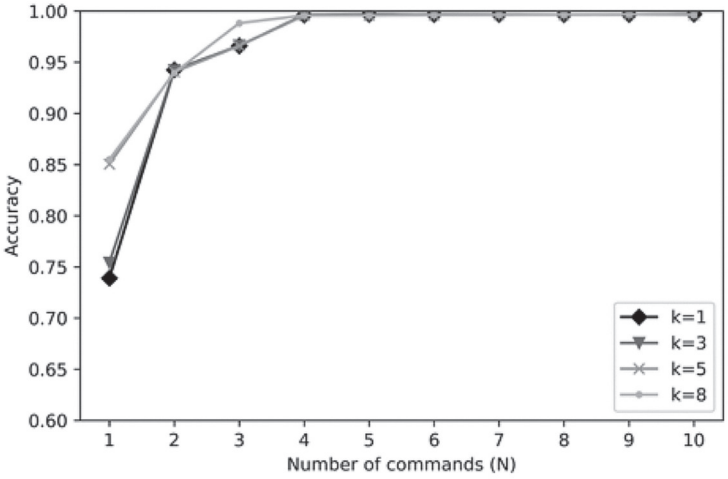


FIGURE 7. ACCURACY OF THE N-COMMAND CLASSIFIER AS A FUNCTION OF N (FOR T = 500 AND M = 3). N > 3 SIGNIFICANTLY IMPROVES THE ACCURACY COMPARED TO THE 1-COMMAND CLASSIFIER.



Since the N-command classifier needs to wait until a user has executed at least N commands, it is desirable to have N as small as possible. Therefore, N = 4 is an optimal choice. At this point (and for k = 3), the classifier achieves TPR = 0.994, TNR = 0.997 and accuracy = 0.996. Increasing N only slightly improves the results (by approximately 0.2% until N = 10) and requires waiting for more commands. Note that increasing the number of commands has no noticeable effect on the time it takes to classify those sequences.

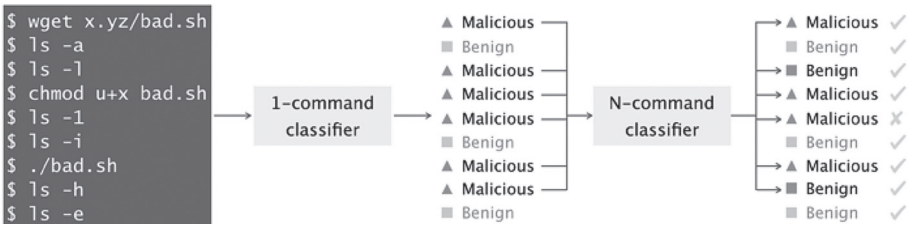
D. Robustness Against Obscuring Attempts

While the previously evaluated N-command classifier works well because it considers the context in which a command was executed, it is vulnerable to attackers that try to obscure the context. In particular, an attacker can interleave their malicious commands with benign commands. For example, assuming that the following commands are malicious [“wget x.yz/bad.sh”, “chmod u+x bad.sh”, “./bad.sh”], an attacker could add (presumably) non-malicious commands in between and run [“wget x.yz/bad.sh”, “ls”, “ls -a”, “ls -l”, “ls -al”, “chmod u+x bad.sh”, “ls”, “ls -a”, “ls -l”, “ls -al”, “./bad.sh”] instead. In this example, the benign “ls” commands hide the contextual information from the N-command classifier for N <= 5 because each sequence of 5 commands contains at most one malicious command.

As a promising solution, we propose to use the 1-command classifier in combination with the N-command classifier. That is, (i) we apply the 1-command classifier to identify and remove benign commands inserted by an attacker; and (ii) classify the

sequences of the remaining commands using the N-command classifier. We illustrate this approach in Figure 8 and describe it in detail in the following paragraphs.

FIGURE 8. THE 1-COMMAND CLASSIFIER IS USED AS A FILTER AND ONLY COMMANDS CLASSIFIED AS MALICIOUS BY THE 1-COMMAND CLASSIFIER ARE PROVIDED TO THE 4-COMMAND CLASSIFIER. THUS, THE 4-COMMAND CLASSIFIER IS PRESENTED WITH A SEQUENCE OF COMMANDS THAT IS AN APPROXIMATION OF THE ORIGINAL MALICIOUS SEQUENCE.



