Passbolt: a bold use of HaveIBeenPwned (./passbolt-a-bold-use-of-haveibeenpwned.html)

Date Wed 17 April 2024 By Philippe Teuwen (./author/philippe-teuwen.html) Category Cryptography (./category/cryptography.html) Tags cryptography (./tag/cryptography.html) vulnerability (./tag/vulnerability.html) password cracking (./tag/password-cracking.html) 2024 (./tag/2024.html)

Passbolt, an Open Source Password Manager, is using the Pwned Passwords service from HaveIBeenPwned to alert users if their password is present in a previous data breach. Pwned Passwords API is based on a mathematical property known as k-Anonymity guaranteeing that *it never gains enough information about a non-breached password hash to be able to breach it later*. Sounds good, right?

Introduction

In 2017, Troy Hunt (@troyhunt) introduced in a blog post a service to allow people to check if a password is known to be already present among the 306 million of leaked passwords from various breaches.[1] Of course, it was not recommended to submit your real password or even a hash of it.

The following year, Junade Ali from Cloudflare proposed a very elegant solution to be able to query the service without revealing too much information.[2]

[...] our approach adds an additional layer of security by utilising a mathematical property known as k-Anonymity and applying it to password hashes in the form of range queries. As such, the Pwned Passwords API service never gains enough information about a non-breached password hash to be able to breach it later.

The principle is very simple: compute the SHA1 hash of the password you want to assess and send the first 5 nibbles (i.e. the first 20 bits) of the SHA1 to the service API. In return, the API gives you all the known SHA1 hashes starting with these 5 nibbles. Then you check if your SHA1 appears in the returned list or not.

If yes, it means your password is known to be part of previous breaches and you should refrain from using it.

If not, clearly knowing only 5 nibbles of a SHA1 makes any bruteforce attempt impossible due to the huge number of collisions among password candidates.

Consequently, Troy Hunt announced the deployment of a *k-Anonymity* enhanced version of the Pwned Passwords service with its API v2 and its huge gain on privacy, enabling its use with real passwords.[3, 4]

[...] even if you don't trust the service or you think logs may be leaked and abused (and incidentally, nothing is explicitly logged, they're transient system logs at most), the range search goes a very long way to protecting the source.

Pwned Passwords API v2 & v3

The API is very easy to use. Send your request to https://api.pwnedpasswords.com/range/xxxxx with the 5 top nibbles of `SHA1(your_password)` and see if the remaining part of your SHA1 is present in the results.

Here is an example.

```
$ echo -n p@ssword | sha1sum
36e618512a68721f032470bb0891adef3362cfa9
# => send 36e61 and watch for 8512a68721f032470bb0891adef3362cfa9
$ wget -q -O - https://api.pwnedpasswords.com/range/36e61|grep -i 8512a68721f0324
70bb0891adef3362cfa9 || echo "Not found"
8512A68721F032470BB0891ADEF3362CFA9:21804
```

Hmm, `p@ssword` is not a very good password, it has been seen 21804 times in previous breaches...

```
$ echo -n p@sszort | sha1sum
d452118ad7a65b151ac323e8abab71737896fc46
# => send d4521 and watch for 18ad7a65b151ac323e8abab71737896fc46
$ wget -q -O - https://api.pwnedpasswords.com/range/d4521|grep -i 18ad7a65b151ac3
23e8abab71737896fc46 || echo "Not found"
Not found
```

On the other side, `p@sszort` has never been seen so far (which is quite surprising...)

Since then, an API v3 was released, adding the possibility to request padded answers so the size of the answer, even over TLS, does not become a source of leakage.[5] But as far as we are concerned, the API is identical to the v2.

By the way, checking a password against previous breach corpora is even a NIST recommendation for any service asking you to create an account or change your password on their platform, and historically the reason why Troy Hunt offered the Pwned Passwords service to the community in the first place.[6]

Here comes Passbolt

Passbolt (https://www.passbolt.com/) is a *Security-first, open source password manager* specialized in the sharing of passwords across teams and businesses. It is based on a website you can host yourself (or use their cloud solution) and on a browser plugin which takes care of the cryptography, securing the storage on the website and providing end-to-end encryption across people sharing passwords.

