Machine Learning and Feature Engineering for Detecting Living off the Land Attacks

Tiberiu Boros[1], Andrei Cotaie[1], Antrei Stan[1], Kumar Vikramjeet[2], Vivek Malik[2] and Joseph Davidson[2]

[1]*Adobe Systems, Romania*

[2]*Adobe Systems, USA*

{*boros,cotaie,astan,vikrakum,vivmalik,davidson*}*@adobe.com*

Keywords: Machine Learning, Living-Off-The-Land (LotL), Feature Engineering, Artificial Intelligence, Random Forest, Commands, CommandLine, OpenSource, Linux

Abstract: Among the methods used by attackers to avoid detection, living off the land is particularly hard to detect. One of the main reasons is the thin line between what is actually operational/admin activity and what is malicious activity. Also, as shown by other research, this type of attack detection is underrepresented in Anti-Virus (AV) software, mainly because of the high risk of false positives. Our research focuses on detecting this type of attack through the use of machine learning. We greatly reduce the number of false detection by corpora design and specialized feature engineering which brings in-domain human expert knowledge. Our code is open-source and we provide pre-trained models.

1 INTRODUCTION

Attackers spend a lot of time trying to trick and circumvent sophisticated malware detection algorithms that are present in modern AV software. A particularly interesting method is to rely on binaries and tools that are often part of the base operating system (OS) distribution to perform reconnaissance, privilege escalation and lateral movement. Because it leverages what is already present in the system, this technique is called living off the land (LotL) and it is hard to detect for several reasons:

- **Context**: When analyzed on its own, the execution of a specific standard binary might not provide sufficient evidence of malicious activity. This is the case for most reconnaissance activities which might look like normal applications checking for privileges or trying to find if a service is up or not on a remote host;
- **Scope**: Most of the LotL tools and binaries are normally used by system administrators and power users to achieve goals that are sometimes indistinguishable from those of the attackers (e.g. add a new user or change a password). Also, some legitimate software installers are known to use a similar approach. This renders the line between administrative operations and malicious activities thin;
- **Reliability**: Based on the double scope of these tools, it is likely that the detection mechanisms would generate false positives, which could sometimes block normal operations for legitimate software and cause outages in corporate networks (where there is a higher level of administrative/power user operations than on home systems).

The motivation for the present research comes from Barr-Smith et al. (2021), where the researchers focus on the evasion rate of LotL-based malware, concluding that the detection of LotL attacks is under represented in modern AV software.

In the present work we propose a system aimed at highlighting LotL related-activity on a given host. We mitigate some issues related to data sparsity and false positives (see Section 3 for clarifications) through our feature-engineering process (which is described in section 3.2) and through the way we design our dataset (Section 3.1). Also, we rely on a classical approach to machine learning: feature engineering and well-established classifiers. To account for this we argue that in theory, a deep-learning approach could provide superior results, but the classical process allows for hand-crafted exceptions and it makes it easy to explain the results and see where and why the sys-

tem fails.

Because a large-scale LotL dataset is hard to obtain, we had to build our own corpus in order to train and test the models. We describe the process we used and we include the open-source repositories that we collected malicious examples from. Everything described in this paper is currently freely available as an open-source repository (link provided in Section 6) and a PIP package with pre-trained models (lolc). However, we are unable to share the corpus because the benign examples could contain sensitive information. Instead, we offer insights into our data and tag distribution (Section 5).

2 RELATED WORK

Whether or not we are concerned about LotL detection, intrusion detection systems fall in two main categories: (a) Signature-Based (SB) and Anomaly-Based (AB).

Signature-Based detection relies on identifying patterns of commands that were previously observed to have been misused in other attacks Modi et al. (2013). With a high number of rules comes a high accuracy. However, it generalizes poorly on new attack methods and, to our knowledge, the most efficient way to automatically catch and generate rules for new attacks methods is through the use of honey pots Kreibich and Crowcroft (2004). The later mentioned come with their own ups and downs, which will not be discussed here, except for the fact that, in time, attackers learn how to detect and avoid honey pots themselves.

