# Automated Search for Leaked Private Keys on the Internet: Has Your Private Key Been Pwned?

Henry Hosseini, Julian Rengstorf and Thomas Hupperich
*Department of Information Systems, University of Münster, Germany*

Keywords: Public Key Authentication, Leakage Detection, Security Services.

Abstract: Public key authentication is widely used alternatively to password-based credentials, enabling remote login with a generated key pair consisting of a private key and a public key. Like passwords, private keys are required to remain confidential to prevent unauthorized access to resources. These secrets might become subject to theft or publicly exposed unintentionally by the key's owner. In such cases, the keys are deemed compromised and need to be revoked and abandoned instantaneously. Unfortunately, it is rarely possible for users to know whether their secret keys have been publicly exposed.
Closing this gap, we introduce a private key leakage checker titled *KeyPwned* crawling the Internet for exposed authentication keys. We present a continuously updated database of leaked keys' fingerprints discovered on websites or in source code repositories. For community-driven enhancement, we allow suggestions of URLs to scan for additional leaked keys, following our standardized process. We furthermore offer users a registration with their public keys to be notified if we detect leakage of their corresponding private key. *KeyPwned* is designed to run as a service following common software design standards, empowering users to verify their keys' confidentiality and take action if a private key has been exposed.

## 1 INTRODUCTION

Authentication methods aside from passwords are used to increase authentication security and diminish the risk of data leakage. A prominent example based on asymmetric cryptography is the SSH protocol, which facilitates public key authentication as a replacement for password-based authentication. In contrast to password-based authentication, public key authentication relies on a stored private key file on the local machine (Lonvick and Ylonen, 2006). Although this authentication mechanism is considered more secure since keys cannot be guessed like passwords, private key files might be leaked accidentally or stolen from a machine. Consequently, for this authentication method to be secure, it is essential to keep private keys secret and the exposure of private keys is deemed a threat to the confidentiality of personal data, one of the most important goals of information security (Avizienis et al., 2004). Thus, it is crucial to know whether an authentication key has been leaked.

For password-based login credentials, there exists an approach named *HaveIBeenPwned* (Hunt, 2022), which recently became fostered by the Federal Bureau of Investigation and open source (Hunt, 2021). The project's website allows checking whether given login credentials appear in known data breaches. With over 11 billion compromised accounts as of October 2021, this service makes an essential contribution to security by providing a powerful service that checks the confidentiality of users' credentials. However, the service of *HaveIBeenPwned* does not take key pair leakage into account. Our work attempts to close this gap by providing key authentication users a means of verifying their private keys' confidentiality. We propose an implementation for a private key leakage checker capable of reporting whether a private SSH key has been publicly exposed. The software developed for this purpose is titled *KeyP̄wned*, in acknowledgment of *HaveIBeenPwned*.

*KeyPwned* has been implemented as a public website[1] on which users can enter SSH fingerprints to check if they occur in the database. By storing the SSH fingerprint of a private key as a public identifier instead of the key itself, confidentiality is ensured. To build up the initial database of *KeyPwned*, we searched for leaked authentication keys manually and derived their fingerprints. The database of leaked keys is extensible, as presumably more keys will become exposed over time. Therefore, *KeyPwned* al-

[1] https://keypwned.uni-muenster.de/



Hosseini, H., Rengstorf, J. and Hupperich, T.
Automated Search for Leaked Private Keys on the Internet: Has Your Private Key Been Pwned?.
DOI: 10.5220/0011308000003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 649-656
ISBN: 978-989-758-588-3; ISSN: 2184-2833

lows users to point URLs out that are searched for leaked keys. We follow a community-driven, collaborative approach to maintain the database as comprehensively as possible by letting users point at potential key leakages. These potential key leakages are verified along with other precautions so that malicious users cannot submit arbitrary input, which may corrupt the database. Thus, our approach follows an advanced, guided search strategy.