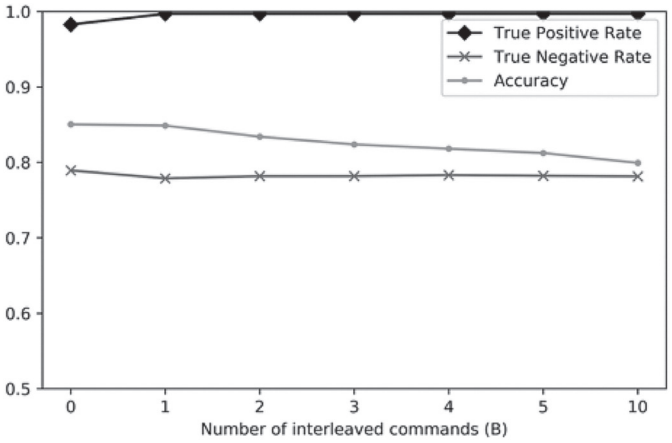
In the first step, we run the 1-command classifier on all commands to identify potentially malicious commands. From previous experiments, we know that the 1-command classifier has a TNR of approximately 80%. Therefore, after removing all commands that the 1-command classifier labeled as benign, we end up with approximately 20% of the benign commands (false positives) and 98% of the malicious commands (true positives).

In the second step, we apply the 4-command classifier on the remaining commands. The intuition behind this is that the 4-command classifier has a small FPR and can therefore compensate for the high FPR of the previously applied 1-command classifier.

Because the 4-command classifier analyzes sequences of 4 commands in a sliding window, each command is contained in up to 4 such sequences and gets a set of up to 4 predictions. To determine the final prediction of an individual command, we simply select the one that appeared most often within this set.

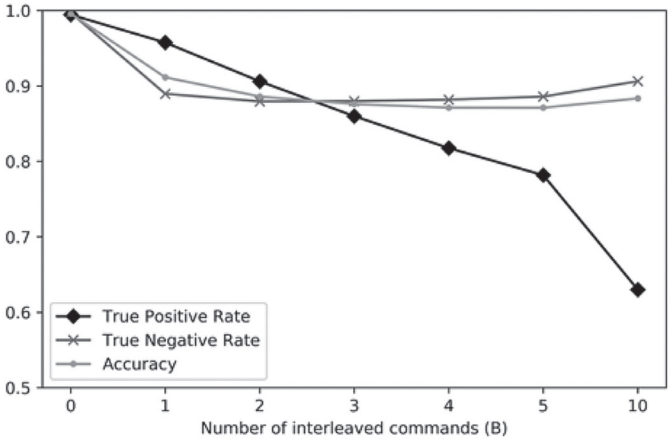
To evaluate this approach, we interleave B benign commands between any two malicious commands from our malicious testing dataset. The performance of the 1-command classifier on the interleaved dataset is equal to the results described in Section 5. B (as expected because the 1-command classifier does not depend on the context; the minor differences can be explained by the larger number of benign commands in the dataset) and are shown in Figure 9.

FIGURE 9. PERFORMANCE OF THE 1-COMMAND CLASSIFIER ON A MALICIOUS DATASET INTERLEAVED WITH B BENIGN COMMANDS (T = 500, M = 3, K = 5). THE RATHER LOW TRUE NEGATIVE RATE, WHICH IMPLIES THAT SOME BENIGN COMMANDS ARE MISTAKENLY CLASSIFIED AS MALICIOUS, IS NOT AN ISSUE SINCE THE 1-COMMAND CLASSIFIER IS ONLY USED FOR PRE-FILTERING BEFORE PROVIDING THE COMMANDS TO THE N-CLASSIFIER FOR FINAL CLASSIFICATION.



In Figure 10, we plot the results for the 1-command classifier combined with the 4-command classifier. As expected, a drop in accuracy occurs as benign commands are interleaved. However, even after interleaving with B = 5 benign commands, the 4-command classifier has an accuracy of almost 90%. This is due to the higher number of benign commands being classified correctly as B increases.

FIGURE 10. PERFORMANCE OF THE 1-COMMAND CLASSIFIER COMBINED WITH THE 4-COMMAND CLASSIFIER ON A MALICIOUS DATASET INTERLEAVED WITH B COMMANDS (T = 500, M = 3, K = 5).



Since the 1-command classifier lets through approximately 20% of benign commands, more benign commands get through as B increases. Therefore, the 4-command classifier has more sequences containing one or more benign commands to classify, reducing its performance.