It integrates the Pwned Passwords feature since 2018 (in the v2.0.10) and is very convenient: when typing in a new password, it immediately alerts you if "the password is part of an exposed data breach" and adjusts its verdict while you are typing in more chars for a better password.

Could it be a problem?

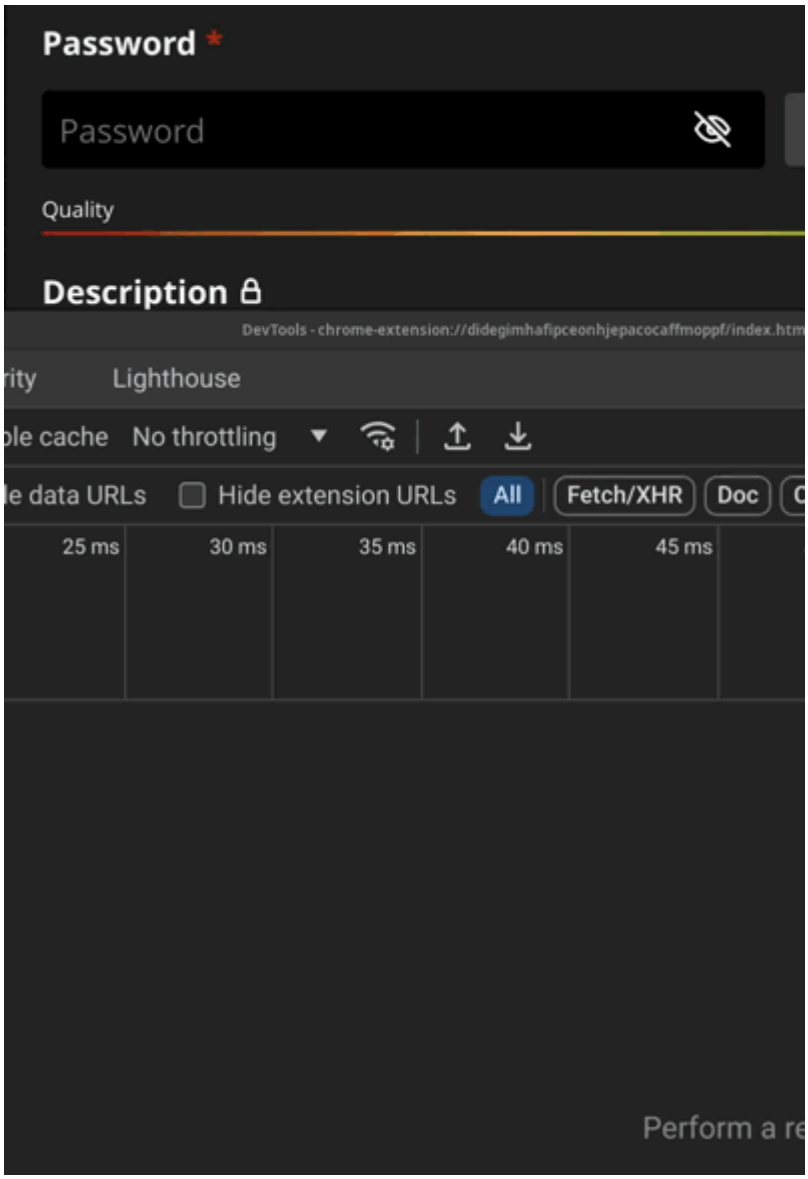
The first naive thought when we saw this feature was that if it emits queries as soon as you type, you could easily guess which first char was typed in. You have about 92 possible chars including alphanumeric and special chars and 5 nibbles of SHA1 is largely enough to discriminate which char was typed in. Then typing the second char would trigger another call to the API and another partial SHA1 would be emitted, again making trivial the recovery of the second char of your password. Etc. no matter how many chars your password ends up to.