To ensure high software quality, we apply a product quality model as defined in the ISO/IEC 25010 standard (ISO/IEC, 2011). By developing and evaluating *KeyPwned* based on this standard, high-quality software and secure utilization by users is ensured.

Our approach raises the question, when exactly a key may be deemed *leaked*. As there is no ultimate definition, for the scope of our work we consider a private key as leaked if it has become publicly available on the Internet without protection, regardless of the duration it has been available.

In summary, the main contributions of our work are as follows:

- We built a database of over 240,000 leaked private key fingerprints.
- We introduce *KeyPwned* as a checking service for private key leakage, allowing users to test if their private keys have been publicly exposed.
- We perform a standardized product quality assessment of our approach based on ISO/IEC 25010.
- We allow a community-driven database extension.
- We offer a public key registration for notifications if the corresponding private key has been exposed.

The next section of this paper describes the software design of *KeyPwned* and the search for publicly available keys. The paper will then go on to evaluating *KeyPwned* and describing the collected data. We discuss our solution in Section 4 and provide an overview of related work in Section 5. The final section summarizes the main findings of this work.

# 2 SOFTWARE DESIGN

The proposed service software comes in two parts. The first part, named private key leak checker, provides the databases and interfaces necessary for users to check their private keys, suggest websites for automated crawling, and register for key exposure notifications. The second part addresses the search for leaked keys, extracting and processing their information for our database.

## 2.1 Private Key Leak Checker

Three user interfaces form the central part of interaction with *KeyPwned*, described in the following. The request interface allows the submission of one or more key fingerprints to receive feedback on whether the private keys were leaked in a data breach incorporated in the database or if they are presumably still safe to use. The request interface is complemented with a report interface to enter a list of URLs that may contain private keys. The software retrieves all possible keys from these URLs automatically and extends the database. As a third user interface, the notification interface provides the users with the possibility to register their fingerprints with their email addresses to receive notifications in case of leakage.

To initially populate the database with leaked private keys, we collected website URLs by search terms indicating the presence of private key files. For this purpose, the *Google Hacking Database (GHDB)*, several websites hosting plain text files and source code, as well as a public *GitHub* activity dataset are used. Regular expressions are tailored to extract all private key data from downloaded web pages, and their SHA-256 as well as MD5 fingerprints are calculated and stored. Note that for confidentiality and ethical reasons, the extracted private keys are not stored themselves. As to password-protected private keys, the fingerprints cannot be calculated without the password.

To illustrate the interaction and architecture of the application, we apply the C4 model (Brown, 2021) for visualizing the software architecture.

### 2.1.1 Request Interface

The main interaction point is a web interface that contains elements such as a text input area, a submit button, and a section with FAQs to explain the application. Users can interact with the application by generating the fingerprints of their private keys on their local machines and copying these fingerprints into the text input area on the website. A request containing the inserted fingerprints is sent to the application's back-end by clicking the submit button. The fingerprint is looked up in the database to find all possible matches. A short manual section includes helpful notes in the form of FAQs about how the application works, how private key fingerprints can be generated, the format the application expects them, and what measures should be taken in case of a leak.

### 2.1.2 Report Interface

As a second part of the application, the report interface allows reporting URLs containing leaked pri-

vate keys to extract and import these keys into the database. The interface allows entering multiple URLs into a text field. A web crawler fetches the web content of the reported URLs nightly and searches for private keys that can be converted into fingerprints for the database. The URLs are automatically checked with the Google Safe Browsing Lookup API to prevent our tool from visiting unsafe web resources. To verify the public accessibility of keys, we do not directly import fingerprints of reported leaked keys by users into the database. Instead, users may point us to URLs containing leaked keys. This process ensures a unified procedure and that the fingerprints in the database remain valid.