We conclude from these results that combining the 1-command classifier and the 4-command classifier makes sense to minimize false positives. However, using the 1-command classifier directly minimizes false negatives.

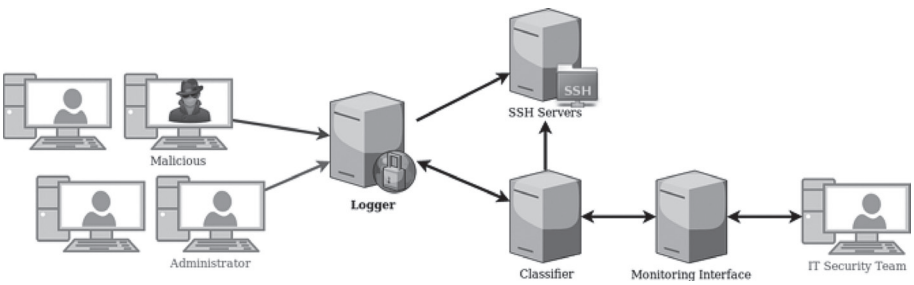
6. USE CASES

In this section, we propose two use cases for which our system is useful: (i) as a passive monitoring tool to protect SSH servers, and (ii) for continuous authentication.

A. Passive Monitoring

The primary use case of our system is for passive monitoring of SSH servers. As illustrated in Figure 11, this tool would be composed of the following components: (i) a *logger* of SSH sessions to record executed commands (e.g. using an SSH transparent proxy, by adapting an SSH server like OpenSSH [18], or by using a tool such as Snoopy Logger [19]); (ii) a *classifier* that receives the stream of commands recorded by the logger and applies our system on them. It should also update the models as they become stale over time; and (iii) a *monitoring interface* for the IT security team to configure the system and observe triggered alerts. However, due to the nature of the malicious data we used (bots targeting a honeypot) automated attacks are more likely to be detected.

FIGURE 11. USING OUR SYSTEM FOR PASSIVE MONITORING OF SSH SERVERS.



B. Continuous Authentication

As another use case, an extended version of our system could be used for continuously authenticating users. Instead of classifying sessions as malicious or not, the system

could check whether the commands in the current session are similar to those in past sessions of the same user. This setting is similar to previous works [20] [21], which shows that it is possible to differentiate between users based on keystroke dynamic analysis. Applying this approach to SSH logs would allow the detection of attackers who use stolen credentials of legitimate users. In this case, models should be trained per user.

7. DISCUSSION

In this section, we discuss the answers to our research questions and possible extensions as well as limitations of our system.

A. Research Questions

In the following paragraphs, we answer the three research questions from Section 1.

Do individual commands contain enough information for a classification between malicious and benign purposes?
The results for our 1-command classifier show that it is possible to distinguish between malicious and benign commands from our datasets with a TPR of more than 98% and a TNR of more than 79%. While this is a good starting point, our N-command classifier proves that analyzing multiple commands can lead to significantly better scores.

How many commands need to be analyzed to accurately identify malicious remote shell sessions?
Our evaluation of the N-command classifier shows that 4 commands are enough to accurately distinguish between our two datasets (with TPR 99.4% and TNR 99.7%). Compared to the 1-command classifier, the TNR therefore increases by approximately 20% when analyzing 4 commands (and thus is not significantly increased when analyzing more than 4 commands).

Can we detect attackers who try to obscure their commands?
We observed that when attackers interleave their malicious commands with benign ones, the accuracy falls. However, by combining the 1-command classifier and the 4-command classifier, the overall accuracy is still over 90%. We discuss additional countermeasures as well as other strategies for attackers in the next section.

B. Evading the Classifier

In this paper, we analyzed individual commands and sequences of commands.

However, attackers who know that such a system is being used could employ multiple techniques to evade it.

The first technique that we reproduced here was to interleave benign commands between each malicious command. As expected, the classification performs significantly worse than without interleaved commands. One of the possible solutions against this is to ignore commands that do not change context during the classification (i.e., non-malicious commands such as “ls” could no longer be used to interleave malicious commands). Another approach is to combine our system with keystroke dynamics analysis [20] [21] to detect compromised sessions or accounts.

Another circumvention strategy is to write malicious commands in a script file and execute it or to define aliases for malicious commands. In this case, we would still be able to use the classifier presented in this work, although it would require a tool to analyze the script file or the aliases before executing it. This is feasible for shell commands, but rather difficult for arbitrary programming languages, as one would have to classify commands on the system API level.