Anomaly-Based detection relies on modeling, usually through statistics, what can be regarded as normal operations and on alerting whenever a system or application falls outside the normal behaviour Boros et al. (2021); Butun et al. (2013); Lee et al. (1999); Silveira and Diot (2010)[1]. They can either rely on directly modeling a monitored system or on computing the model based on a preexisting datasetDurst et al. (1999), though the later would normally fall into the supervised learning class, rather than the unsupervised (anomaly-based) class. While all AB systems are better at adapting and detecting new attack methods, purely unsupervised methods usually yield a higher number of false positives than their supervised counterparts. On the other hand, supervised methods are better at avoiding false alerts, but they are only as good as the labeled data they are trained on, thus requiring periodic updates and higher maintenance.

Notice: This research only focuses on misuse of LotL binaries and tools. It might seem obvious for most security experts, but we are going to say it anyway: Relying on just one type of detection, including LotLs detection, is not effective from the security standpoint. Only by combining signature based, anomaly based, network profiling, obfuscation detection, system auditing and all the other well-established methods, a Defense in Depth approach, can one obtain a decent level of security and safety.

3 PROPOSED METHODOLOGY

Our methodology can be classified as a special case of signature based intrusion detection that employs machine learning for modeling. There are two main ML related issues we need to take into consideration:

(a) **Data Sparsity**: A naive approach would be to rely on n-gram related features that can be easily extracted from the command-lines in the dataset. This is probably one of the most common approaches when it comes to using ML on text data. However, this results in a rich feature set, which in turn generates data sparsity and enables the model to quickly over-fit the dataset and generalize poorly on previously unseen examples.

(b) **False Positives**: Given the skewed nature of real-life data[2], there is always a "power-struggle" between precision (how many of the examples that are marked as malicious are actually labeled correctly) and recall (how many of the malicious examples from the overall mass are actually identified). This is best captured by specific metrics for skewed datasets, such as the F-Score.

In order to mitigate data sparsity we used a feature extraction scheme that is mostly manually designed and incorporates lots of human-expert knowledge. This keeps the feature-set to a decent size and focuses mostly on features that weigh heavily on the decision of the classifier (as opposed to automatically extracted features, such as n-grams, that would probably introduce a lot of useless information). To address false positives, we manually compile our dataset and we use a huge ratio between benign and malign examples. Also, we repeatedly tweaked our feature extraction scheme and retrained our classifier, until we obtained a high f-score, with the following mentioned:

[1]Some of the cited work, refers to network based anomaly detection. The basic ideas and principles still apply for LotLs detection, but the volume of data is much lower.

[2]In a standard and relatively secure environment, most of the collected data will probably be benign, and not malicious