Observations

At this stage, we want to validate our hypotheses and observe the calls to the API by the Passbolt browser plugin.

Sniffing the calls to the API on a Chrome-based browser looks like this.

- Go to your Passbolt website instance
- Go to brave://extensions/
- Activate the Developer mode
- Search for the Passbolt extension / inspect views / index.html
- Clicking on index.html will raise a popup with the DevTools
- Watch its Network tab while you are creating a password in your Passbolt website instance



We observe that it invariably tells you that the password is exposed for any password up to 7 chars. Then it starts querying api.pwnedpasswords.com only for longer passwords.

```
1 to 1234567 -> "breached" (without actually checking)
12345678     -> API query with 7C222 (SHA1[0:5] of 12345678)
123456789    -> API query with F7C3B (SHA1[0:5] of 123456789)
123456789A   -> API query with BE472 (SHA1[0:5] of 123456789A)
123456789AB  -> API query with 4A3C4 (SHA1[0:5] of 123456789AB)
```

The requests are done 300 ms after last keystroke. So, if someone types at most 3 chars per second, we get all the intermediate queries. For faster typing, we will miss some intermediate queries. This small delay is due to the use of a debounce function (from npm `debounce-promise` package).

Attack feasibility

At first, an attacker would only see the first partial hash of a 8-char password. There are still quite a number of candidates. But then when seeing the second request very close to the first one, she should just look a which 8-char candidates, extended by a single char, match the second partial hash. And so forth.

We can estimate the complexity of recovering a password from such partial hashes.

Assuming that the user is typing a new password generated perfectly randomly from a charset of 92 numbers, letters and symbol (a task virtually impossible for a human BTW).

- After 8 chars, the password entropy is $log_2(92^8) = 52.2$ bits and we learned $log_2(16^5) = 20$ bits.
  The entropy is reduced to 32.2 bits.
- After 9 chars, the password entropy is $log_2(92^9) = 58.7$ bits and we learned $2\ log_2(16^5) = 40$ bits.
  The entropy is reduced to 18.7 bits.
- After 10 chars, the password entropy is $log_2(92^{10}) = 65.2$ bits and we learned $3\ log_2(16^5) = 60$ bits.
  The entropy is reduced to 5.2 bits.
- After 11 chars, the password entropy is $log_2(92^{11}) = 71.8$ bits and we learned $4\ log_2(16^5) = 80$ bits.
  The entropy is null.
  The password can be fully recovered.

Big-Endians prefer to break their egg from the bigger end. Not sure it is a good idea here. Breaking the 11-char password based on the knowledge of 4 partial hashes requires to generate all candidates and filter them based on these hashes. Which means... generating $3\ 996\ 373\ 778\ 857\ 415\ 671\ 808$ candidates.

From the smaller end, it seems slightly more doable as we would

- first generate a mere $5\ 132\ 188\ 731\ 375\ 616$ 8-char candidates;

- keep about one millionth of them (the ones matching the first partial hash);
- extend these candidates to 9 chars;
- keep about one millionth of them (the ones matching the second partial hash);
- extend these candidates to 10 chars, etc.

But is that practically feasible?

By the rules of PoC||GTFO (https://www.alchemistowl.org/pocorgtfo/), it is time for writing a Proof-of-Concept! Ok, this was also an excuse to delve into Hashcat...

Writing a Hashcat module and its kernels

Hashcat (https://hashcat.net/hashcat/) is *the world's fastest and most advanced password recovery utility, supporting five unique modes of attack for over 300 highly-optimized hashing algorithms.*

Let's build a Hashcat module that will

- take 4 partial hashes;
- assume they correspond to the API calls while typing the 8th, 9th, 10th and 11th chars of our password;
- crack the 11-char password.

A very interesting resource for writing a Hashcat module and its kernels (a term referring to the OpenCL code snippets being compiled for the GPU) is the Hashcat Plugin Development Guide[7].