### 2.1.3 Notification Interface

To provide users with the possibility to get notified in case of key leakage, we designed a third publicly accessible interface in which users may register their key fingerprints along with an email address. The registration process involves the following challenge-response protocol. First, the user provides their public key and email address. Next, the user receives a generated nonce via the provided email address. The user needs to sign this nonce digitally using the corresponding private key of the uploaded public key. Finally, the resulting signed nonce must be entered in the user interface to confirm that the user owns the private key and the email address. The uploaded public key is automatically discarded after the registration process, saving only fingerprint and email address.

### 2.1.4 Software Architecture

From an architectural point of view, a scalable solution is advisable because it is impossible to determine the future required performance with certainty as the platform grows. Therefore, the software architecture is designed following the service-oriented architecture (SOA) pattern due to its flexibility and popularity. In the realization, this means that services are defined, modeled, and implemented as containers. The Container diagram of *KeyPwned* is depicted in Figure 1 and shows the applied technologies in each container. Users can interact with the system by finding out about their private keys' confidentiality status, by reporting URLs pointing to newly leaked private keys, or by conducting the registration of their private key fingerprint. The software system of *KeyPwned* consists of six separate containers. The three databases contain a) stored key fingerprints retrieved from publicly accessible sources, b) reported URLs pointing to key leaks, and c) registered data of users wishing to get informed about leakage of their keys. Details on the database setup are given in subsubsection 2.1.5. The interfaces are implemented as separate containers based on the user interaction possibilities with *KeyPwned*. The *Downloader* container is used for downloading new keys that were reported through the report interface. The *Notifier* container is a service sending email notifications to the owners of leaked keys that were registered through the notification interface. All containers use database connection clients for interacting with the database container.

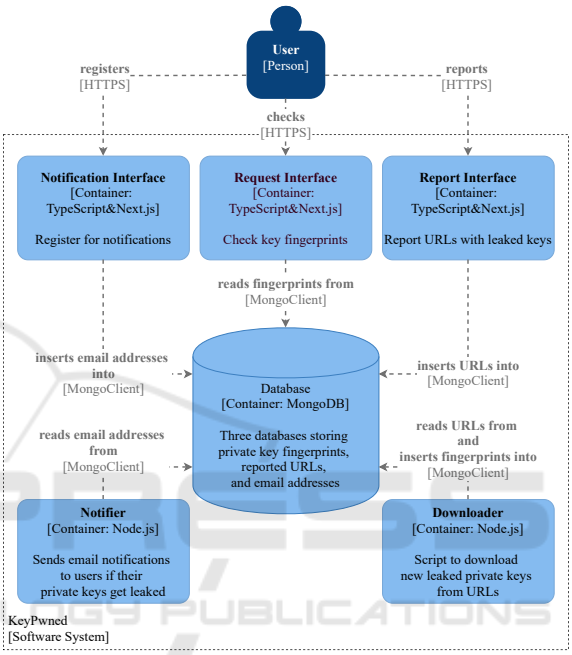


Figure 1: C4 Container diagram showing the software architecture of *KeyPwned*.

The software is deployed using *Docker* containers that can be distributed consistently and platform-independently. The application is developed in *TypeScript*, and runs in a *Node.js* environment. The technology and language decisions are based on maintainability being a major aspect of software quality.

On the front end, the *React*-based *Next.js* framework is used. The *Next.js* framework adds valueable features to the *React* application, allowing the pre-rendering of pages at build time through static site generation (SSG) or rendering at request time using server-side rendering (SSR). An *Express* web server is used, for securing traffic using HTTPS, protecting the page from DoS attacks, as well as defining custom behavior for API requests.

### 2.1.5 Database Setup

Three databases store the fingerprints of leaked private keys, the reported queued URLs pointing to po-

tential leaks, and the registered key fingerprints for notification purposes. The fingerprint database is filled with initial data as described in subsection 2.2.

We selected the document-oriented database software *MongoDB* due to its flexibility and scalability. A database is created, containing a collection that stores the fingerprint documents. Similar design concepts are applied for the databases of reported URLs and registered fingerprints.

