Protecting accounts from credential stuffing with password breach alerting

Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, and Sarvar Patel, Google; Dan Boneh, Stanford; Elie Bursztein, Google

This paper is included in the Proceedings of the 28th USENIX Security Symposium.
August 14–16, 2019
Introduction
Breach compilations

• Collection 1-5 (2.2 billion credentials)
• Antipublic (450 million credentials)
• Exploit.in (600 million credentials)
Breach compilations

The wide-spread availability of usernames and passwords has trivialized criminal access to billions of accounts.
Previous studies have shown that 6.9% of breached credentials remain valid due to reuse, even multiple years after their initial exposure.
The solution

• User can reduce the hijacking risk by resetting an account’s password.
• Discovering which accounts require a password reset remains critical.
The problem still persists
A more proper solution

- Sites like HaveIBeenPwned and PasswordPing actively check the above breached credential compilation to notify users for a possible breach.
- At present, these services make a variety of tradeoffs spanning user privacy, accuracy, and risk to share private information to malicious users or servers.
The proposed solution

• In this paper the authors propose a new privacy protocol that allows the client to learn if their credentials appear in a breach without revealing the information queried.
Backround
Abstract Protocol

- CreateRequest(u,p) that produces a local state LS and request Req that it sends to the breach alerting service.

- Database of unsafe credentials
  \[ S = \{(u_1,p_1),\ldots,(u_n,p_n)\} \]
Abstract Protocol

- Upon receiving a request, the server accesses its credential store $S$, runs $\text{CreateResponse}(S,\text{Req})$, and sends the resulting response $\text{Resp}$ to the client.

- Finally, the client arrives at a verdict whether the credential queried was exposed through a breach by calculating $\text{Verdict}(\text{Resp},\text{LS})$. 
Abstract Protocol

CreateRequest \( (u, p) \) \( \rightarrow \) LS

Verdict \( (\text{Resp, LS}) \) \( \rightarrow \) \{true, false\}

Req \rightarrow \) CreateResponse \( (S, \text{Req}) \)

Resp \rightarrow \)
Design Principles
Democratized Access

- The breach alerting service should be available to all end users and password providers and not require trust between the involved parties
The breach alerting service should provide the user with actionable security advice such as a password reset prompt.
Breached not Weak

• Alerting should trigger a warning when all the necessary to access the account information is exposed. The system should not be triggered by the use of just a weak not breached credential.
Near real-time

• The time between the client sending a query to the server and learning the breach status of their credentials should be near real time.
Threat Model
Adversarial Client

- The adversarial client has access to their own dataset of breached credentials
- $D = \{(u_1,p_1),..., (u_n,p_n)\}$.

Seeks to learn:

- $u \in S - D$ (e.g., a new email to spam)
- $p \in S - D$ (e.g., a new password to add to a cracking dictionary)
- $(u,p) \in S - D$ (a new credential)
Adversarial Server

The server can learn:

- the client’s **identity** $u$ (enables tracking)
- the client’s **password** $p$ (identifies active usage)
- the client’s **credentials** $(u,p)$
Anonymity Sets

• An anonymity set is a set of values (credentials in this case) that are large enough to prevent the attacker to make a valid conclusion.

• Anonymity sets are denoted with K
 Requester credential anonymity

For each \((u_1,p_1)\) exists a large anonymity set \(K\) containing \((u_1,p_1)\) such that:

- \(\forall (u_2,p_2) \in K:\)
- \(\text{CreateRequest}(u_1,p_1) \approx \text{CreateRequest}(u_2,p_2)\)

As a result clients whose credentials belong to the same Anonymity set produce requests that are indistinguishable to the server.
Responses with bounded leakage

- As L we denote the information leaked.
- The service should bound the L from the membership of other credentials in S.

To do this we need a Simulator that can:

- CreateResponse(S,Req) ≈ Sim(L,Req).

So that the attacker cannot extract info about other \((u,p) \in S\)
Inefficient oracle

• For an attacker to learn user’s credentials or if user’s credentials are breached should be more or equally difficult from making guesses at the login portal.
• \( t(\text{CreateRequest}(u_i, p_i)) > T \) as \( T \) is a period large enough to prevent guesses.
• The same extends to an attacker with direct access to the database.
Resistance to Denial of Service

• A response from the server should not require significantly more computation time than a request from a client.