We will use the SHA1 module and its kernels as a basis. There is no need to change any source file of Hashcat to develop a module, we just need to add a few files. We start by copying the SHA1 module `src/modules/module_00100.c` to `src/modules/module_90100.c` (90000-99999 is free for personal uses) and changing a few parameters: a new `HASH_NAME`, its reference number, called `KERN_TYPE`, we disable the self-test capabilities by adding `OPTS_TYPE_SELF_TEST_DISABLE` and we modify the expected hash structure from 40 to 32 bytes as follows.

`000aaaaa000bbbbb000ccccc000ddddd` with

- `aaaaa` the 5 top nibbles of the SHA1 of the first 8 chars of pwd
- `bbbbb` the 5 top nibbles of the SHA1 of the first 9 chars of pwd
- `ccccc` the 5 top nibbles of the SHA1 of the first 10 chars of pwd
- `ddddd` the 5 top nibbles of the SHA1 of the first 11 chars of pwd

So for the previous example `123456789AB` we get the "hash" `0007C222000F7C3B000BE4720004A3C4`. The alignment of each 5-nibble hash to 4 bytes is just to make our life easier when we will need to compare them with candidate

hashes. Actually, Hashcat does not transmit full hashes to the GPU kernels anyway, but 4 snippets of 4 bytes each. The SHA1 module is using snippets 3, 4, 2, 1 in that order (and never using snippet 0) but we will reorder them more simply as 0, 1, 2 and 3 by changing `DGST_POS0` to `DGST_POS3` constants.

That's it for the CPU side, we can compile with `make -j`.

The SHA1 module is using the *fast-hash mode* which means candidates are not all produced by the CPU and transmitted to the GPU, because the GPU is si fast it would be always waiting for the PCIe bus to get new candidates. Instead, *base* candidates are given to the GPU and the GPU generates a few derived passwords autonomously.

It features several kernel attack modes (i.e. how password candidates are generated) but we will only implement the *a3* mode, which is the brute-force mode. Each kernel implements several function versions: *S* and *M* for cracking *single* or *multiple* hashes. Indeed as there is no diversification seed, from one candidate you just compute its hash once and then you can compare it to as many hashes as you want at very little cost. The *M* version implements some smart bloom filter and binary tree search.

Moreover, each kernel comes in two implementations: a *pure* one, easy to develop and understand, and an *optimized* one, slightly more hardcore with some unrolled SHA1 code inline... The optimized kernels have some password size limitation but we do not really care as we will only break 8-char (extended to 11-char) passwords. One can call the optimized kernel by using the `-O` option of the Hashcat client.

So we will start our PoC development by copying `OpenCL/m00100_a3-pure.cl` to `OpenCL/m90100_a3-pure.cl` and removing the multi-hash *M* functions, to keep things simpler. We keep the SHA1 computation but when the time comes to compare a candidate hash to the target hash, we remove the comparison logic.

```
     sha1_final_vector (&ctx);
-    const u32x r0 = ctx.h[DGST_R0];
-    const u32x r1 = ctx.h[DGST_R1];
-    const u32x r2 = ctx.h[DGST_R2];
-    const u32x r3 = ctx.h[DGST_R3];
-    COMPARE_S_SIMD (r0, r1, r2, r3);
```

