Private Password Auditing
Short Paper

Amrit Kumar and Cédric Lauradoux

INRIA, France
{amrit.kumar,cedric.lauradoux}@inria.fr

Abstract. Passwords are the foremost mean to achieve data and computer security. Hence, choosing a strong password which may withstand dictionary attacks is crucial in establishing the security of the underlying system. In order to ensure that strong passwords are chosen, system administrators often rely on password auditors to filter weak password digests. Several tools aimed at preventing digest misuse have been designed to aid auditors. We however show that the objective remains a far cry as these tools essentially reveal the digests corresponding to weak passwords. As a case study, we discuss the issues with *Blackhash*, and develop the notion of *Private Password Auditing*— a mechanism that does not require a system administrator to reveal digests to an external auditor and symmetrically the dictionaries remain private to the auditor. We further present constructions based on *Private Set Intersection* and its variant, and evaluate a proof-of-concept implementation against real-world dictionaries

Keywords: Password, Auditing, Password cracking, Private Set Intersection, Private Set Intersection Cardinality.

1 Introduction

Passwords are pervasive to data and computer security. However, despite their utmost reliance on passwords, users often deliberately choose one which is common and easy to remember. This has been confirmed by the data leakages over the recent past.[1] In addition to revealing the password habits of users, these leakages have further increased the number of dictionaries containing common and weak passwords. These dictionaries form the basis of password cracking tools such as John the Ripper[2] and Hashcat[3]. Furthermore, Narayanan et al. [5] show that as long as passwords remain human-memorable, they are vulnerable to "smart dictionary attacks" even when the space of potential passwords is large.

The advances made in password crackers and the ever evolving dictionaries have forced system administrators to take drastic measures to protect their users. Password auditing is one of them. System administrators periodically audit system passwords to inform users to change their passwords in case they are found to be weak (with respect to the available dictionaries and the cracking tools). Typically, they extract password digests from systems and then they themselves perform an internal audit.

[1] http://bit.ly/19xscQO
[2] http://openwall.com/john/
[3] http://hashcat.net/oclhashcat/

Another alternative is to outsource this task to an expert third-party security auditor or to an in-house security team. Since a system administrator has privileged access to several sensitive user information, revealing the weakness of a user password to him may lead to massive security breach. Furthermore, considering the expertise of external auditors and to ensure transparency of the process, the latter approach to auditing is often preferred.

Several proprietary tools such as `l0phtcrack.com` as well as free softwares have been developed to aid password auditors. Most of these auditing tools go beyond determining whether a password is weak. For instance, they also allow the auditor to verify whether the passwords are periodically changed by the users. Some free softwares, a notable example being *Blackhash* [1], are essentially restricted to knowing whether system passwords are weak. However, these tools can be easily adapted to perform a full scale auditing.

While tools capable of performing full scale auditing require the password digests of all the users, some specialized tools such as Blackhash claim to filter weak passwords without having access to the full digests. Contrary to the claims, we highlight that these password auditing tools, in particular Blackhash require the system administrator to reveal the password digests corresponding to *easy-to-crack* passwords. Eventually, these tools require the administrator to reveal weak passwords. A malicious auditor may use these passwords for his own benefit before reporting its potential weakness to the administrator.

To this end, we present *Private Password Auditing*: a mechanism that allows a user or a system administrator to filter weak passwords from the password digests without revealing the digests to the auditor. Furthermore, the dictionaries used for auditing remain private to the auditor. The presented tool relies on *Private Set Intersection* [4] and *Private Set Intersection Cardinality* [3]. We finally evaluate the performance of a proof-of-concept implementation of the tool. This leads us to the conclusion that in the general auditing scenario, private password auditing tools are practical.

2 Password Auditing

Password auditing may be considered as a preventive mechanism to resist password cracking tools. In its restricted form, password auditing consists of determining whether any of the system passwords are weak and hence susceptible to cracking tools. This is essentially performed with the help of an auditor who uses dictionary based tools to filter weak digests. In the following we present existing approaches to password auditing of this kind and analyze their weaknesses.

