

Deep Dive into Hunting for LotLs Using Machine Learning and Feature Engineering

Tiberiu Boros¹ ^a and Andrei Cotaie²

¹Security Coordination Center, Adobe Systems, Bucharest, Romania

²Security Operations, UIPath, Bucharest, Romania

Keywords: Machine Learning, Feature Engineering, Living Off the Land Attacks.

Abstract: Living off the Land (LotL) is a well-known method in which attackers use pre-existing tools distributed with the operating system to perform their attack/lateral movement. LotL enables them to blend in along side sysadmin operations, thus making it particularly difficult to spot this type of activity. Our work is centered on detecting LotL via Machine Learning and Feature Engineering while keeping the number of False Positives to a minimum. The work described here is implemented in an open-source tool that is provided under the Apache 2.0 License, along side pre-trained models.

1 INTRODUCTION

Living of the Land (LotL) is not a brand-new concept. The knowledge and resources have been out there for several years now. Still, LotL is one of the preferred approaches when we are speaking about highly skilled attackers or security professionals. There are two main reasons for this:

- Experts tend not to reinvent the wheel;
- Attackers like to keep a low profile/footprint (no random binaries/scripts on the disk).

The industry standard approach the LotL detection involves SIEM Alerting (i.e. static-rules). Static rules are often too broad or too limited, thus the output becoming somewhat unreliable. Some well known downsides of static rules:

- (a) They are dependent highly on the experience of the SME (Subject Matter Experts) that creates them;
- (b) They can generate a high number of False Positives (because of the thin line in terms of tools and syntax between sysadmin operations and attacker operations) or miss well known LotL activity;
- (c) Their rules grow organically, to the point where it is easier to retire and rewrite rather than maintain and update.

In our previous work (citation will be added after the blind review process) we introduced a supervised model aimed at distinguishing between normal operations and LotL activity, which was trained on a custom designed dataset. The model was a Random Forest (Pal, 2005) and the dataset was composed of 7.9M examples with a ratio of 0.02% (2/10000) between benign and malicious examples.


Since then, we focused on increasing the size and quality of our dataset (see Section 3) and explored deep-learning alternatives that are able to learn meaningful patterns and latent representations for our corpus/task (see Section 5). Our evaluation shows that the deep-learning approach provides better F-scores for both the initial and updated datasets.

Furthermore we discuss how the enhancements on the model influence the accuracy of the classifier and provide an in-depth analysis on how various feature classes contribute (Section 6). Finally, we present our conclusions and future work plans (Section 7).

2 RELATED WORK

Classical intrusion detection systems, including LotL, fall in three main categories: (a) **Signature-Based (SB)**; (b) **Anomaly-Based (AB)** and (c) **Hybrid-Based (HB)**.

While signature-based relies on predefined patterns (Modi et al., 2013), anomaly-based detection is

^a  <https://orcid.org/0000-0003-1416-915X>

data-oriented. It works by spotting behavioral outliers and by assuming that bad-actor activity will be included in this category (Boros et al., 2021; Butun et al., 2013; Lee et al., 1999; Silveira and Diot, 2010). Both categories show strengths and weaknesses. For instance, in signature-based systems, a high number of rules implies high accuracy, but this approach scales poorly, is high-maintenance and does not cope well with previously unseen attack methods. Thankfully, there are ways to automatically append rules and signatures, including the well-known honey pots strategy (Kreibich and Crowcroft, 2004). On the other hand, because anomaly-based systems rely on data modeling, they can automatically scale to new attack methods. However, not all anomalies are bad actor activity and, as a rule of thumb, the number of false-positives is high for this class of detection, often hinging production.

One class of hybrid methods refers to machine learning supervised classification. ML is hard to include into one of SB or AB categories, since it shares similar traits with both. Given that the classifiers work on anything from raw data to engineered features and that they require labeled data, they fit well into the HB category.

3 DATASET

Our initial dataset was composed of 1609 examples of LotL commands compiled from open data-sources¹ and 7.9M benign examples, which were obtained by random sampling² logs from our infrastructure. Since then, we focused on further refining and increasing our corpus for the second iteration of our models.