And we replace it by ours.

```
    if (ctx.h[0] >> 12 == search[0])            // shift to keep first 5 nibbles
and compare
    {
      for (u32 p9 = 20; p9 < 128; p9++) {       // guess 9th char:
        w[2] = p9 << 24;                        // append chr to current pwd
        sha1_ctx_vector_t ctx;                  // compute SHA1
        sha1_init_vector (&ctx);
        sha1_update_vector (&ctx, w, pw_len + 1);
        sha1_final_vector (&ctx);
        if (ctx.h[0] >> 12 == search[1])        // shift and compare with second
partial hash
        {
          for (u32 p10 = 20; p10 < 128; p10++) { // guess 10th char...
            w[2] = (p9 << 24) | (p10 << 16);
            sha1_ctx_vector_t ctx;
            sha1_init_vector (&ctx);
            sha1_update_vector (&ctx, w, pw_len + 2);
            sha1_final_vector (&ctx);
            if (ctx.h[0] >> 12 == search[2])
            {
              for (u32 p11 = 20; p11 < 128; p11++) {
                w[2] = (p9 << 24) | (p10 << 16) | (p11 << 8);
                sha1_ctx_vector_t ctx;
                sha1_init_vector (&ctx);
                sha1_update_vector (&ctx, w, pw_len + 3);
                sha1_final_vector (&ctx);
                if (ctx.h[0] >> 12 == search[3])
                {
                  // initial hashcat logic once a hash is cracked
                  const u32 final_hash_pos = DIGESTS_OFFSET_HOST + 0;
                  if (hc_atomic_inc (&hashes_shown[final_hash_pos]) == 0)
                  {
                    mark_hash (plains_buf, d_return_buf, SALT_POS_HOST, DIGESTS_C
NT, 0, final_hash_pos, gid, il_pos, 0, 0);
                  }
                }
              }
            }
          }
        }
      }
    }
```

Kernels are compiled on-the-fly when you use them, which is very convenient. But always remember to delete the compiled version from the cache when you are developing and modifying it! (`rm kernels/m90100_a3*`).

Once a match is found given a 8-char candidate, we recovered the first 11 chars of the password, but there is no mechanism to return a modified password from the GPU to the host. We could just print it with `printf` from the kernel itself but it does not cost much to recompute these 3 chars later.

Performances

On our laptop equipped with a RTX 2000, this first *pure* and *single-hash* version runs at 3440 MH/s (millions of hashes per second).

We can also create an *optimized* version based on `OpenCL/m00100_a3-optimized.cl` and use the same comparison logic as above. But to get it working, we also need to revert a couple of optimizations incompatible with our partial hashes:

- we remove the computation of `e_rev` and the early abort condition using it `if (MATCHES_NONE_VS (e, e_rev)) continue;`
- we remove a precomputation step which was done in the SHA1 module (subtracting a constant `SHA1M_A` from the first 4-byte hash segment to find) and we change our first test condition into `if (((a + SHA1M_A) >> 12) == search[0])` to compensate it.

Now this *optimized single-hash* version runs at 4066 MH/s.

Is it fast enough? In the worst-case scenario where we need to test all possible combinations out of a 92-symbol charset, the attack will take maximum 15 days. Knowing that Hashcat SHA1 runs at 5659 MH/s on a RTX 2000 and 50 GH/s on a RTX 4090, we can extrapolate that these 15 days can be squeezed into a mere 5 hours on a 8x RTX 4090 that you can rent for $4/h.

This is really an upper bound as we are not targeting passwords generated by a random generator but passwords chosen and typed in by a human. On another extreme, if we assume the first 8 chars of the password were just lowercase letters, the attack takes less than a minute on our laptop, *no matter how complex and long the full password might be*.

Demo

Let's simulate the typing of the password `iwashere$&@!2=[#)`.

```
#!/usr/bin/env python3
import hashlib
import sys

PASS = sys.argv[1]
assert len(PASS) >= 11
lh = [hashlib.sha1(PASS[:l].encode()).hexdigest()[:5] for l in range(8, len(PASS)
+ 1)]
print(' '.join(lh))
```

```
./passbolt_generate_partial_hashes.py 'iwashere$&@!2=[#)'
bb0c6 814f8 a9c98 9d4ef 35310 78a67 e7cfe a32d2 37735 0540a
```

Knowing the partial hashes, we can call our Hashcat module against the corresponding "hash" `000bb0c6000814f8000a9c980009d4ef`.

```
echo 000bb0c6000814f8000a9c980009d4ef > hash
time ./hashcat -m 90100 hash -a 3 --potfile-disable --quiet '?l?l?l?l?l?l?l?l'

000bb0c6000814f8000a9c980009d4ef:iwashere

real    0m6,226s
```

The first 8 chars were recovered in 6 seconds.