Newly added private keys are downloaded and converted to fingerprints. If the fingerprint calculation fails, the key is considered invalid and therefore no invalid keys can be added to the database. Storing fingerprints not only ensures key privacy as it is not possible to exploit our service for retrieving private keys, it also empowers the scalability of our approach as it is more efficient than storing the found keys.

Both SHA-256 and MD5 fingerprint formats are stored in the database to enable users to request information for either format. The source domain, formatted including subdomains, and the date of retrieval are saved as these are displayed to the user who looks up a particular key. The full URL of the key source is only saved for documentation purposes and is not shown to the user to preserve confidentiality.

## 2.2 Key Retrieval

The workflow to build the initial database of leaked private keys encompasses three major steps including the search, extraction, and processing of private keys, as depicted in Figure 2 and described in the following.

### 2.2.1 Search

To build our initial database of publicly accessible private keys, we used *Google Search* and *Google BigQuery*. The search is assisted by three main types of sources that are described in the following.

First, we used the Google Hacking Database (GHDB)[2] and found suitable dorks for discovering publicly available RSA private key files.

Second, in addition to RSA, we extend our search to the DSA, ECDSA, and Ed25519 algorithms. Using the search flag `site:` allows for deeper search on specific websites. This feature is exploited to find private keys on three more websites known for sharing plain text files and source code snippets, namely *Pastebin*, *GitHub Gist*, and *Searchcode*.

Third, we used *Google BigQuery* to search for leaked private keys. One of the publicly available datasets on this platform includes a complete snapshot of more than 2.8 million open-source *GitHub* repositories that can be filtered using SQL queries with regular expressions.

### 2.2.2 Extraction

In the next step, the private keys need to be extracted and temporarily stored before being converted to fingerprints. As Figure 2 shows, this process varies depending on the data sources. Regarding the web pages found via *Google Search* queries to contain private keys, these were extracted from the page contents using regular expressions. These regular expressions are inspired by the ones that Meli et al. used to find private keys on *GitHub* (Meli et al., 2019). URLs reported via the report interface of *KeyPwned* are processed in the same way. For the approach using *Google BigQuery*, this step can be omitted, as the private keys are directly extracted using SQL queries.

### 2.2.3 Processing

Each extracted (not-password-protected) private key was temporarily saved, converted to fingerprints, and inserted into the `fingerprints` collection along with its originating URL. This processing procedure slightly differs depending on the source. Therefore, customized code is developed per data source to calculate the fingerprints and check whether an extracted private key is valid. For the calculation of the SSH fingerprints, the Python library *cryptography* is used (Python Cryptographic Authority, 2022). Additionally, further processing is performed to acquire the other required database fields such as source domain and timestamp of key retrieval.

# 3 EVALUATION

We assess the quality of the software created for our service, and evaluate the data collection process and the resulting database containing fingerprints of publicly exposed authentication keys.

## 3.1 Quality Assessment

The design choices of *KeyPwned* are based on the requirements of the software product quality model given in ISO/IEC 25010. Our software quality assessment was guided by examining each of the eight characteristics of this product quality model.

Evaluating the first characteristic, **functional suitability**, showed that the software fulfills its function to provide users with the possibility to check their private keys against a database of leaked keys and pro-

[2]https://www.exploit-db.com/google-hacking-database

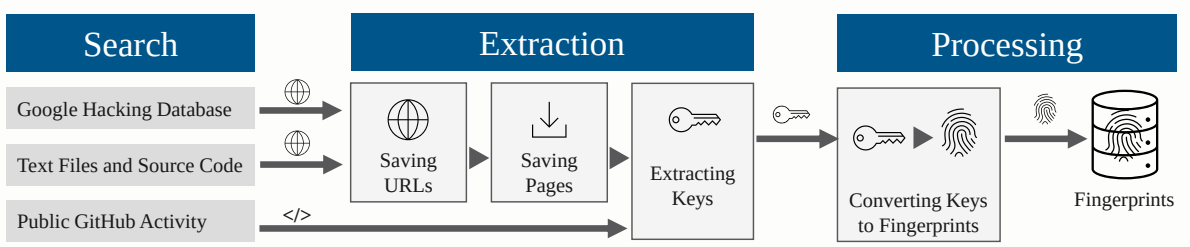


Figure 2: Workflow for the retrieval of private keys and conversion into SSH fingerprints.