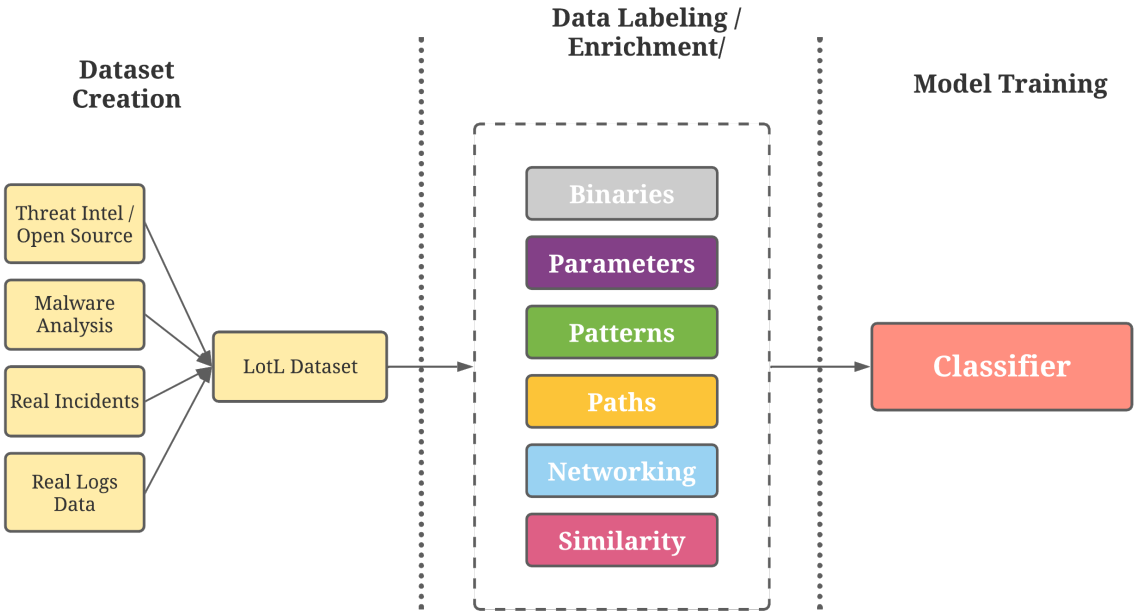

Figure 1: Our proposed methodology: (a) take raw data, (b) add labels for known commands, interesting paths, commandline parameters, known patterns, (c) get enriched dataset and (d) apply classifier on generated labels

whenever the case, we preferred to reduce the recall at the expense of the f-score, rather than have a classifier that generates a lot of noise and ends up being ignored by security analysts over time.

In what follows, we describe our dataset (Section 3.1), discuss our feature extraction scheme (Section 3.2) and evaluate our system (Section 4)

3.1 Corpora Description

Our primarily focus is to build a model that takes as input raw command-lines and is able to detect if they are used in LotL attacks. One challenging aspect of any ML-based approach is getting high quality data to train on. To our knowledge, there is no complete dataset that enables this. However, it is fairly easy to get negative examples from multiple online resources[3]. Thus we started by collecting all available examples of known LotL attacks and we manually filtered the dataset for ambiguous commands, by removing any command that, taken on its own, could be used in both attacks or normal operations, such as:

```
$ stat ~/.ssh
```

```
$ iptables -L -n
```

The total number of negative examples at the end of this process was 1609. For collecting the positive examples we used our own infrastructure logs, removed duplicates and performed random sampling. In order to correctly detect duplicates we used an open-source tool called Stringlifier[4], which is able to spot random strings, GUIDs, numbers, IP Addresses and JWT tokens, thus enabling us to establish when two string instances are actually the same command-line with different parameters. We assumed that randomly selecting the positive examples is likely to capture normal operations and has a less chance of hitting sys-admin/power-user activity. We also preferred an unbalanced dataset (only 0.02% negative examples) because it better reflects real-life data and, with the right feature engineering and tweaking, one can build a robust classifier with a low false-positive rate and good recall.

3.2 Feature Extraction and Modeling

Our feature extraction process is primarily inspired by human experts and analysts. During analyzing

[3]Our data was collected from the following urls: https://gtfobins.github.io/ and https://lolbas-project.github.io/

[4]https://github.com/adobe/stringlifier

Table 1: k-fold validation results for random forest (with 50 estimators), SVM and Logistic Regression with k=5

Classifier	F1-score	Standard deviation	Avg. train time
Random forest	0.95	0.013	18 minutes
SVM	0.95	0.027	3.5 hours
Logistic regression	0.93	0.014	1.2 hours

a command-line, people rely on certain cues, such as what binaries are being used, what paths are accessed, they quickly browse through parameters and, if present, they look at domain names, IP addresses ports. Thus we designed special labels for the following classes of features:

1. **Binaries**: We look for a list of common linux binaries, regardless if they are part of LotL attacks or not. Whenever we encounter a known binary we tag the training example with a specific label that identifies the feature class (for later analysis of the results) and the binary. For example, "`nc -e sh 10.20.30.40 1234`" will receive two labels: `CMD_NC` and `CMD_SH`;
2. **Paths**: We compiled a limited list of interesting paths that are usually in attacks such as temporary locations that don't require any special writing permissions, special devices found in "`/dev/`", known locations for sensitive data (e.g. "`~/.ssh/`"), etc. For example, "`sh -i >& /dev/udp/(...)`", will receive the label `PATH_DEV_UDP`;
3. **Parameters**: Similarly to paths and binaries, we also label common parameters, using a similar schema: command "`nc -e sh 10.20.30.40 1234`" will also receive the label "`PARAM_E`";
4. **Networking**: We look for cues of networking in all command-lines, such as protocol names (e.g. `http`, `ftp`) domain names and IP addresses. For IP addresses we distinguish between private, public, all and loopback. For clarity, the command "`nc -e sh 10.20.30.40 1234`", will receive the label `IP_PRIVATE`.
5. **Pattern Detection**: Using a list of known LotL malicious looking commands we build a series of regexes where purpose is to identify LotL components in commands. For example, let's take the following command:

```
python -c 'import sys, socket, os,
    pty
s=socket.socket()
s.connect((os.getenv(""RHOST""),
          int(os.getenv
              (""RPORT""))))
          )
[os.dup2(s.fileno(),fd) for fd in
    (0,1,2)]
pty.spawn(""/bin/sh"")'.
```

   There are a couple of pieces of information we can extract using regex. We can detect the use of the pty library `r'python.*\-c.*pty'` or we can detect the socket library `r'python.*\-c.*socket'`, the connection itself `r'python.*\-c.*\. connect'` or the shell invocation itself `r'python.*\-c. *pty.*sh'`. The regexes can be more or less permissive but the scope is to generate as much context as possible.

For the command in the last example, the feature extraction phase would generate tags as: *PATH_/BIN/SH, COMMAND_PYTHON, COMMAND_FOR, KEYWORD_-C, KEYWORD_SOCKET, KEYWORD_OS, KEYWORD_PTY, KEYWORD_PTY.SPAWN, python_socket, python_shell, import_pty*.

The classifier is trained to output a binary decision LotL/non-LotL, based on the labels we previously discussed. Not adding any rich text-based features avoids over-fitting the training data. Also, by analyzing the decision and the generated labels for failed examples, it is fairly easy to tweak the feature extraction process and enhance the model.

4 EVALUATION

To evaluate our approach, we performed 5-fold cross-validation. We split the input dataset into 5 subsets that shared the same positive/negative examples ratio. We trained the classifiers on every combination of 4 subsets and report results as the average f1-metric on the 5-th subset (see Table 1). We experimented with multiple classifiers from Sklearn Pedregosa et al. (2011) and we report the results for Logistic Regression, SVM(linear kernel)[5] and Random Forest Classifier Pal (2005).

[5]The convergence speed for the SVM classifier is reported on a system with Intel Sklearn Patch (https://github.com/intel/scikit-learn-intelex).

In terms of accuracy, we obtained comparable results with SVMs and Random Forests. However, the convergence speed for the latter was significantly better, as the implementation allowed training on multiple threads. We noted that the speed for SVMs is reported after installing the Intel Sklearn patch. Without this patch, the classifier didn't complete a single fold after 8 hours.

4.1 Runtime Enhancement

This section covers a runtime enhancement, we included in our system that is neither the scope of our evaluation, nor it was used when we computed the accuracy of the classifier. However, it is part of the python library we created and we feel that it has to be presented.

As a part of the data collection, we have collected a wide range of Living off the Land commands. It is easy for a human to observe their malicious character but it can be hard to detect from a rule-based approach. Slight variations of the command can escape all sort of static rules in place. Pattern based static rules which are too permissive can generate a lot of noise and false positives. One way to try to detect potential Living off the Land commands is to compare new commands with well known LotL commands, like the ones we collected.