Breaking the next chars is straightforward as we just have to try all of the possibilities for each next char and check it against the next partial hash.

```
#!/usr/bin/env python3

import hashlib
import sys

PASS = sys.argv[1]
args = sys.argv[2:]

assert args.pop(0) == hashlib.sha1((PASS).encode()).hexdigest()[:5]
while args:
    H = args.pop(0)
    for i in range(20, 128):
        if hashlib.sha1((PASS + chr(i)).encode()).hexdigest()[:5] == H:
            PASS += chr(i)
            break
print(PASS)
```

```
time ./passbolt_recovery_step2.py 'iwashere' bb0c6 814f8 a9c98 9d4ef 35310 78a67
e7cfe a32d2 37735 0540a

iwashere$&@!2=[#)

real    0m0,033s
```

So the full password got recovered with an extra 33 milliseconds...

Variants

As we can only get all the partial hashes if the password is typed moderately slowly (3 char/s), we may occasionally miss a few partial hashes in other scenarios.

It is easy to design variants of the module tolerating some missing intermediate hashes. Missing the initial 8-char hash would be quite costly to overcome but missing n intermediate hashes would just cost e.g. 92^{n+1} instead of 92 trials, still quite affordable

compared to the massive brute-force of the initial 8-char hash. We will still need 3 or 4 partial hashes in total to make sure we can isolate a few or a single candidate.

Another type of variation to consider is to be able to catch cases when the user does not type linearly the password but edits it to insert or mutate some of the chars to make a better password in reaction to the breach notification returned by Passbolt. These cases are also pretty easy to take into consideration and will add only marginal complexity to the recovery process.

The problems documented here are quite specific to Passbolt's eagerness to assess the user password while being typed, but similar issues may arise as soon as related passwords are assessed in a small time frame. Actually, while documenting our findings, we stumbled upon a very interesting article by Jack Cable.[8] He details the related passwords problem which may occur when a user changes a compromised password or reuses a robust password with only some minor changes on several websites. The principle is that given e.g. N partial hashes of N related passwords, one may theoretically enumerate all candidates of each partial hash and look for candidates quite similar across the N sets. If you are interested in more math and probabilities, we warmly recommend you to read the article.

Jack Cable reported the potential problem to Troy Hunt and to the developers of two password managers: 1Password, Bitwarden. It is very unfortunate that Passbolt was not alerted of the risks at that time, especially given the fact that Passbolt chose a usage of Pwned Passwords that makes the attack highly practical. Not even mentioning that the current implementation of Passbolt doubles its queries to the API for no reason (as the most sagacious of you might have noted in the animated snapshot), a perfect signature for the server to spot Passbolt queries among its traffic.

Who are the potential attackers?

The only technical requirement is to be able to observe Passbolt plugin queries to the Pwned Password API.

Obviously, Troy Hunt and Cloudflare (which is massively caching Pwned Passwords queries in its nodes) are in such position. We are not questioning their good faith, just stating facts.

But also anyone deploying the service based on the downloadable hashes, e.g. within a corporate network.[9] Or simply anyone in position to intercept and observe the queries, such as a TLS proxy. In these two examples, we assume a corporate certificate authority has been deployed on the laptops.

While we only implemented the *simple* hash functions in our Hashcat PoC, an attacker collecting many partial hashes would add the *multiple* hashes functions. Remember that by construction, there is no seed and each computed hash can be compared to many intercepted hashes at once. At scale it may even be interesting to build dedicated rainbow tables.

1Password and Bitwarden

1Password uses Pwned Passwords as well, but with the following differences:

- opt-in: it needs to be activated in the application settings, cf. *Privacy ⇒ Check for vulnerable passwords*;
- the API is called only when saving the password, so once it has been typed entirely by the user.