vides the correct results for any input. However, the database can never guarantee completeness, as leaked private keys in the wild might not have been discovered and added to the database. In order to extend the database in the future, a report interface is provided that allows users to report new leaks for adding new private keys to the database.

For assessing the second characteristic, **performance efficiency**, we performed a comprehensive study to measure the response times of different input sizes. The test results show that the average response times for sampled fingerprints from the database and the response times for randomly generated fingerprints do not differ significantly.

As for the third characteristic, **compatibility**, the assessment showed that the application could be used flexibly by other software, ensuring coexistence and interoperability. Other applications may use the database and the web application features through REST API endpoints.

Evaluating the fourth characteristic, **usability**, highlighted that the functional appropriateness of the application could be recognized by keeping the UI minimalistic. A classless CSS framework was applied, which provides a simple and accessible interface. The website also provides a simple manual to improve learnability.

Regarding **reliability** as the fifth characteristic, the website instrument modern database and container virtualization software and automatic backups every other week. Furthermore we prevent the exploitation of our service as DDoS relay by process design: users may suggest URLs for crawling. These suggestions are gathered once a day and then scanned in batch. This way, a user may suggest an URL several times but our system does not take action on user command immediately, which could potentially result in flooding other websites and servers.

The sixth characteristic, **security**, is critical to the application's success, which deals with confidential data. The confidentiality of the user's private keys is protected by ensuring that only the key fingerprint and not the plain text private key is transmitted. The report interface for adding new private keys to the database is protected by allowing only URLs to be reported. Private key files without a verifiable source are not considered for the database. For legally securing the application, the IP addresses of requests are stored to hold users accountable for any illegal activity.

Assessing the seventh characteristic, **maintainability**, highlighted that it is particularly important for further extensions to the application. Choosing a modular software design with containers and outsourced components ensures reusability. The choice of *TypeScript* as a language has a strongly positive impact on modifiability, as post-modification bugs are noticed during development.

As the last characteristic, the **portability** of the application was assessed. Since the currently used virtual machine for the proof-of-concept may not be sufficient in terms of resources in the future, the software was designed with the ulterior motive of transfer capability to a more resourceful system without affecting the functionality. By implementing all services as *Docker* containers, it is ensured that the application can easily be adapted to changing hardware requirements and transfer to a new environment.

In summary, the results of the assessment indicate that *KeyPwned* is an application developed with consideration for high software quality standards. It was designed so that scaling and further enhancements can be realized without harming the current functionality.

## 3.2 Data Collection

As a first step, we performed search queries, as described in subsection 2.2. Private keys in five different formats were extracted from these pages. This search covered public key algorithms, including RSA, DSA, ECDSA, and Ed25519.

The first three of the source datasets presented in Table 1 are based on queries from the GHDB. In total, 618 keys were extracted from the pages that were found using these queries. Duplicates were removed. Random inspection of the URLs and contents of these pages showed that the keys are found on various domains, such as privately hosted *GitLab* instances of universities and organizations or servers exposing their entire content, including the `.ssh` folder.

Table 1: Number of keys per source, showing the total number and unique keys found within the respective dataset.

| Source | Results | Unique Keys |
|---|---|---|
| GHDB #6337 | 52 | 11 |
| GHDB #3888 | 321 | 117 |
| GHDB #4455 | 245 | 73 |
| Pastebin | 138 | 85 |
| GitHub Gist | 385 | 226 |
| Searchcode | 2,646 | 880 |
| GitHub | 238,162 | 12,743 |
| Total | 241,949 | 14,135 |

As opposed to the queries from the GHDB, the second block of queries is targeted towards specific domains, namely *Pastebin*, *GitHub Gist* and *Searchcode*. In total, another 3,169 keys were discovered on these pages and deduplicated accordingly.

