KEEPASS, KEEPASSRPC, VULNERABILITY

Exploiting KeePassRPC

Aug 01, 2020 Philipp Danzinger

While taking a university course on security, I discovered two critical related vulnerabilities in `KeePassRPC`, an addon for the popular password manager `KeePass`. Both vulnerabilities allow a malicious web site to read and leak (unlocked) KeePass databases, while being very hard or impossible to detect, provided the KeePassRPC addon is installed. Shortly after I reported them, the vulnerabilities were patched and publicly disclosed by the developer.

With this blog post I aim to provide a bit of context and background about the discovery, as well as some technical details.

First, a short summary for users:

- **Am I affected?** You are affected if you are using the password manager `KeePass` and have installed the plugin `KeePassRPC` (prior to version `1.12.0` released on the 29th of July 2020). The `KeePassRPC` plugin is used to interface with the Firefox/Chrome extension `Kee` as well as the Thunderbird extension `KeeBird`.
- In case this applies to you, you are advised to update `KeePassRPC` immediately.
- **Have my passwords been compromised?** If you are or were using a vulnerable version of `KeePassRPC`, this is possible. Exploiting the vulnerabilities is possible while leaving little or no trace. On the other hand, the vulnerabilities were not publicly known prior to the release of the patch, and at the time of writing this, I am not personally aware of any real-world exploit using them. Based on this, one may consider it unlikely that individual users were compromised. In the end, you have to weigh the risks yourself.
- For more details, see the official announcement.

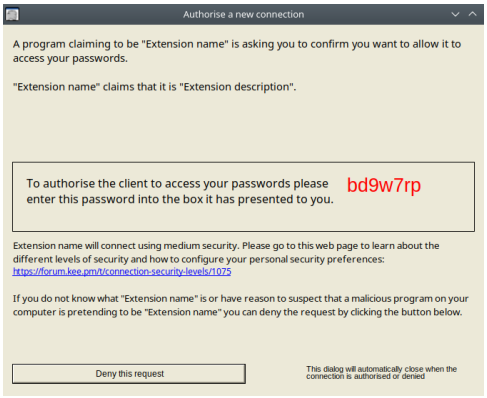
Without further ado, let's jump into the details.

I had one job…

This tale starts with the finale of another one. Internet Security, one of the ~~hardest~~ most rewarding computer science lectures at TU Wien, was about to end. A big part of the lecture consisted of solving practice problems in offensive security (which is code for hacking).

One part of the final challenge was a simple implementation task: the lecturers had set up an automated environment with KeePass, KeePassRPC, and a web browser. The assignment was to adapt the official Kee extension to leak passwords once a connection had been established (or to create a new browser extension mimicking Kee).

KeePassRPC works by creating a web socket server, which Kee can connect to. To secure the connection, a protocol called `SRP-6a` is employed. This involves a popup window by KeePassRPC containing a password, which the user has to enter into the browser. This proves to KeePassRPC that the connection is authorized by the user.



In our assignment, a (virtual) user would automatically enter this password. Our only job was to change the extension so that it steals some passwords afterward.

...but...

I didn't know this at first. The assignment had just been published and the grader wasn't quite ready yet.

So, after essentially solving the challenge (and being unable to check) I thought "wouldn't it be interesting if I could do it without a password?".

It turned out that bypassing the password was very much possible. Only later did I find out (to my surprise and shock) that this could be abused remotely since web browsers don't prohibit web sites from opening a web socket connection to localhost.

Once all the dust from the lecture had settled, I consulted with the lecturers, Michael Pucher and Georg Merzdovnik, to develop proof-of-concept exploits and responsibly disclose the vulnerabilities.

Kee, KeePassRPC and SRP-6a

KeePassRPC and the Kee browser addon use (a slight variation of) the SRP-6a protocol to establish an encrypted connection. For the purposes of this post, SRP-6a can be thought of as an extension of the well-known Diffie-Hellman key exchange.

Diffie-Hellman is a protocol that allows two parties to establish a shared secret key even in the presence of an eavesdropping attacker. To that end, they each pick a random secret number (say, `a` and `b` respectively) and perform a modular power calculation (i.e. a power modulo `N`, where `N` is a fixed large number) to arrive at public numbers (`A` and `B`), which they exchange. By means of mathematical magic, each party then uses its own secret number and the other person's public number to arrive at the same shared key, called `S`.

SRP-6a can be thought of as adding an authentication layer to Diffie-Hellman. First off, the two parties receive asymmetric roles of *user* ('client') and *host* ('server'). New calculations and parameters are introduced to ensure that a connection can only be established if both parties are in possession of some password and if they agree on said password.

More details can be found in the official SRP-6(a) documentation and this post about the SRP-6a implementation in KeePassRPC.

Vulnerability 1: A=0 edge case

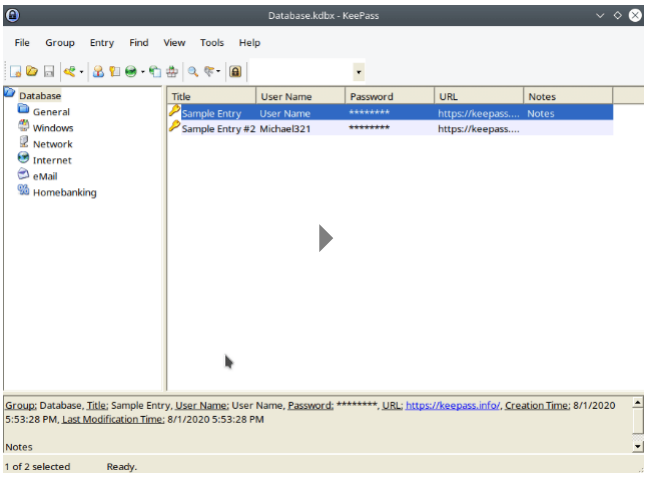
In SRP-6a the *user* has little freedom. In fact, apart from a `username` (which is not used by KeePassRPC) the only parameter the *user* can freely control is `A`. According to the protocol, `A` gets calculated as `A = g^a` where g is a conventional constant number, and `a` is a secret number picked by the *user*.