Bitwarden usage of Pwned Passwords is as follows.

- opt-out while creating or changing the master password: it presents an already selected checkbox *"Check known data breaches for this password"*;
- opt-in for every single account password: the user needs to press a button to *check known data breaches for this password*.

Alternative protocols

Inspired by Pwned Passwords, Google and Stanford published in 2019 another protocol mixing k-anonymity and private set intersection methods.[10] The protocol has been implemented in the Google Password Checkup.[11] Theoretically it could potentially be misused as well: a partial 2-byte hash is sent to the server. Imagine a client implementing the same logic as Passbolt: after typing 12 chars of a strong password (entropy of $log_2(92^{12}) = 78.3$ bits), it could be recovered from 5 intercepted 2-byte hashes ($5\ log_2(256^2) = 80$ bits). But two factors temper this risk: here the hash is computed over a concatenation of the username and the password, and the hash algorithm is Argon2, which is intentionally slow to compute. So an actual brute-force would be way less practical. Note that the 2019 paper hinted at a plan to migrate to a zero-password leakage variant of the protocol, avoiding completely the k-anonymity leakage. It is unclear if it has been done as of today.[12]

Let's also mention the paper of Cloudflare and Cornell University, which is exploring more advanced protocols (FSB and IDB) to further reduce information leakage in the APIs, and the subsequent paper of Cloudflare, Cornell University and University of Wisconsin-Madison, aiming at proactively warning users if *similar* passwords have been leaked and

could jeopardize a user password by credential tweaking attack.[13,14] Their MIGP protocol (*Might I Get Pwned*, a nod to Troy's *Have I Been Pwned*) is extensively described on Cloudflare blog.[15]

Conclusion

Pwned Passwords is an interesting example of the dangers of designing a cryptographic protocol with a deemed acceptable risk, being used in an unforeseen way such that the risk becomes suddenly unacceptable.

We demonstrated that for an entity able to spy on Pwned Passwords API usage by Passbolt while a user types in a new password of 11 chars or more, it is surprisingly doable to recover the password.

That said, such a service, when used properly, is still very valuable for the password managers users to get alerted of compromised passwords.

Passbolt chose to fix the issue in the browser extensions v4.6.2 by querying the Pwned Passwords API

- only when saving a new or updated password;
- and only if the password estimated entropy is larger than 60 bits (which is about 10 chars if using mixed symbols, or 20 chars if only using numbers, for example).

At the time Jack Cable alerted Troy Hunt about some risk of information leakage via queries about related passwords, adding warnings to the service documentation was discarded as part of an assumed trade-off.[8]

Let us hope that this time the risk is tangible enough for Troy Hunt to document bad usages and associated risks of his API.

Update - 2023-04-20

After the initial publication of this blog post on April 17, 2024, we notified Troy Hunt, and as a result, the Pwned Passwords APIv3 documentation has been updated to include a warning about the risks of incremental searches.[16]

Acknowledgements

Communication with Passbolt was quite a model of vendor reactivity and openness. We thank them for their prompt reaction to all our inquiries. They provided clear communication throughout the process, and actively collaborated to ensure a smooth resolution.

We also thank Troy for his quick update of the API documentation!

And finally, thanks to our colleagues at Quarkslab for proofreading this article!