• As a result the attacker needs to have access to as much resources as the server has.
Tradeoffs of existing schemes
<table>
<thead>
<tr>
<th>Query by</th>
<th>Setup</th>
<th>Actionable, not informational</th>
<th>Breached, not weak</th>
<th>Near real-time</th>
<th>Requester credential anonymity</th>
<th>Inefficient oracle</th>
<th>Bounded leakage response</th>
<th>Resistant to Denial of Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Username</td>
<td>Plaintext</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td>Plaintext</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hash prefix</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain</td>
<td>Plaintext</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Username, then password</td>
<td>Plaintext, hash</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hash, hash</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1**: Summary of protocols supported by HaveIBeenPwned and PasswordPing and their tradeoffs according to our design principles and threat model.
Breach alerting protocol
Two variants

1. Resource-constrained attacker variant
2. Zero-password leakage variant
Basic algorithms

CreateDatabase:
• Prior to any client lookup, the server must construct a secure database containing all known breached credentials.

CreateRequest:
• The client generates a request to the server.
Basic algorithms

CreateResponse:
• The server responds to the previous request.

Verdict:
• Finally the client determines whether their credential was exposed to a breach.
Resource-constrained attacker variant

Algorithm 1 CreateDatabase: Store a blinded and strongly hashed copy of all known breached credentials.

Require: \( S = \{(u_1, p_1), \ldots, (u_n, p_n)\} \), \( b = \text{rand()} \), and \( n = 2 \), a prefix length

1: function CreateDatabase\((S, b, n)\)
2: \hspace{1em} for \((u_i, p_i) \in S\) do
3: \hspace{1.5em} \( u'_i \leftarrow \text{Canonicalize}(u_i) \)
4: \hspace{1.5em} \( H \leftarrow \text{Hash}(u'_i, p_i) \)
5: \hspace{1.5em} \( H^b \leftarrow \text{Blind}(H, b) \)
6: \hspace{1.5em} \( H[0:n] \leftarrow \text{ByteSubString}(H, n) \)
7: \hspace{1.5em} \text{PartitionStore}(H[0:n], H^b) \)
8: \hspace{1em} end for
9: end function
Algorithm 2 CreateRequest: Client query to determine whether a blinded username and password with a cleartext hash prefix was exposed in a breach.

Require: $n$, a prefix length

1: function CreateRequest($u, p, n$)
2:     $a \leftarrow \text{RAND}()$
3:     $u'_i \leftarrow \text{CANONCIALIZE}(u)$
4:     $H \leftarrow \text{HASH}(u', p)$
5:     $H^a \leftarrow \text{BLIND}(H, a)$
6:     $H_{[0:n]} \leftarrow \text{BYTE_SUBSTRING}(H, n)$
7:     \text{LOCALSTORE}(a)$
8:     return HSTSRequest($H_{[0:n]}, H^a$)
9: end function
Resource-constrained attacker variant

Algorithm 3 CreateResponse: Server response for all information known about the cleartext hash prefix.

Require: \( b = \text{rand}() \)

1. function CreateResponse\((H_{[0:n]}, H^a)\)
2. \( H^{ab} \leftarrow \text{Blind}(H^a, b) \)
3. \( S' \leftarrow \text{PartitionLookup}(H_{[0:n]}) \)
4. return HSTSResponse\((H^{ab}, S')\)
5. end function
Resource-constrained attacker variant

Algorithm 4 Verdict: Final client-side verdict for whether a username or password was exposed in a breach.

Require: \( a \), secret key for original request

1: function \( \text{VERDICT}(H^{ab}, S', a) \)
2: \( H^b \leftarrow \text{UNBLIND}(H^{ab}, a) \)
3: return \( H^b \in S' \)
4: end function
Zero-password leakage variant

Algorithm 1 CreateDatabase: Store a blinded and strongly hashed copy of all known breached credentials.