The third and by far the largest number of keys was found in the public *GitHub* activity dataset on *Google BigQuery*. Here, 238,162 keys were extracted in the process. Although the results contain many duplicates and only 12,743 keys are unique, the high number of repositories containing secrets confirms the findings of related work, which emphasizes the prevalence of the issue of secret leakage in public source code repositories (see section 5).

### 3.3 Leakage Database

By retrieving publicly available private keys, we build up the initial database of *KeyPwned* to evaluate the feasibility of checking private keys for leakage. As of October 2021, the *KeyPwned* database contains 14,135 unique fingerprints of leaked private keys, as presented in Table 1. The study demonstrates that the static *GitHub* activity snapshot is a promising source of secrets, with over 90 % of the final database stemming from *GitHub*. It implies that that secret leakage is a widespread problem in public source code repositories. A variety of public key algorithms and key lengths were detected in the process. Table 2 provides an overview of the most frequent types of keys. Interested users can query the confidentiality status of their fingerprints on a dedicated publicly available website.

Table 2: Overview of the number of keys per public key algorithm and key length.

| Algorithm and Length | Unique Keys |
|---|---|
| RSA-2048 | 6,493 |
| RSA-1024 | 2,774 |
| ECDSA-256 | 1,383 |
| RSA-512 | 698 |
| RSA-4096 | 470 |
| DSA-1024 | 328 |
| other | 1,989 |
| Total | 14,135 |

## 4 DISCUSSION

**Real-world Application.** The use cases of authentication key pairs, e. g., logging into servers, suggests their user group belonging to IT professionals, administrators, and developers. The population of this user group happens to be smaller than average users, who would only check their usernames and passwords for data leaks. Consequently, *KeyPwned* finds applications among the tech-savvy users who would check for the confidentiality of their private keys.

**Impact.** IT professionals, administrators, and developers are usually responsible for operating IT resources, including protecting confidential or personal data. This raises the importance of their authentication credentials' security, including their corresponding private keys, since a compromised admin account would endanger the security of user data. Hence, *KeyPwned* indirectly contributes to the security of users as well. As there have been several studies on the leakage of private keys, the complementary requirement of a trustworthy database of these leaked private keys seems to be essential. We believe that it provides a significant contribution to the security world and closes an existing gap. Still, a longitudinal study to acquire the target group's opinion and feedback about our service is to be conducted.

**Ethics.** Some of the over 240,000 downloaded private keys may be invalid, unused, or only used for testing purposes and not necessarily of high-security importance. However, this could only be tested by performing brute-force login attempts with discovered keys. We did not attempt to use any of the exposed private keys, e. g.to authenticate at publicly reachable services. Furthermore, for the final database, only the fingerprints of private keys are kept to decrease the risk of a potential data leak at our side and ensure general anonymity regarding leaked keys.

**Disclosure Process.** Our approach follows the principle of self-check, as users may check the confidentiality of keys themselves. The more informative option is the registration of key fingerprints and automated notification in case of leakage. This practice requires an upfront registration and storage of personal data, i. e.an email address for each key fingerprint, and, hence, abdicates anonymity.

**Future Improvements.** The current work focuses on SSH keys as the most prevalent type of authentication keys. However, the implementation allows abstraction and arbitrary key formats with public information in the keys' headers. Taking more key types into account is a planned enhancement to achieve a well-sorted database with a larger coverage.

The extension of the key fingerprints' database currently relies on our own automated scans and user-driven suggestions. In the future, this could be enhanced by direct contributions of leaked keys' fingerprints to the database. On the one hand, such a manual insertion of leaked keys' fingerprints as a bulk by trusted third parties could improve the coverage of our service as system administrators could directly report leaked keys, even if the keys are not yet observed on the Internet. On the other hand, it is important to keep a unified procedure and verify key leakage to ensure the database's validity.