2.1 Naive Approach

A naive approach to password audit would typically involve extracting password digests from systems and then sending them to a third-party security auditor or an in-house security team. The auditor relying on tools such as John the Ripper or Hashcat may easily uncover potentially weak passwords. However, such an approach ensues serious risks. The password digests may be lost or stolen from the security team. Furthermore, a rogue security team member may secretly make copies of the password digests and

may mount *pass-the-hash attacks*. Worse, some of these digests may correspond to easy-to-crack passwords. The auditor may recover in clear the weak passwords and use it for malicious purposes before reporting it to the system administrator.

Consequently, it is hard to guarantee that the password digests are handled and disposed of securely and that access to the digests is not abused. Indeed, only the system administrator and his team should have access to password digests. Extracting the digests and giving them to someone else fundamentally compromises the security of the system.

2.2 Auditing Without Full Hashes

This kind of auditing checks system digests for weak passwords without actually having access to the full digests. A notable example is Blackhash [1], which is based on Bloom filters [2]. In the following we briefly describe Bloom filters and in the sequel we present Blackhash.

Bloom Filter. Bloom filter [2] is a space and time efficient probabilistic data structure that provides an algorithmic solution to the *set membership query problem,* which consists in determining whether an item belongs to a predefined set.

Classical Bloom filter as presented in [2] essentially consists of k independent hash functions $\{h_1, \ldots, h_k\}$, where $\{h_i : \{0,1\}^* \to [0, m-1]\}_k$ and a bit vector $\boldsymbol{z} = (z_0, \ldots, z_{m-1})$ of size m initialized to $\mathbf{0}$. Each hash function uniformly returns an index in the vector $\boldsymbol{z}$. The filter $\boldsymbol{z}$ is incrementally built by inserting items of a predefined set $\mathcal{S}$. Each item $x \in \mathcal{S}$ is inserted into a Bloom filter by first feeding it to the hash functions to retrieve k indices of $\boldsymbol{z}$. Finally, insertion of x in the filter is achieved by setting the bits of $\boldsymbol{z}$ at these positions to 1.

In order to query if an item $y \in \{0,1\}^*$ belongs to $\mathcal{S}$, we check if y has been inserted into the Bloom filter $\boldsymbol{z}$. Achieving this requires y to be processed (as in insertion) by the same hash functions to obtain k indexes of the filter. If any of the bits at these indexes is 0, the item is not in the filter, otherwise the item is present (with a small *false positive probability*).

The space and time efficiency of Bloom filter comes at the cost of false positives. If $|\mathcal{S}| = n$, i.e. n items are to be inserted into the filter and the space available to store the filter is m bits, then the optimal number of hash functions to use and the ensuing optimal false positive probability p satisfy:

$$k = \frac{m}{n} \ln 2 \qquad \text{and} \qquad \ln p = -\frac{m}{n} (\ln 2)^2 \quad . \tag{1}$$

Blackhash.

Blackhash [1] is a tool for restricted auditing of passwords, i.e. check for weak password digests in the system file without having access to the full digests. It works by building a Bloom filter from the system password digests. The system manager extracts the password digests and then uses Blackhash to build the filter. The filter is saved to a file, then compressed and given to the audit team. The audit team maintains a set of dictionaries of weak passwords against which the password digests are to be tested. Upon reception of the filter, the auditor simply checks for each entry of the dictionary, whether or not it is present in the filter. If weak passwords are found to be present in

the filter, the security team creates a weak filter of these passwords and sends it back to the system manager. Finally, the system manager tests the weak filter against the system digests to identify individual users with weak passwords.

Bloom filter parameters. The filter size m to store the system digests is 2^{26} bits, and can accommodate up to 1 million password digests. The number of hash functions $k = 2$, and the hash functions employed are either MD4 or MD5. Developers claim to achieve a false positive probability of 0.0008. Clearly, these parameters are not optimal. To achieve a false positive probability of 0.0008 for 1 million digests, one would need a filter of size $14,842,031 \approx 2^{24}$ bits.