One mechanism to do this similarity comparison is by computing the BLEU Scoring distance between two observations. The BLEU score is typically used in Machine Translation to measure the similarity between two proposed translation of a sentence. In our case, we considered using BLEU scoring to express the functional similarity of two command lines that share common patterns in the parameters (Figure 2). In our experiments, we found that a weighted variant of BLEU is well suitable to capture command-line patterns and also LotL commands.

If the score generated is greater than an established score (0.7 in our implementation) the observation is attributed a `KnownLoL` tag. For such observation our implementation overwrites the decision of the RandomForest classifier and marks the observation as bad. This runtime enrichment can be considered a fail-safe mechanism to ensure that well known LotL do not bypass the algorithm in place.

5 INSIGHTS INTO OUR DATASET

Because we are currently unable to share the dataset we find it useful to provide some insights regarding its composition from the tagging perspective. For each unique tag (t) in our dataset, we counted the number of times it appears in benign (negative) examples and how many times it appears with malicious/LotL (positive) examples. We refer to these metrics as $C_{n(t)}$ and respectively $C_{p(t)}$. Given the skewness of our dataset we compute the "log plus 1" equivalents for this metrics for graphical representations as: $L_{np1(t)} = log(C_{n(t)} + 1)$ and $L_{pp1(t)} = log(C_{p(t)} + 1)$. Finally, we add the safe ratio metric as $S_r(t) = (C_{p(t)} + 1)/(C_{n(t)} + 1)$, for highlighting tags that have a bias toward positive or negative examples.

Table 2 shows the metrics for the top-30 high frequent tags in both positive and negative example, and Figure 3 compares the distribution between the two classes on the log scale for the top-30 frequent tags. Similarly, Figure 4 shows the same distributions but for tags that have a high occurrence rate in positive examples (ordered by $C_{p(t)}$ descending) while Figure 5 highlights tags that have a bias toward positive examples (ordered by $S_{r(t)}$ descending).

Based of Figure 5, one can easily observe that tags such as "lua_socket", "OS_EXECUTE", "PTY_SPAWN", etc. have a high tendency to appear in LotL attacks. One notable outlier is "COMMAND_RVIM" which can be observed in a significant number of benign examples. However, Figures 3 and 4 show an overwhelming mostly benign tendency for the tags they highlight, which suggests that these tags can never be used as sole indicators for LotL activity.

The final metrics that we present about the dataset is the average number of tags, 1.73 for benign examples and 6.40 for malign commands. This shows that our tags primarily target keywords and commands that are found in LotL attacks. Interestingly, the maximum number of tags generated for a benign example is 30 (a long automation script), whereas the maximum number of tags generated for a malign example is only 19 (also a long script).

6 CONCLUSIONS AND FUTURE WORK

We addressed the issue of detecting Living off the Land attacks and introduced a ML based method to handle this. The source code is freely available on GitHub[6] and we provide pre-trained models (for the Random Forest Classifier) as well as PIP packaging for easy installation and integration with other projects.