The present version of the dataset contains 1826 LotL examples and 24M benign examples.

There two important notes about the skewness of our dataset:

- The ratio between malicious and benign examples is closer to real-life scenarios - in practice, you don't have 1/2 split between LotL activity and normal operations;
- By preserving the same ratio during training, validation and testing, we ensure that reported results are close to how the classifier behaves in production environments.

¹<https://gtfobins.github.io/> and <https://lolbas-project.github.io/>

²Random sampling was used to reduce the risk of including actual LotL activity as benign

Currently, we are unable to share the dataset, because it likely contains sensitive information. However, we are working on a public version of the dataset that can open-sourced. This would likely provide a common testing and reporting grounds for researchers. Until then, we are only able to offer some information regarding the composition of our dataset. Figure 1 shows the distribution between benign and LotL examples on a log scale, for the top 10 commands based on three metrics:

- commands that mostly appear in benign examples: `java`, `postgres`, `mv`, `ps`, `chown`, `sleep`, `sshd`, `docker`, `vi`, `du`;
- commands that mostly appear in LotL examples: `rvim`, `rlogin`, `masscan`, `byebug`, `socat`, `dirb`, `cobc`, `ghc`, `ltrace`, `nc`;
- ambiguous commands that appear in both types of activity: `jrunscript`, `xxd`, `mknod`, `tftp`, `mkfifo`, `smbclient`, `aria2c`, `which`, `whois`, `rvim`.

4 FEATURE EXTRACTION

Given the sparsity of command lines, parameters and attributes, performing adequate feature extraction is crucial to preventing the over-fitting of the model. This is primarily achieved by observing and capturing re-occurring security-related events and avoiding features that are based on rare yet discriminative patterns that would likely led to poor generalization of the machine learning algorithms.

We distinguish 5 classes of features that we use in our approach, which are based on (a) binaries, (b) parameters, (c) paths, (d) networking and (e) LotL Patterns. Although they are well described in our previous paper, for completeness we will present them here as well:

- Binaries.** Binaries carry a significant weight in determining if LotL activity is happening on a station, since they capture the capabilities of a command line. For instance, “netcat” usually means bi-directional network communication capabilities, “tcpdump” means monitoring capabilities and “whoami” indicates standard reconnaissance capabilities. Of course, whether these tools are actually used for malicious purposes or if they can be successfully exploited, depends on the context. For a complete set of commands that are used as features, see Table 2;
- Parameters.** Most of the tools we monitor are multipurpose and their parameters help determine

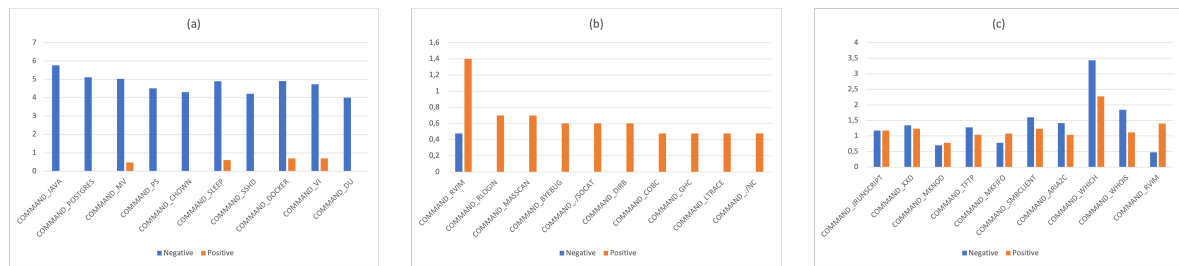


Figure 1: Command (binary) distribution in the dataset: (a) mostly used in benign activity; (b) mostly used in LotL activity; (c) ambiguous usage.

Table 1: List of paths that are included in the feature-set.

