LOLBin detection through unsupervised learning:
An approach based on explicit featurization of the command line and parent-child relationships

A LOLBin/Lib/Script must fulfill the following three conditions to be classified as such: 1. Be a Microsoft-signed file, either native to the OS or downloaded from Microsoft. 2. Have extra "unexpected" functionality. 3. Have functionality that would be useful to an APT or red team.

The LOLBAS project provides a classification of LOLBins into one or more of the following four categories, based on the goal that the attacker is trying to achieve through their use:

- **Proxied Execution, UAC / AWL bypass**: User Account Control (UAC) is a Windows-specific program that allows a program to elevate its privileges. When bypassed, programs can potentially perform administrative-level functionality while remaining undetected. In a similar vein, application whitelisting (AWL) allows system administrators to restrict the set of programs that users can run. Bypassing AWL allows for arbitrary code execution within the given privilege environment of the current user.

- **Download or Upload (Exfiltration)**: This functionality can be used to download malicious payloads (with the LOLBin effectively acting as a stager), exfiltrate data, or establish a command-and-control (C2) connection.

- **Process or Credential Dump**: LOLBins can also be used to dump the memory image of a process. When this process (e.g., lsass.exe in Windows) contains information related to authentication and authorization within the environment, this is called a credential dump.

- **File Copy or ADS (alternate data streams)**: Copy functionality can be used to move files locally or between network shares. Alternate data streams (ADS) can be used to hide data in otherwise benign applications within NTFS file systems.

Looking at LOLBins as well as malware detection in general, there are two choices on how to tackle this problem: rule matching and heuristics *or* machine learning and model-based approaches.

Rule matching and heuristics is an unsatisfactory approach in the long term. LOLBins can include completely unknown or unexpected behavior, which might introduce a lot of false negatives when following a rule-based blacklisting approach - we can only block known LOLBin behavior, neglecting the potentially larger set of unknown LOLBins. On the other hand, manually whitelisting program behavior is prone to a high number of false positives, especially when the underlying workflows within the system change as the whitelisting rules were written with the old workflows in mind.

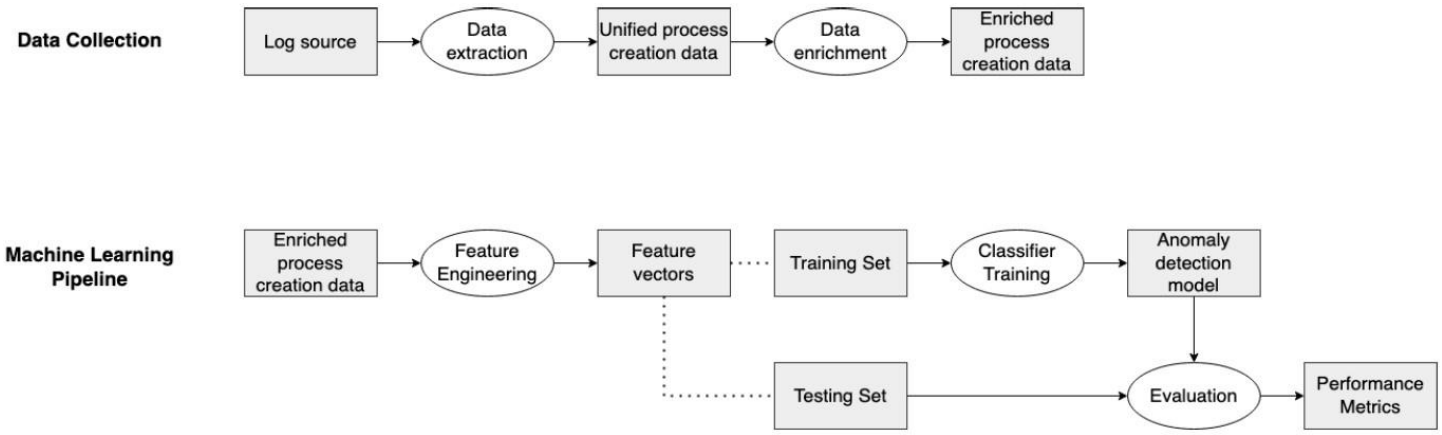
Machine learning has the advantage of being able to create a model fitted to the underlying data as closely as possible, without manual intervention. For supervised approaches, this simplifies the problem from labeling malicious behavior and crafting rules to simply labeling it. For unsupervised approaches, the problem is shifted towards finding an accurate distinction boundary between malicious and benign samples. In both cases, the need for expert interaction is reduced. In the literature, researchers have focused on two primary data sources as the foundation of their detection efforts:
command line analysis and provenance graphs.