This is only one small step in the direction of han-

[6] https://github.com/adobe/libLOL

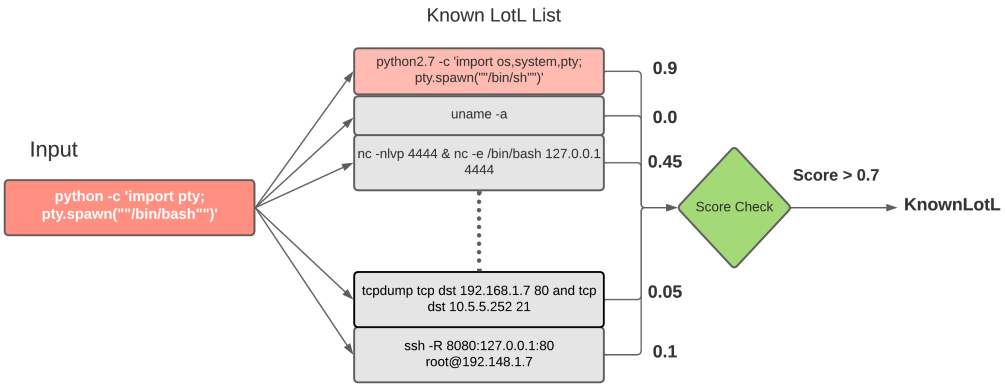

Figure 2: BLEU scoring strategy. Compute BLEU of a new observation using a known list of Living off the Land commands. Attribute a KnownLoL label if the score provided is greater than a specified threshold

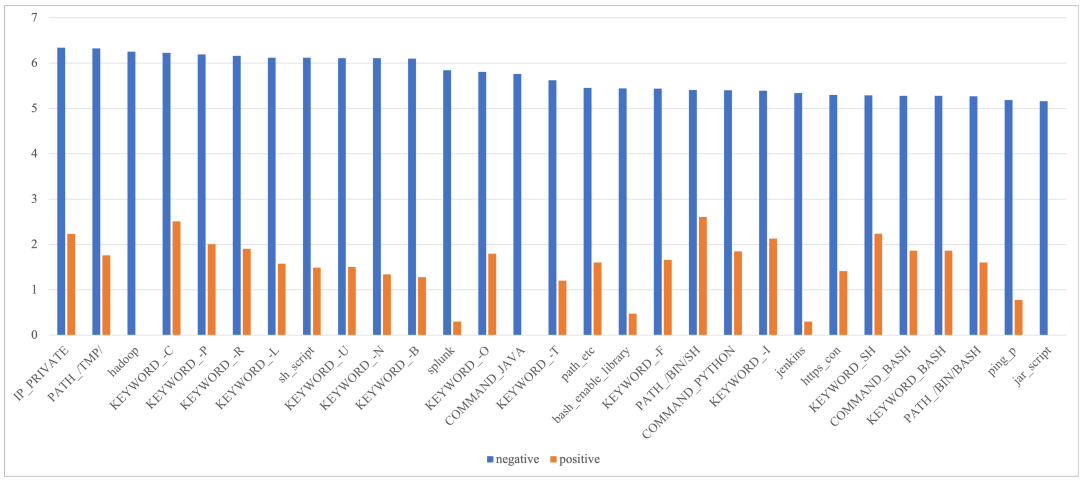

Figure 3: The distribution of most frequent tags between negative(benign) and positive(malicious) examples, computed on a log scale

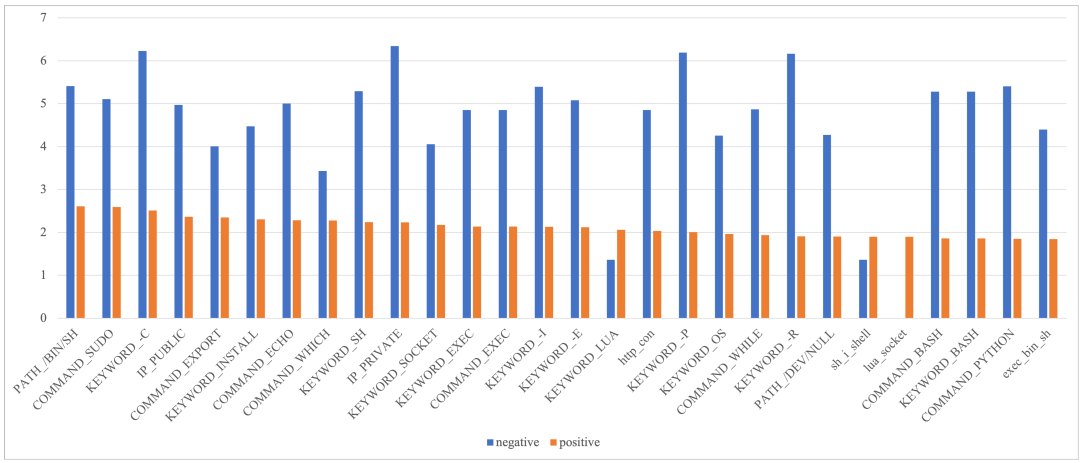

Figure 4: The distribution of most frequent positive tags between negative(benign) and positive(malicious) examples, computed on a log scale