Require:
\[ S = \{(u_1, p_1), \ldots, (u_n, p_n)\} \], \( b = \text{rand()} \), and \( n = 2 \), a prefix length

1: function CREATEDATABASE($S, b, n$)
2: \hspace{1em} for \((u_i, p_i) \in S \) do
3: \hspace{2em} $u'_i \leftarrow \text{CANONICALIZE}(u_i)$
4: \hspace{2em} $H \leftarrow \text{HASH}(u'_i, p_i)$
5: \hspace{2em} $H^b \leftarrow \text{BLIND}(H, n)$
6: \hspace{2em} $H_{\text{part}} \leftarrow \text{HASH}(u'_i)$
7: \hspace{2em} $\text{PARTITIONSTORE}(H_{\text{part}}[0,n], H^b)$
8: \hspace{1em} end for
9: end function
Zero-password leakage variant

Algorithm 2 CreateRequest: Client query to determine whether a blinded username and password with a cleartext hash prefix was exposed in a breach.

Require:
- \( n \), a prefix length

1: function CREATEREQUEST\((u, p, n)\)
2: for \((u_i, p_i) \in S\) do
3: \( a \leftarrow RAND() \)
4: \( u' \leftarrow CANONICALIZE(u) \)
5: \( H \leftarrow HASH(u', p) \)
6: \( H^a \leftarrow BLIND(H, a) \)
7: \( H_{part} \leftarrow HASH(u') \)
8: LOCALSTORE\(a\)
9: return \( HSTSREQUEST(H_{part}^{[0:n]}, H^a) \)
10: end for
11: end function
Zero-password leakage variant

Algorithm 3 CreateResponse: Server response for all information known about the cleartext hash prefix.

Require:

\[ b = \text{rand()} \]

1: function CREATERESPONSE \((H_{\text{part}[0:n]}, H^a)\)
2: \[ H^{ab} \leftarrow \text{BLIND}(H^a, b) \]
3: \[ S' \leftarrow \text{PARTITIONLOOKUP}(H_{\text{part}[0:n]} \) \]
4: return \(HSTSRESPONSE(H^{ab}, S')\)
5: end function
Zero-password leakage variant

Algorithm 4 Verdict: Final client-side verdict for whether a username or password was exposed in a breach.

Require:
   a, secret key for original request
1: function VERDICT($H^{ab}, S', a$)
2: \quad $H^b \leftarrow UNBLIND(H^{ab}, a)$
3: \quad return $H^b \in S'$
4: end function
Analysis
Protocol Deployment

- They implemented the protocol on Google Cloud on the server side and as a Chrome browser extension on the client side.
- In total 667,716 user installed the extension over the measurement period.

<table>
<thead>
<tr>
<th>Duration</th>
<th>Median</th>
<th>90%</th>
<th>95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argon2 username hash</td>
<td>0.1s</td>
<td>0.3s</td>
<td>0.3s</td>
</tr>
<tr>
<td>Argon2 credential hash</td>
<td>4.4s</td>
<td>9.8s</td>
<td>12.7s</td>
</tr>
<tr>
<td>End-to-end API query</td>
<td>8.5s</td>
<td>18.8s</td>
<td>26.9s</td>
</tr>
</tbody>
</table>

Table 2: Time spent performing API operations including hashing and downloading potentially matching breached credentials.
Credential stuffing risk and remediation

- The API recorded 21,177,237 login attempts.
- 316,531 (1.5%) of these attempts involved breached credentials and received a warning.
- This percentage is probably the lower bound of what is happening to the average population as the users who adopted the extensions were probably more security conscious.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension users</td>
<td>667,716</td>
</tr>
<tr>
<td>Logins analyzed</td>
<td>21,177,237</td>
</tr>
<tr>
<td>Domains covered</td>
<td>746,853</td>
</tr>
<tr>
<td>Breached credentials</td>
<td>316,531</td>
</tr>
<tr>
<td>Warnings ignored</td>
<td>81,368 (26%)</td>
</tr>
<tr>
<td>Passwords reset</td>
<td>82,761 (26%)</td>
</tr>
</tbody>
</table>

*Table 3: Summary of the anonymous telemetry data reported over the course of our analysis window from February 5–March 4, 2019.*
Ignoring breached credentials

• 81,368 (25.7%) of the breach warnings were ignored by the users.