/dev/mem	/dev/tcp	/dev/udp	/dev/kmem
/dev/null	/inet/tcp	/bin/sh	/inet/udp
/etc/crontab	/var/spool/cron	/bin/bash	/etc/passwd
/etc/issue	/etc/shadow	/proc/version	/etc/ssh/ssh_config
/etc/services	/etc/network/interfaces	/etc/sysconfig/network	/etc/resolv.conf
/etc/fstab	/etc/group	/etc/hostname	/etc/hosts
/etc/inetd.conf	/boot	.ssh/	/etc/fstab
/etc/profile	/bin/bash	/tmp/bash	/etc/sudoers
/etc/vncpasswd	/proc/	/var/tmp/	/dev/tty
/bin/ssh	/tmp/	/usr/bin/yum	

Table 2: List of commands that are included in the feature-set.

cat	chmod	cp	curl	sqlite3	postgres	runuser
ldapsearch	adduser	export	ftp	history	ping	egrep
bzip	ifconfig	kill	last	mkdir	mv	passwd
lsof	nkf	ps	rm	sshd	tar	node
uname	unzip	nc	useradd	userdel	vi	vim
wget	whoami	sshd	w	socat	telnet	pip
easy_install	nmap	scp	sftp	smbclient	tftp	whois
finger	crontab	cpan	apt-get	byebug	cobc	cpulimit
rvm	ltrace	nohup	top	nice	netstat	find
pwd	ls	ssh	gdb	sudo	netdiscover	chroot
lsbrelease	cut	awk	gawk	chowm	chkconfig	selinux
visudo	runlevel	slapd	openssl	auditctl	usermod	groups
chgrp	cobc	emacs	journalctl	ltrace	rsync	strace
rvm	cpan	mkfifo	perl	lpstat	python	arp
php	ruby	node	jrnscrip	ifconfig	tcpdump	echo
strings	mono	mknod	backpipe	tee	busybox	exec
route	emacs	rlogin	xxd	view	rpcclient	dnsdomainname
aria2c	mysql	debugfs	iptables	masscan	bash	tmux
screen	gcc	grep	for	while	gpg	hostname
sleep	dpkg	which	fstab	env	set	base64
sed	dd	ssh-keyscan	locate	unset	printenv	crash
miner	minerd	dirb	mount	ldconfig	lld	ftp
proxytunnel	yum	mailcap	openvpn	valgrind	rpm	xargs
java	postfix	ansible	git	docker	slsh	hexdump
ghc	chef	salt	showmount	zsh	go	chattr
shalsum	syn.scan	modprobe	lsb-release	du	df	timeout
time	bzip	apt-key	chown	javac	more	ssldump
traceroute	ldapsearch	mtr	ntpdate	csvtool		

if the actual intent is malicious or not. For instance, “netcat” can be used by legitimate scripts to determine if an application is up and running, by checking if a specific port is listening and maybe sending a specific greeting message. On

the other hand, “netcat” used in conjunction with “-e” or with pipes and redirects to “bash”, likely indicates reverse shell activity;

(c) **Paths.** Features based on paths play an important role in determining the legitimate/illegitimate

usage of binaries. From sensitive locations such as `/etc/passwd` to block devices such as `/dev/tcp`, paths are a strong indicator that something less than honest is happening on a box. Of course, there are many paths that can be leveraged on a system, but thankfully, most of the paths used in LotL attacks belong to a finite set (see Table 1 for a complete set of paths we are using in the feature extraction process);

- (d) **Networking.** Communication to other hosts represents an notable use-case for LotL attacks. While the exact IP address, range or ASN is a strong indicator of compromise, it is a highly volatile information and requires a reliable Threat Intelligence (TI) source of information. Also, it becomes obsolete very quickly. Instead, our networking features provide macro-level information about the nature of communication (internal, external, loop-back or localhost references) trading specificity and accuracy for stability over time and TI independence;
- (e) **Well-known LotL Patterns.** Features based on known LotL patterns are regular expressions that look for what can be considered as important signatures that disambiguate the usage of a tool between legitimate and illegitimate. The number of rules is significant and we are unable to include them in the paper. However, if one wants to gain access to them, they are available in the constants file from our public repository.