We currently do not consider whether a leaked key is still in use for authentication, but only register if it has been exposed publicly. For gaining more insight on the key leakage threat, a future enhancement could be an extension for registered users to flag their keys as abandoned or revoked. This way, it would be apparent if a leaked key is still in use, posing a security risk, or this risk has been mitigated already.

Ultimately, a key testing API would allow integration in authentication services. During a key-based authentication, an automated check of whether the used key has been publicly exposed would be performed using this API. If so, users and administrators can be made aware and revoke leaked keys immediately. This way, our service would contribute to other services' security directly.

# 5 RELATED WORK

Several studies have examined the problem of secret leakage in public source code repositories. The word *secret* is used as a collective term for all kinds of credentials, including tokens, usernames, passwords, and private keys, among others. Sinha et al. (Sinha et al., 2015) focused on the problem of leakage of API tokens and suggested methods to prevent and handle key data leakage. The study by Meli et al. (Meli et al., 2019) in 2019 characterizes the prevalence and extent of secret leakage in public *GitHub* repositories. Based on a snapshot and six-month recording of newly committed files, their study showed that over 100,000 repositories were affected by secret leakage, and thousands of new credentials got leaked daily. These studies illustrate the still relevant issue of secret leakage in public source code repositories.

*GitHub* has taken measures by including scanning services to detect secrets in repositories (GitHub, 2021). Currently, this service notifies service providers, e. g., cloud providers, of leakage of issued secret authentication tokens. Nonetheless, the *GitHub* scanning approach does not automatically scan for all secrets. The user needs to define custom patterns to scan for other types of secrets that would include not only API keys but also SSH private keys, client secrets, or generic passwords, as the definition of *secret* by Saha et. al. (Saha et al., 2020) suggests.

In addition to the commercial secret scanning solution of *GitHub*, a variety of open-source tools exist designed to scan single code repositories for potential secrets. Three of the most popular ones are *truffleHog* (Ayrey, 2018), *git-secrets* (AWS Labs, 2019) and *gitrob* (Henriksen, 2018), differing in their search mechanisms (regular expressions, entropy checks, and file extensions) and purposes (leak prevention or detection). In contrast to these tools, *shhgit* (Price, 2019) is not targeted towards specific repositories but scans the whole space of *GitHub*, *GitHub Gist*, *GitLab*, and *BitBucket* repositories in real-time. The tool was turned into a commercial solution in 2021. Another commercial solution for detecting leaked secrets is *GitGuardian* which offers free services to small teams and public repositories listed as *GitHub organizations*. While their solution can detect leaked secrets instantaneously, it is not open-source and does not provide the ability to upload fingerprints directly to check their confidentiality status. It is still feasible to find secrets in public source code repositories.

To prevent credential stuffing attacks by reducing the number of active credentials leaked, the current best-practice approach lets users check if their login credentials appear in known data breaches. For this purpose, several services for checking compromised credentials have been developed (Li et al., 2019). The service *HaveIBeenPwned* (Hunt, 2022) was launched by Troy Hunt in 2013 and includes a large database of over 10 billion compromised accounts and nearly 500 websites that suffered from a data breach. This service allows users to check an email address or password in real-time against the database and acquire information on their exposure in any known data breach. Moreover, it is possible to sign up for an email notification service that notifies users when their credentials are leaked in a data breach. Additionally, the service of *HaveIBeenPwned* is provided as a public API integrated into password managers. In a similar approach, the German Hasso Plattner Institute (HPI) has developed a credential checking service called *HPI Identity Leak Checker* which in con-

trast to *HaveIBeenPwned*, does not provide a result in real-time but notifies the user via email to increase confidentiality (Hasso Plattner Institute, 2021).