Issues with Blackhash. Developers claim that Blackhash does not reveal password digests to the auditor. Hence, it constitutes a better and secure tool compared to the naive approach. Contrary to the claim, using a Bloom filter of password digests instead of full digests does not improve user's privacy. The most serious issue with Blackhash is that the auditor while finding the weak passwords with the help of dictionaries actually retrieves the weak passwords in clear. To paraphrase, Blackhash requires the system administrator to reveal the weak passwords. Furthermore, due to the false positive probability of Bloom filters, strong passwords might get detected as being weak. Keeping the false positive probability extremely low however comes at the cost of space/time required to store/query the filter.

3 Private Password Auditing (PPA)

In the previous section, we highlighted the issues with Blackhash. The most serious one being that, it requires the administrator to reveal weak passwords. To this end, we propose *Private Password Auditing* (PPA), a mechanism which does not require a user or the administrator to reveal password digests while auditing. Two scenarios may be considered where PPA may play important role:

- There is a system administrator with a list of system password digests and wishes to know the ones which correspond to easy-to-crack passwords. Once these passwords are identified, the respective owners are contacted and asked to change their passwords.
- There is a user who wishes to know whether his password digest is easy-to-crack.

We suppose that auditing in both the scenarios is performed with the help of an external auditor who may be malicious and that auditing is restricted to verifying whether provided password digests contain weak ones. We also suppose that the auditor performs a dictionary based password cracking, i.e. the auditor checks whether a password digest corresponds to the digest of a word in the given dictionary (or a set of dictionaries).

Privacy guarantees. In addition to the fact that the user or system digests are not revealed to the auditor, the external auditor himself may not wish to reveal the dictionaries he uses for password auditing. This is usually the case for proprietary tools. Hence, PPA simultaneously ensures privacy for both the system administrator/user and the auditor. The digest(s) hence remain private to the user/administrator and symmetrically, the dictionaries used for auditing remain private to the auditor.

In the following, we present construction of a PPA tool that relies on a primitive called *Private Set Intersection* and its variant. The construction can be seen as an application of private set intersection in password auditing. We succinctly present private set intersection protocols and in the sequel we present its variant called *Private Set Intersection Cardinality*. For each primitive, we discuss its applicability to private password auditing.

3.1 PPA based on Private Set Intersection

Private Set Intersection (PSI) considers the problem of computing the intersection of private datasets of two parties. The scenario consists of two sets $\mathcal{U} = \{u_1, \ldots, u_m\}$, where $u_i \in \{0,1\}^\ell$ and $\mathcal{DB} = \{v_1, \ldots, v_n\}$, where $v_i \in \{0,1\}^\ell$ held by a user and the database-owner respectively. The goal of the user is to privately retrieve the set $\mathcal{U} \cap \mathcal{DB}$. The privacy requirement of the scheme consists in keeping $\mathcal{U}$ and $\mathcal{DB}$ private to their respective owner. There is an abounding literature on novel and computationally efficient PSI protocols. The general conclusion being that for security of 80 bits, protocol by De Cristofaro et al. [4] performs better than all other protocols, while for higher security levels, other protocols supersede the protocol by De Cristofaro et al.

PSI provides a primitive to design a PPA tool in the first scenario where a system administrator has a list of system digests and wishes to know the digests which correspond to weak passwords. We suppose that the auditor has a dictionary of weak digests $\mathcal{DB} = \{w_1, \ldots, w_n\}$ and the administrator owns the digest set $\mathcal{U} = \{d_1, \ldots, d_m\}$. Then by invoking a PSI protocol on the sets, the administrator may know the digests which are easy-to-crack. The security of PSI ensures that the sets remain private to their respective owner.

3.2 PPA based on Private Set Intersection Cardinality

Private Set Intersection Cardinality (PSI-CA) is a variant of PSI where the goal of the client is to privately retrieve the cardinality of the intersection rather than the contents. While generic PSI immediately provide a solution to PSI-CA, they however yield too much information. While several PSI-CA protocols have been proposed, we concentrate on PSI-CA protocol of De Cristofaro et al. [3], as it is the most efficient.

PSI-CA builds a PPA primitive in the single digest scenario, where a user wishes to know if his password is weak with respect to the existing dictionaries. As earlier, the auditor has a dictionary of digests $\mathcal{DB} = \{w_1, \ldots, w_n\}$ and the user owns a digest d. Clearly, invoking an instance of PSI-CA protocol on the sets, the user may privately know if his password digest is easy-to-crack: if the intersection set is of cardinality 1, then the password digest is weak. The security of PSI-CA again ensures that data remain private to their respective owner.