Of course, an attacker need not follow the protocol to the letter and can choose `A` freely. One choice turns out to be particularly interesting. When `A=0`, the computation of the session key `S` becomes particularly simple. When we (somewhat counterintuitively) start with the calculation usually performed

by the host, we obtain `S = ((Av)^u)^b = (0^u)^b = 0`. Yup, the session key just becomes `0`, regardless of the password.

This is not an issue with SRP-6a itself, as the specification [1] makes the following demand: `The host will abort if it detects that A == 0 (mod N)`. Unfortunately, this check was not correctly implemented in KeePassRPC, allowing an attacker to connect instantly without the password.

The following clip shows the exploit in action. If you pay close attention, you can see the password popup from KeePassRPC becoming visible for a short time before quickly vanishing again.



Vulnerability 2: weak secret random numbers

After figuring out the `A=0` exploit, I decided to revisit another peculiarity I had noticed previously. Namely, a few variables, including the secret *host* parameter `b`, were being generated with a cryptographically weak random number generator.

```
// Language: C#

// SRP.cs before fix (https://github.com/kee-org/keepassrpc/blob/67ba4de94bbd81368a37e911
// Comments are mine

class SRP
{
  // [...]

  internal void Setup()
  {
      _b = new BigInteger();
      _b.genRandomBits(256, new Random((int)DateTime.Now.Ticks));
      //                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

      _B = (_k * _v) + (_g.modPow(_b, _N));
      while (_B % _N == 0)
      {
          _b.genRandomBits(256, new Random((int)DateTime.Now.Ticks));
          //                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          _B = (_k * _v) + (_g.modPow(_b, _N));
      }
      _Bstr = _B.ToString(16);
  }

  // [...]

}
```

To be exact, the random bits for `b` (or '`_b`') were taken from a `Random` instance seeded with `(int)DateTime.Now.Ticks`.

According to the official C# documentation, the `Ticks` attribute of a `DateTime` object contains the number of 100-nanosecond intervals (*okay...*) that have elapsed since 0001-01-01 midnight (*seriously?*).

Also, if used with `DateTime.Now`, this means 0001-01-01 midnight in local time (*I am out of sarcastic remarks*).

The point is, the server's secret is based on local time, and it is ticking at a rate of 10_000 intervals per millisecond. If guessed correctly, this server secret `b` can be used to calculate the session key `K`. Referencing the KeePassRPC technical documentation, this works by first reversing step 2 to determine `v`: `B = kv+g^b => v = (B - g^b)/k`. Then, an attacker can compute the session key `S` as the host would in step 7: `S = (Av^u)^b`.

Guessing this value may seem like a daunting task, but there are a few factors at play that make it possible.

- Firstly, KeePassRPC allows for infinite retries. With the processing time for KeePassRPC and local web socket latencies, a few hundred tries per second are possible.
- Secondly, KeePassRPC also generates a 'seed' parameter `s` shortly before creating `b`. `s` created in the exact same way as `b` and is known to the *user*. Therefore, an attacker can arrive at a reasonable estimate for the tick used for `b` by 'locally' finding the tick for `s` first. This eliminates many web socket round trips and hash calculations associated with the protocol.
- Since the timing when generating `s` and `b` is usually around `1-3` milliseconds apart, this leaves a few ten-thousand tries for `b`.

All in all, my exploit for this vulnerability usually took around 1 minute on a modern desktop PC. The time likely scales strongly with (single-core) CPU speed. In fact, the relationship seems to be quadratic, because slower hardware increases the number of ticks to try as well as the time required to try each one.

On a final note: with the first vulnerability present, this one is far less relevant for practical purposes. The main point was to show that this aspect of the implementation was vulnerable as well and to make a few interesting observations along the way.

Disclosure and Patch

I disclosed the vulnerability to the vendor in private on July 28th, 2020. The report was acknowledged within 12 hours. Within 24 hours of my initial report, the vendor issued a patch for KeePassRPC (with version 1.12.0 and later 1.12.1 to fix a compatibility problem) and publicly disclosed the vulnerabilities.

From my understanding, the patch fixes the first vulnerability by throwing an error if `A=0`. The second one is fixed by using a cryptographic number generator instead of a generator seeded by the system time.

Furthermore, an additional security layer is introduced that should prevent web sites from connecting to KeePassRPC in the first place. This is done by validating the origin of incoming connections against a whitelist of different origins used for browser extensions (for instance `chrome-extension://`).

Timeline

- **2020-07-28 22:00 UTC** - I **disclose** the vulnerabilities to the vendor
- **2020-07-29 10:00 UTC** - Vendor **acknowledges** the vulnerabilities and lays out a plan to fix and publicly disclose them
- **2020-07-29 17:00 UTC** - KeePassRPC 1.12.0 is released to **fix** the vulnerabilities
- **2020-07-29 21:00 UTC** - Vulnerabilities are **publicly disclosed** by the vendor
- **2020-08-01 22:00 UTC** - **This blog post** is released
- **2020-08-02 11:00 UTC** - Blog post updated to add details about the disclosure and patch

`updated_at 02-08-2020`