C. Classifying Unseen Commands

The classification performance depends on the data quality. We require as many classified commands and scripts as possible to train the classifier. Unknown commands might get decomposed into the n-grams that we previously encountered. However, for truly unseen commands, a default classification to malicious could be used, as most of the benign commands would have been observed after enough time. By design, our system is particularly successful in settings where similar malicious commands are executed frequently, as is the case for automated bots.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented a machine learning-based system to distinguish between malicious and benign shell commands and sessions. Our classifier works solely based on the commands that a remote shell user executes and is able to identify benign and malicious commands with more than 99% true positive and true negative rate after observing 4 commands.

We have shown that our approach works very well in a passive setting where the attacker is not aware of our system’s presence. From simulations with a sophisticated attacker who tries to circumvent our system by obfuscating malicious activities, we observed a decrease in the accuracy of our system and discussed possibilities to counteract such attackers.

We see potential future work particularly in two directions: (i) combining our approach with additional measures to identify attackers who try to circumvent the system; and (ii) training our approach on a per user basis to decide whether two remote shell sessions originate from the same user (which would allow us to use our approach for continuous authentication).

REFERENCES

[1] "SSH servers search on shodan.io," [Online]. Available: https://www.shodan.io/search?query=ssh. [Accessed 2 December 2018].

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Level, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas and Y. Zhou, "Understanding the Mirai Botnet," in *26th USENIX Security Symposium*, Vancouver, BC, Canada, 2017.

[3] CA Technologies, "Insider Threat 2018 Report," 2018.

[4] D. X. Song, D. Wagner and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *10th USENIX Security Symposium*, Washington, D.C., USA, 2001.

[5] L. Hellemons, L. Hendriks, R. Hofstede, A. Sperotto, R. Sadre and A. Pras, "SSHCure: A Flow-Based SSH," in *6th International Conference on Autonomous Infrastructure, Management and Security, AIMS'12*, 2012.

[6] A. Sperotto, R. Sadre, P. d. Boer and A. Pras, "Hidden Markov Model Modeling of SSH Brute-Force Attacks," in *Integrated Management of Systems, Services, Processes and People in IT*, pp. 164-176.

[7] D. Ramsbrock, R. B. and M. Cukier, "Profiling attacker behavior following SSH," in *37th Annual IEEE/ IFIP International Conference on Dependable Systems (DSN)*, 2007.

[8] E. Alata, V. Nicomette, M. Kaâniche, M. Dacier and M. Herrb, "Lessons learned from the deployment of a high-interaction honeypot," in *European Dependable Computing Conference (EDCC06)*,, Coimbra, Portugal, 2006.

[9] A. Shabtai, R. Moskovitch, Y. Elovici and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," in *Inf. Secur. Tech*., 2009.

[10] D. Reddy. and A. Pujari, "N-gram analysis for computer virus detection," *Journal in Computer Virology,* vol. 2, p. 231–239, 2006.

[11] J. Choi, H. Kim, C. Choi and P. Kim, "Efficient malicious code detection using n-gram analysis and SVM," in *14th International Conference on Network-Based Information Systems*, 2011.

[12] "Github search for bash history files, requires to be connected," [Online]. Available: https://github.com/search?l=Shell&q=filename%3Abash_history&type=Code. [Accessed 25 November 2018].

[13] "Github API," [Online]. Available: https://developer.github.com/v3/search/#search-code.

[14] "Cowrie Honeypot Github repository," [Online]. Available: https://github.com/cowrie/cowrie. [Accessed 25 November 2018].

[15] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, pp. 2721-2744, 2006.

[16] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Commun. Surveys Tutorials*, vol. 10, pp. 56-76, 2008.

[17] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153-1176, 2016.

[18] "OpenSSH," [Online]. Available: https://www.openssh.com/. [Accessed 10 December 2018].

[19] "Snoopy Logger Github," [Online]. Available: https://github.com/a2o/snoopy. [Accessed 1 March 2019].

[20] F. Monrose and A. Rubin, "Authentication via keystroke dynamics," in *4th ACM Conf. Computer and Communications Security*, 1997.

[21] F. Bergadano, D. Gunetti and C. Picardi, "User authentication through keystroke dynamics," in *ACM Transactions on Information and System Security 5*, 2002.