For every raw command-line, we perform the feature extraction process and enrich the examples with tags that are later used in the classification process. We don't rely on any text-based features in our process (for instance n-grams) in order to reduce data sparsity, avoid classifier over-fitting and gain better generalization on previously unseen examples.

Note 1. The tags or features are discrete values formed by concatenating a "class" prefix with the command, parameter, path, networking or pattern attribute that we detect. For example, if a command line contains `netcat`, `bash`, `/var/tmp` and connects to a public IP address, the extracted features are: `COMMAND_NC`, `COMMAND_BASH`, `PATH_/VAR/TMP` and `IP_PUBLIC`. While we could skip this step and move directly to a ML-friendly representation (n-hot encoding or embedding), we prefer to explicitly do this in our tool, and present the tags along side the classification, because this increases visibility into the dataset and makes it easier for an analyst to understand why a command-line has been classified as LotL.

Note 2. Sometimes, connection information is not present in the command line itself. However, most

implementations for collecting system logs rely on Endpoint Detection and Response (EDR) solutions, that enrich data with inbound/outbound connections. In such cases, it is recommended that the IP address(es) is/are concatenated to the command line, so that the feature extraction process will pick it up and generate the appropriate tag. All our training data has been semi-automatically enhanced with this information (automatically for benign examples and manually for the LotL examples).

Note 3. There is one more tag called `LOOKS_LIKE_KNOWN_LOL`, which is not used in the training process, but it is used during run-time to override the decision of the classifier when a previously unseen command resembles something malicious in our dataset. More details about the tag can be found in our initial work. However, we must mention that it is not used the model evaluation or in the ablation study.

Note 4. Tags for patterns don't follow the same naming convention. Instead, we allow for selecting the name of the tag that is generated whenever a regular expression matches and multiple expressions can yield the same tag.

5 ENHANCED CLASSIFICATION USING CUSTOM DROPOUT

Dropout is a well-established regularization technique (Srivastava et al., 2014), that is preponderantly used in the training process to prevent overfitting. During each training iteration, every unit (neuron) has a pre-defined probability of being masked (dropped out) or, in some cases, of being scaled up. We extend this framework to perform full input-feature which is aimed at reducing the impact of two major issues:

- (a) Because we have a complex feature extraction process, some of the features might be redundant and increase model instability, since they are not linear-independent: `COMMAND_SSH`, `KEYWORD_-R` and `SSH_R` are usually triggered at the same time;
- (b) Regular expression-based feature extraction is not error-free and some rules might not trigger in all cases, while their binary, keyword and IP-based counterparts will work.

To clarify, by full-feature dropout we mean that we randomly mask input features or their corresponding embeddings and we don't perform any scaling on the features that are left untouched. This way, the model is able to learn to predict whether an example is LotL or not using less features than the superfluous

set we generate in our FE process. This yields a robust model with higher F-scores on previously unseen data (see Section 6 for details).

To assess the performance of our methodology we perform 5-fold validation on our previous dataset (small) and static validation with fixed train/dev/test on our newly created dataset (large)³.

In Table 3, MLP-300-300-300 represents the current model on which we apply full input-feature dropout. The model is a three layer Perceptron with a gated activation function ($\tanh \cdot \text{sigmoid}$). The size of each layer is 300, but the output is halved, since we use 1/2 of the units for computing sigmoid and the other 1/2 for computing \tanh , after which they are combined by element-wise multiplication. As can be observed, the MLP-300-300-300 outperforms all other classifiers on the small dataset by a large margin and by a smaller gap on the larger dataset, probably because there is less chance of overfitting on the later mentioned dataset.

The dropout rates for input and hidden layers is set to 10%. For the layer sizes we used truncated grid-search on the small datasets: we tested the same layer size throughout the model from the set {50, 100, 150, 200, 250, 300, 350, 400, 500} by training on a fixed train/dev/test split of the small corpus.