4 Practicality of PPA Tool

We implemented the PSI protocol by DeCristofaro et al. [4] and the PSI-CA protocol of [3], since they are the most efficient. Recommended parameters of $|p| = 1024$ and $|q| = 160$ bits have been used for PSI-CA, while an RSA modulus of 1024 bits has been considered for PSI. For both the primitives, SHA-1 hash function has used for

signatures. These parameters ensure a security of 80 bits in the *semi-honest* adversary model.

We evaluated PPA tools based on these protocols and compared their performance with Blackhash. The tests were performed on a 64-bit processor desktop computer powered by an `Intel Xeon E5410 3520M` processor at `2.33 GHz` with `6 MB cache`, `8 GB RAM` and running `3.2.0-58-generic-pae Linux`. We have used `GCC 4.6.3` with `-O3` optimization flag. The implementation uses GMP library[4] v4.2.1.

For Blackhash and PSI based tool, we fix the number of system digests to be 59, 169. This corresponds to a representative data provided with the Blackhash source code. In order to evaluate the performance of the techniques, we tested these implementations against real-world dictionaries of varied sizes, from 100 entries up to 14 million entries. The dictionaries are presented in Table 1a.

Table 1: Dictionaries used and the results obtained for 59, 169 system digests.

(a) Dictionaries used.

Dictionary	**# entries**
Top-100	100
John the Ripper (JtR)	3107
Xato Top-10k (Xato)	10,000
Cain & Abel (C&A)	306,706
Dazzlepod	2,151,220
RockYou	14,344,391

(b) Cost incurred by different auditing tools.

		Time					
	Tool	Top-100	JtR	Xato	C&A	Dazzle.	RockYou
	Blackhash [1]	6s	6s	6s	15s	1m	2m
PPA	PSI-CA [3]	47ms	359ms	1s	28s	3m	23m
	PSI [4]	1m	1m	2m	6m	37m	4h

Table 1b presents the results obtained for SHA-1 password digests. We observe that while Blackhash is not privacy-friendly, it is the most efficient. This is due to the time efficiency of the underlying Bloom filters. PPA tool based on PSI-CA is faster than Blackhash for smaller dictionaries since PSI-CA considers only one digest. Moreover, even for moderately large dictionaries (2M), the audit time remains very practical, i.e. 3 mins. PSI based tool incurs considerable cost for large dictionaries. In fact, both PSI and PSI-CA based private auditing against very large dictionaries are suitable in the settings where password auditing is not supposed to be instantaneous, which usually is the case. Indeed a security audit may last for days.

5 Conclusion

In this work, we discussed the issues faced by system administrators in face with malicious auditors. The existing password auditing tools essentially require the system administrator or the user to reveal weak passwords. While password auditing tools like Blackhash may prevent pass-the-hash attacks, they are yet susceptible to revealing weak passwords to the auditor. Considering the extreme sensitivity of passwords, more secure means must be deployed to ensure the privacy of passwords. To this end, we provide a private password auditing tool which does not require the user to reveal the

[4] `https://gmplib.org/`

password digests to the external auditor. Symmetrically, the auditor keeps his dictionaries private. The tool is based on private set intersection and its variants. An evaluation reveals that privacy friendly tools are practical in scenarios where password auditing is not instantaneous (which usually is the case). We highlight that the primitives used in PPA require heavy public key operations, a future work consists in designing efficient and dedicated PPA protocols relying only on symmetric cryptographic primitives.

Acknowledgements

This research was conducted with the partial support of the Labex PERSYVAL-LAB(ANR–11-LABX-0025) and the project-team SCCyPhy.

References

1. Richard B. Tilley. Blackhash software. `http://16s.us/software/Blackhash/`.
2. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13, July 1970.
3. Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security*, volume 7712 of *Lecture Notes in Computer Science*, pages 218–231. Springer Berlin Heidelberg, 2012.
4. Emiliano De Cristofaro and Gene Tsudik. Experimenting with Fast Private Set Intersection. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 2012.
5. Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 364–372, New York, NY, USA, 2005. ACM.