Disclosure timeline

- 2024/03/22 - Vulnerability reported to Passbolt
- 2024/03/23 - Acknowledgement of the vulnerability
- 2024/03/27 - Informal and open discussion *IRL* with Passbolt CEO at InCyber 2024, Lille. Thanks for the hoodie ;)
- 2024/03/28 - Passbolt communicated their action plan, reaching out for our feedback on their remediation strategy
- 2024/03/28 - Release of passbolt_styleguide v4.6.3 (https://github.com/passbolt/passbolt_browser_extension/releases/tag/v4.6.3) implementing the actual fixes
- 2024/03/29 - Release of passbolt-browser-extension v4.6.2 (https://github.com/passbolt/passbolt_browser_extension/releases/tag/v4.6.2) mentioning the 15 implemented fixes in the changelog
- 2024/03/30 - Chrome extension v4.6.2 published
- 2024/04/03 - Firefox extension v4.6.2 published
- 2024/04/04 - Edge extension v4.6.2 published
- 2024/04/04 - Quarkslab sends some comments on the proposed strategy, calling for opt-out option and clearer messages to the user when the password is evaluated as very weak (< 60 bits) but not sent to Pwned Passwords API
- 2024/04/04 - Passbolt indicates opt-out is available to admins on the Pro/Cloud versions and will be extended to the Community Edition as well. User message will be made clearer as well.
- 2024/04/11 - Windows application v1.0 pushed
- 2024/04/11 - Release of passbolt_api v4.6.2 (https://github.com/passbolt/passbolt_api/releases/tag/v4.6.2), *This version is a targeted security release of both the API and the browser extension focusing on fixing security issues reported by security researchers.*
- 2024/04/16 - CVE requested by Passbolt
- 2024/04/17 - Synchronized publication of this post and the Passbolt Incident Report (https://help.passbolt.com/incidents/pwned-password-service-information-leak)
- 2024/04/17 - Quarkslab brings this work to the attention of Troy Hunt and suggests a warning about such type of misuse in the APIv3 documentation
- 2024/04/20 - APIv3 documentation has been updated to include such warning, and this blog post has been updated accordingly [16]

---

1. Introducing 306 Million Freely Downloadable Pwned Passwords (https://www.troyhunt.com/introducing-306-million-freely-downloadable-pwned-passwords/) ↩
2. Validating Leaked Passwords with k-Anonymity (https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/) ↩
3. I've Just Launched "Pwned Passwords" V2 With Half a Billion Passwords for Download (https://www.troyhunt.com/ive-just-launched-pwned-passwords-version-2/) ↩
4. API v2 - Pwned Passwords overview - Searching by range (https://haveibeenpwned.com/API/v2?ref=troyhunt.com#PwnedPasswords) ↩
5. API v3 - Pwned Passwords overview - Searching by range (https://haveibeenpwned.com/API/v2?ref=troyhunt.com#PwnedPasswords) ↩
6. NIST Special Publication 800-63B - Digital Identity Guidelines - Authentication and Lifecycle Management (https://pages.nist.gov/800-63-3/sp800-63b.html) ↩
7. Hashcat Plugin Development Guide (https://github.com/hashcat/hashcat/blob/master/docs/hashcat-plugin-development-guide.md) ↩
8. Attacks on applications of k-anonymity for password retrieval (https://cablej.io/blog/k-anonymity/) ↩↩
9. How to create an offline self-hosted haveibeenpwned API service (https://community.passbolt.com/t/how-to-create-an-offline-self-hosted-haveibeenpwned-api-service/5541) ↩
10. Protecting accounts from credential stuffing with password breach alerting (https://www.usenix.org/system/files/sec19-thomas.pdf) ↩
11. Protect your accounts from data breaches with Password Checkup (https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html) ↩
12. Why password managers are your safety net during a data breach (https://blog.google/technology/safety-security/why-password-managers-are-your-safety-net-during-a-data-breach/) ↩
13. Protocols for Checking Compromised Credentials (https://arxiv.org/pdf/1905.13737.pdf) ↩
14. Might I Get Pwned: A Second Generation Compromised Credential Checking Service (https://www.usenix.org/system/files/sec22-pal.pdf) ↩

15. Privacy-Preserving Compromised Credential Checking (https://blog.cloudflare.com/privacy-preserving-compromised-credential-checking/) ↩
16. API v3 - Incremental Searching (https://haveibeenpwned.com/API/v3#PwnedPasswordsIncrementalSearching) ↩↩

If you would like to learn more about our security audits and explore how we can help you, get in touch with us (https://content.quarkslab.com/talk-to-our-experts-blog)!

↑ Back to top