6 ABLATION STUDIES

We set out to evaluate two aspects of our approach: (a) the effect of full input dropout and (b) the effect of different classes of features (commands, keywords, paths, IP and regular expressions).

In order to see how full input dropout affects the F-score of the MLP classifier on previously unseen data we conducted our experiments on both the small and the large dataset, by training the same classifier with and without full input dropout⁴. In all cases we used a static train/dev/test split, because training the classifier is a time-consuming process. Table 4 shows the the full input dropout versions outperform the non-input dropout experiments. The gap is larger on the smaller dataset, since there is a higher chance of over-fitting the classifier. Using full input feature dropout we achieve F-scores around 0.97 on both datasets showing the dataset size has less impact when this type of masking is applied.

³The size of this new dataset prevented us for reporting k-fold validation

⁴When we removed the full input-dropout we added in-place dropout on the internal layers. The reported results are for 10% dropout, a rate that provided best results

Next, we assess how features belonging to the 5 different classes (commands, keywords, paths, IP information and patterns) contribute to the overall accuracy of the model. The full input-feature dropout scheme is no longer useful for this ablation study and since the MLP is harder to train than the Random Forest, we prefer to use the later mentioned classifier in our experiments. We use the merged training and development set to build the classifier (Random Forest does not require validation) and train 6 models by masking features belonging to each class, with an additional model that only relies on the regular expressions.

Table 4 shows the results for all experiments and the baseline results obtained by training a model on the entire feature set. As seen, commands, keywords, IP information and paths, each taken on its own have relatively small contributions to the overall accuracy, with the smallest measured disruption for IP information and largest for command lines. This is somewhat expected, since there is an overlap between the regular expressions and features belonging to the aforementioned classes. The contribution of regular expressions (i.e. human expert knowledge) is extremely important. Without this class of features, the overall F-Score of the model drops to 79.23. To make sure the model does not only rely on the REGEX feature class, we trained a classifier only on this type of features and we got an F-Score of 87.97. While this is a really high score, it is still nowhere near the full model (96.49), which shows that there is a significant number of features from the other classes that contribute to reaching top-accuracy.

7 CONCLUSIONS AND FUTURE WORK

As discussed in the paper, we made significant efforts to further improve our LoTL classifier and dataset. In this endeavour, we increased the size of the dataset and we introduced a custom dropout strategy, targeted for our overlapping features, which had significant impact on the robustness of the model on previously unseen data.

To better understand the way features influence the model, we trained the same classifier on versions of the same dataset obtained by truncating the feature set, the results showing a strong contribution of the “human expert knowledge” to the overall accuracy.

All the work presented here, except the dataset, is reflected in the public Github repository⁵, which we

⁵<https://github.com/adobe/libLOL>

Table 3: F-Score results in the small dataset using 5-fold validation and on the large dataset using static train/dev/test split.

Classifier	Dataset	F1-score	Standard deviation	Avg. train time
Random forest	SMALL	0.9564	0.013	18 minutes
SVM	SMALL	0.9518	0.027	3.5 hours
Logistic regression	SMALL	0.9309	0.014	1.2 hours
MLP-300-300-300	SMALL	0.9722	0.062	6 hours
Random Forest	LARGE	0.9649	N/A	35 minutes
MLP-300-300-300	LARGE	0.9756	N/A	23 hours

Table 4: Results of ablation experiments.

Classifier	Dataset	Experiment	F1-score
MLP-300-300-300	SMALL	Full features and input dropout	0.9717
		Full features and w/o input dropout	0.9421
	LARGE	Full features and input dropout	0.9756
		Full features and w/o input dropout	0.9609
Random Forest	LARGE	Full features	0.9649
		Full features w/o commands	0.9123
		Full features w/o keywords	0.9517
		Full features w/o paths	0.9589
		Full features w/o IP	0.9609
		Full features w/o regex	0.7923
		Only regex	0.8797

actively maintain and in the PIP-packaged version of the code⁶.

REFERENCES

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.

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.

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

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

⁶<https://pypi.org/project/lolc/>