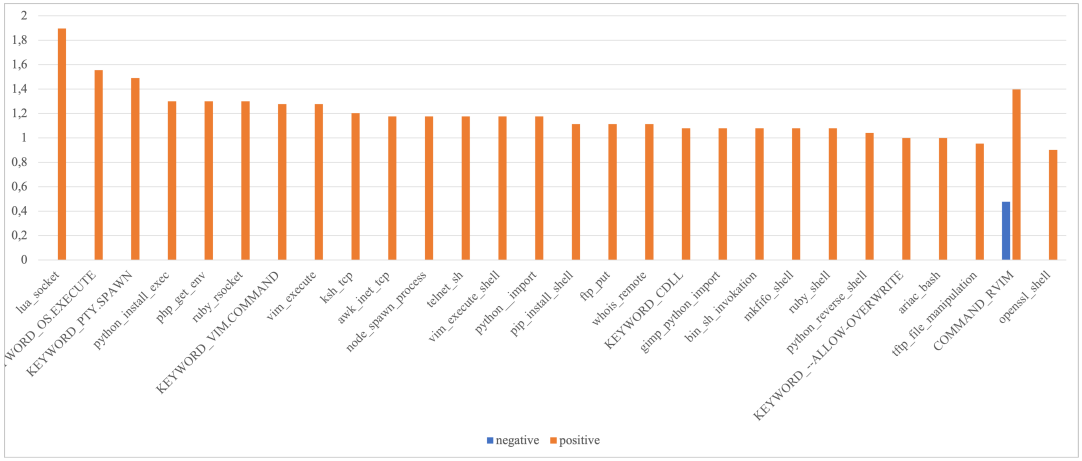

Figure 5: The distribution of most frequent positive tags between negative(benign) and positive(malicious) examples, computed on a log scale

Table 2: Statistics related to the top 30 tags based on their frequency inside our dataset