Our first and primary goal is to investigate the overall viability of an unsupervised, command line-based approach in the scenario of LOLBin detection, and evaluate whether our reasoning on its benefits is sound (RQ1). Second, we want to understand which features affect the performance of our model the most.

- RQ1: To what extent can a command line-based, unsupervised learning algorithm detect LOLBins?

• RQ2: Which features are most relevant to successfully detect LOLBins?

LOLBins can be used at multiple stages across the intrusion killchain, meaning that the features we will ultimately extract need to provide coverage for all of the described use cases. Additionally, the extracted features will need to be precise enough to allow the classifier to accurately distinguish between benign and malicious executions.



Command Line Features:

- *Documentation for the program was found and usage was undocumented.* We use two sources to provide this information: the official Microsoft documentation for Windows commands and xCyclopedia, a community-driven knowledge base of executables. We differentiate between three possible states: 1) at least one of the parameters we encountered was undocumented, 2) all parameters were documented and 3) we could not find any documentation for the binary.

- *An encountered argument occured in less than 1% of all observations*. Here, we check whether any of the documented parameters found within the command line string are anomalous or not. This is achieved by taking each argument and cross-checking with the data structure if the argument occurs in less than 1% of all executions.

- *Obfuscation-related substring is encountered in the command line*. Here, we check whether the command line string contains any potential obfuscation.

    - Character insertion: Carets (^) double quotes (") and semicolons (;) can be inserted arbitrarily without changing program behavior. For example, calc.exe and ;;;;;;c^^^^^^^al"c.exe will both launch the calculator application. Additionally, double ampersands (&&) can be used to chain commands.

    - Environment variables: Replacing suspicious items in the command line string with previously set environment variables can be used to evade detection. For example, set notregsvr=regsvr.exe allows us to launch regsvr.exe by typing %notregsvr in the command line3.

    - For loops: It is possible to construct complex statements at run time with for loops in order to evade detection. For example, an IP address where one or more octets contain numbers that have an algorithmic pattern such as 234 (incremental) or 164 (square root) can be constructed by a for loop that contains this algorithm.

- *Command line contains executable files hosted at a remote location*. In this context, we define an executable file as any file that can be executed either directly (such as .exe or .dll files) or indirectly via a script processor (such as .hta, .ps1 or .vbs files). We define a remote location as either a remote URL or a remote host.

- *Command line contains paths to an executable which is not located in a privileged directory.* Here, we check if the command line contains code files which are not part of aprivileged (and thus protected) directory such as C:\Windows

- *Command line contains files with extensions.* We define a suspicious extension as an extension, whose file type can be and is being used to execute exploitation steps.

- *Public IP address or a blacklisted hostname is encountered in the command line.* Here, we check whether the command line string contains an untrusted hostname. We define an untrusted hostname as a public IPv4 address (i.e., any IP address not within 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16) or as a website which is listed on the LOTS project, which is a collection of sites commonly used in phishing attacks, C2 connections as well as data download and exfiltration attempts.

- *Command line contains a web or remote keyword.* In this feature, we check whether the command line string contains items from the set of strings {"http", "www.", ".com", "html", "tcp", "udp"} which can be indicative of C2, download or upload behavior.

- *Command line contains a keyword related to process or credential dumping.* In this feature, we check for the presence of process memory dump or credential dump-related substrings within the command line string. Strings include: {"lsass", "samsrv", "hklm\\sam", "winlogon", "netlogon", "kerberos.dll", "dump", ".bin", "ntds"}.

While creating a per-program mapping between feature vectors, we believe that there are additional process-level features outside of the command line that can help improve classification performance. Filename mismatches and signature checks could help us determine whether the executable has been tampered with.

- *There is a mismatch between the program name as called in the command line and the original file name.* Here, we analyze whether the file name in the program's PE header matches the file name of the executable. Mismatches might indicate obfuscation attempts or other tampering behavior.

- *Either of the signatures between parent and child process are expired or otherwise invalid.*

Certain types of processes yield a significantly worse performance and due to the limited information depth of the parentchild information, we were not able to determine the exact false positive and false negative ratios. To improve performance across these more ambiguous/uncertain cases, we thus suggest increasing the depth of the information by constructing process (sub-)trees and elicitating features from multiple ancestors.

Future work could leverage additional provenance features to supplement our command line features in order to more accurately classify uncertain samples and reduce the false positive rate for certain programs. Particularly when paired with alert triaging algorithms such as RapSheet, we believe that using anomaly detection algorithms can take a significant burden off of security analysts and provide them with guidance and support when investigating alerts.