In reaction to data breaches, *Google* has integrated *Google Password Checkup* (Thomas et al., 2019) into *Google Chrome* browser's password manager, and *Apple* released a similar feature for its built-in password manager *iCloud Keychain*.

Overall, there is a substantial benefit of credential leak checkers and existing services focus on password-based authentication. However, as public-key authentication is a standard authentication method, there is a need for a similar service dedicated to private keys.

# 6 CONCLUSION

Leaked authentication keys are a threat to security and should be revoked immediately. To act fast, it is of the essence to find out if a private key has been publicly exposed as soon as possible. We have demonstrated that scanning the Internet for leaked keys is one way to achieve awareness regarding key leakage. After building an initial database of publicly available secret keys, we implemented a service for users to check their keys while also administrators may use this service to test their clients' keys. However, we only store the fingerprints of discovered keys. The quality of our implementation was measured to common standards.

We aim to achieve a collaboratively built database of private authentication keys deemed insecure as they have been revealed on the Internet with this work. Therefore, the dataset can be extended by submitting URLs that we then scan for leaked keys. We plan to continue this service and make its final implementation available after publishing this work to allow a community-driven, ongoing extension of the dataset and to be up-to-date so that users may check their keys regularly.

# REFERENCES


Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

AWS Labs (2019). awslabs/git-secrets: Prevents you from committing secrets and credentials into git repositories. https://github.com/awslabs/git-secrets. (Accessed on 27/05/2022).

Ayrey, D. (2018). TruffleHog. https://github.com/dxa4481/truffleHog. (Accessed on 27/05/2022).

Brown, S. (2021). The C4 model for visualising software architecture. https://c4model.com/. (Accessed on 27/05/2022).

GitHub (2021). GitHub Docs: About secret scanning. https://docs.github.com/en/code-security/secret-security/about-secret-scanning. (Accessed on 27/05/2022).

Hasso Plattner Institute (2021). Identity Leak Checker. https://sec.hpi.de/ilc/. (Accessed on 27/05/2022).

Henriksen, M. (2018). michenriksen/gitrob: Reconnaissance tool for GitHub organizations. https://github.com/michenriksen/gitrob. (Accessed on 27/05/2022).

Hunt, T. (2021). Pwned Passwords, Open Source in the .NET Foundation and Working with the FBI. https://www.troyhunt.com/pwned-passwords-open-source-in-the-dot-net-foundation-and-working-with-the-fbi/. (Accessed on 27/05/2022).

Hunt, T. (2022). Have I Been Pwned: Check if your email has been compromised in a data breach. https://haveibeenpwned.com/. (Accessed on 27/05/2022).

ISO/IEC (2011). ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

Li, L., Pal, B., Ali, J., Sullivan, N., Chatterjee, R., and Ristenpart, T. (2019). Protocols for Checking Compromised Credentials.

Lonvick, C. M. and Ylonen, T. (2006). The Secure Shell (SSH) Authentication Protocol. RFC 4252.

Meli, M., McNiece, M. R., and Reaves, B. (2019). How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *NDSS*.

Price, P. (2019). eth0izzle/shhgit: Ah shhgit! Find GitHub secrets in real time. https://github.com/eth0izzle/shhgit/. (Accessed on 27/05/2022).

Python Cryptographic Authority (2022). Cryptography. https://cryptography.io/. (Accessed on 27/05/2022).

Saha, A., Denning, T., Srikumar, V., and Kasera, S. K. (2020). Secrets in Source Code: Reducing False Positives using Machine Learning. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 168–175. ISSN: 2155-2509.

Sinha, V. S., Saha, D., Dhoolia, P., Padhye, R., and Mani, S. (2015). Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 396–400. IEEE.

Thomas, K., Pullman, J., Yeo, K., Raghunathan, A., Kelley, P., Invernizzi, L., Benko, B., Pietraszek, T., Patel, S., Boneh, D., and Bursztein, E. (2019). Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*. Google LLC.