Tag	$C_{n(t)}$	$C_{p(t)}$	$L_{np1(t)}$	$L_{pp1(t)}$	$S_r(t)$	**Total**
IP_PRIVATE	2207350	170	6,343871398	2,23299611	7,74684E-05	2207520
PATH_/TMP/	2138210	57	6,330050559	1,763427994	2,71255E-05	2138267
hadoop	1814008	0	6,258639437	0	5,51265E-07	1814008
KEYWORD_-C	1690794	325	6,228090955	2,5132176	0,000192809	1691119
KEYWORD_-P	1564014	101	6,194240914	2,008600172	6,52168E-05	1564115
KEYWORD_-R	1466919	80	6,16640643	1,908485019	5,52177E-05	1466999
KEYWORD_-L	1331186	37	6,124239068	1,579783597	2,8546E-05	1331223
sh_script	1325071	30	6,122239477	1,491361694	2,3395E-05	1325101
KEYWORD_-U	1294020	31	6,111941324	1,505149978	2,47291E-05	1294051
KEYWORD_-N	1290764	21	6,110847181	1,342422681	1,70442E-05	1290785
KEYWORD_-B	1264105	18	6,101783493	1,278753601	1,50304E-05	1264123
splunk	697940	1	5,843818711	0,301029996	2,86557E-06	697941
KEYWORD_-O	642962	62	5,808185982	1,799340549	9,79839E-05	643024
COMMAND_JAVA	580453	0	5,763767808	0	1,72279E-06	580453
KEYWORD_-T	421556	15	5,624856305	1,204119983	3,79545E-05	421571
path_etc	286768	39	5,457532202	1,602059991	0,000139485	286807
bash_enable_library	278936	2	5,445506126	0,477121255	1,07551E-05	278938
KEYWORD_-F	277278	45	5,442916979	1,662757832	0,000165898	277323
PATH_/BIN/SH	256605	406	5,409266807	2,609594409	0,001586089	257011
COMMAND_PYTHON	254365	70	5,405459061	1,851258349	0,000279125	254435
KEYWORD_-I	249437	134	5,396962616	2,130333768	0,000541217	249571
jenkins	220177	1	5,342773922	0,301029996	9,08356E-06	220178
https_con	199794	25	5,300584616	1,414973348	0,000130133	199819
KEYWORD_SH	196318	173	5,292962333	2,240549248	0,000886313	196491
COMMAND_BASH	191233	72	5,281565109	1,86332286	0,000381731	191305
KEYWORD_BASH	191233	72	5,281565109	1,86332286	0,000381731	191305
PATH_/BIN/BASH	186057	39	5,269648348	1,602059991	0,000214987	186096
ping_p	154827	5	5,189849504	0,77815125	3,87527E-05	154832
jar_script	144942	0	5,161197246	0	6,89926E-06	144942
COMMAND_GREP	133265	25	5,124719362	1,414973348	0,000195099	133290
COMMAND_SUDO	128947	391	5,110414611	2,593286067	0,003039985	129338
COMMAND_POSTGRES	129096	0	5,11091615	0	7,74611E-06	129096

dling this type of attacks, but we feel that it is an important one, especially because this is a community contribution.

Currently this is a "state-less" approach, meaning that it analyzes each command on a stand-alone manner. Our future research will extend to processing sequences of commands, as this type of analysis is better suited for detecting complex LotL-based attacks.

Another goal is to be able to share the dataset we created and to provide researchers with a common ground to experiment and compare models. This is an extremely difficult task, since the benign examples require sanitation in order to remove any sensitive data or intellectual property. The alternative is to create an artificial dataset based on out-of-the-box operating system and software installation, but this will likely not accurately reflect custom designed automation, which is present in most enterprise environments.

REFERENCES

Barr-Smith, F., Ugarte-Pedrero, X., Graziano, M., Spolaor, R., and Martinovic, I. (2021). Survivalism: Systematic analysis of windows malware living-off-the-land. In *Proceedings of the IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers.

Boros, T., Cotaie, A., Vikramjeet, K., Malik, V., Park, L., and Pachis, N. (2021). A principled approach to enriching security-related data for running processes through statistics and natural language processing. *IoTBDS 2021 - 6th International Conference on Internet of Things, Big Data and Security*.

Butun, I., Morgera, S. D., and Sankar, R. (2013). A survey of intrusion detection systems in wireless sensor networks. *IEEE communications surveys & tutorials*, 16(1):266–282.

Durst, R., Champion, T., Witten, B., Miller, E., and Spagnuolo, L. (1999). Testing and evaluating computer intrusion detection systems. *Communications of the ACM*, 42(7):53–61.

Kreibich, C. and Crowcroft, J. (2004). Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM computer communication review*, 34(1):51–56.

Lee, W., Stolfo, S. J., and Mok, K. W. (1999). A data mining framework for building intrusion detection models. *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pages 120–132.

Modi, C., Patel, D., Borisaniya, B., Patel, H., Patel, A., and Rajarajan, M. (2013). A survey of intrusion detection techniques in cloud. *Journal of network and computer applications*, 36(1):42–57.

Pal, M. (2005). Random forest classifier for remote sensing classification. *International journal of remote sensing*, 26(1):217–222.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Silveira, F. and Diot, C. (2010). Urca: Pulling out anomalies by their root causes. *2010 Proceedings IEEE INFOCOM*, pages 1–9.