Possible reasons:

1. Account of low value not worth the effort of a password reset.
2. Users may not have full control of the account.
3. Users may ignore the warning due to frustration.
Remediation of breached passwords

- 82,761 (26.1%) of the breach warnings resulted to password reset by the user.
- 94% of new passwords were as strong or stronger than the original passwords.
Influence of domains on account security

- Popular sites tend to face less of a threat from credential stuffing.
- Larger security investments towards proactively resetting passwords and helping the users avoid weak passwords.
- 6% of domains with 10,000+ logins have a warning rate higher than 3%.
- 15% of the domains with less than 10000 logins have a warning rate higher than 3%.
Influence of domains on account security

Possible reasons:

1. Finance and government domains show the lowest rate of reused credentials. (0.2–0.3%) (Use of password composition policies)

2. Video streaming platforms and adult websites had the highest warning rate. (3.6–6.3%) (Perceived lack of risk as often multiple users share the same accounts or in the case of adult websites to hide the domain from the warning tray)

<table>
<thead>
<tr>
<th>Category</th>
<th>Domains</th>
<th>Total visits</th>
<th>Breakdown</th>
<th>Warning rate</th>
<th>Ignore rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finance</td>
<td>90</td>
<td>1,684,851</td>
<td>8.0%</td>
<td>0.3%</td>
<td>18.6%</td>
</tr>
<tr>
<td>Email, messaging</td>
<td>47</td>
<td>1,519,795</td>
<td>7.2%</td>
<td>0.5%</td>
<td>14.0%</td>
</tr>
<tr>
<td>Social networking</td>
<td>15</td>
<td>1,191,546</td>
<td>5.6%</td>
<td>0.8%</td>
<td>17.8%</td>
</tr>
<tr>
<td>Shopping</td>
<td>29</td>
<td>1,007,103</td>
<td>4.8%</td>
<td>1.2%</td>
<td>16.4%</td>
</tr>
<tr>
<td>Technology</td>
<td>34</td>
<td>624,702</td>
<td>2.9%</td>
<td>0.7%</td>
<td>16.9%</td>
</tr>
<tr>
<td>Business</td>
<td>12</td>
<td>585,797</td>
<td>2.8%</td>
<td>0.7%</td>
<td>20.3%</td>
</tr>
<tr>
<td>Education</td>
<td>16</td>
<td>261,563</td>
<td>1.2%</td>
<td>0.9%</td>
<td>26.5%</td>
</tr>
<tr>
<td>Gaming</td>
<td>11</td>
<td>201,646</td>
<td>1.0%</td>
<td>0.5%</td>
<td>18.6%</td>
</tr>
<tr>
<td>Entertainment</td>
<td>9</td>
<td>168,565</td>
<td>0.8%</td>
<td>6.3%</td>
<td>27.1%</td>
</tr>
<tr>
<td>Travel</td>
<td>14</td>
<td>138,968</td>
<td>0.7%</td>
<td>1.8%</td>
<td>19.6%</td>
</tr>
<tr>
<td>Government</td>
<td>5</td>
<td>60,967</td>
<td>0.3%</td>
<td>0.2%</td>
<td>16.9%</td>
</tr>
<tr>
<td>News</td>
<td>5</td>
<td>54,864</td>
<td>0.3%</td>
<td>1.9%</td>
<td>20.7%</td>
</tr>
<tr>
<td>Adult</td>
<td>3</td>
<td>50,408</td>
<td>0.2%</td>
<td>3.6%</td>
<td>38.5%</td>
</tr>
<tr>
<td>Other</td>
<td>42</td>
<td>429,786</td>
<td>2.0%</td>
<td>1.0%</td>
<td>17.8%</td>
</tr>
</tbody>
</table>
Conclusion

• The authors demonstrated the feasibility of a privacy preserving protocol that allows the client to check if their credentials were exposed in a password breach without revealing the information queried.

• This protocol is the first step in democratizing breach alerting in order to mitigate account hijacking.