A Poor Programmer's Password Manager

Michael Scott

Technical Innovation Institute
michael.scott@tii.ae
May 10, 2023

Abstract. You are a programmer that uses the Internet a lot. You don't want to be one of those unfortunates that gets their favourite password hacked, due to somebody's failure to properly protect the password file on a remote server. You don't want to change the world, you just want to work securely with the world as it is. You haven't the patience for elaborate procedures. You don't want it to cost you. How can you live with the broken Username/Password system, and yet feel certain that while the rest of the world may be vulnerable, you will be OK?

1 Introduction

First let us quickly recall how Username/Password works. The only cryptographic tool required is the humble one-way collision-free hash function, which can convert a string of any length into a unique fixed length (typically 256-bit) fingerprint. So as a function it works like $y = H(x)$ where x is a string, and y is the fingerprint. Inputs can be concatenated, so $y = H(a, b)$ just means that the string b is appended to the end of a to provide the input x. The one-wayness means that given just y, its impossible to determine x. The collision free property means thats it is computationally infeasible to find two x values which generate the same y.

Once you register as a client, the server generates a random "salt" s and stores $s, H(p, s)$ against your username, where p is your password. In practise various schemes have been proposed using different hash functions, and many incorporate iterative application of a fast hash function in order to artificially slow it down. All this information for every client is stored in a password file. When you enter your username and password, the server looks up your salt, calculates the hashed value, and compares it with the stored value. If its good, you are in. Which is all very fine, but then a hacker steals the server's password file, and using a dictionary of common passwords, tries every one until they start getting hits, even if it takes them a substantial effort due to that artificial delay. Your only protection is to make sure your password which is not in that dictionary. Hence those tedious password policies requiring you to remember an awful looking password for every service you use, and which you really need to write down somewhere.

2   A modest proposal

The proposed solution described here falls partially into the category of a "state-less" password manager. Passwords are not stored anywhere, they are generated afresh every time they are needed. It is not a new idea, see for example[1] and[2], and it has its rather strident critics - [3][4][5].

In our case the password manager is not rigidly stateless - it does support a database of non-secret meta-data associated with each password. For each service it stores and displays the associated website, the service name, the username, the policy index and a free-form field for any notes the client wishes to associate with the service. This information is hard-coded into the application, and, yes, if a new service needs to be added the application needs to be recompiled. The application is in fact completely self-contained and has no external dependencies.

But as regards the passwords themselves our password manager is stateless, as no passwords are stored. Its only fair to recall the objections to stateless password managers.

1. The password generated may not meet the website's password policy
2. If the master password is compromised, all passwords are compromised, and each must be laboriously updated.
3. It is difficult to update an individual password, if that should be required.
4. If some existing password cannot be changed, then the password manager generated password cannot be used.
5. If one password is compromised, a brute force search will reveal the master password.
6. If the password manager needs a bug-fix, then all passwords may need to be re-generated.
7. With a traditional encrypted password manager, we need two things to obtain site-specific passwords: the ciphertext of the password vault, and the master password. With a stateless scheme, as an attacker we only need one: the master password.
8. What if the website URL changes?

The twist proposed here is to use our programming skills to individualise our password manager, thus sidestepping most of these known shortcomings of classic stateless password managers. We will return later to respond to these objections.

----

[1] https://github.com/lesspass/lesspass

[2] https://vks.ai/ppm

[3] https://security.stackexchange.com/questions/214301/
what-are-the-cons-of-stateless-password-generators

[4] https://kichik.com/2020/06/19/stateless-password-manager-usability/

[5] https://www.sjoerdlangkemper.nl/2018/01/17/problems-with-pwdhash/

3   Attacks and Defenses

First let us step back a bit and look at the issue from the hacker point of view. For a hacker the aim is to maximize the bang for the buck. In other words with the minimum amount of time and effort they would like to capture as many individual credentials as possible. Attacks can be classified as scalable or non-scalable. A scalable attack captures the credentials of multiple individuals with just one successful attack, and might be launched against the server as described above. A non-scalable attack on the other hand requires an individualized effort again each individual client.

One major weakness of the Username/Password mechanism is that the Username and the Password are transmitted directly to the server, albeit under cover of the TLS/SSL protocol. But this does imply that at both ends of the link there are potential points of attack, maybe by a virus, or by an employee on the server side. But a more likely possibility is that you have been drawn to a fake server that looks just like the real one. In that case when you entered your password, you just gave it away to be bad guys. In short you were the victim of a phishing attack. To avoid such attacks a much better idea would be a zero-knowledge phishing-immune challenge-response based protocol in which the client secret is never transmitted from client to server, but rather used to construct a specific response to a random server challenge. Like MIRACL's M-Pin product. But that's another story.

For this exercise we will simply try and improve Username/Password as much as we can. Our plan is to avoid scalable attacks (a) on the server side by using very strong passwords, and (b) on the client side by deploying an individualized app, requiring a specific and targeted non-scalable attack to hack. We believe that just by forcing a hacker to go after individuals, rather than harvesting millions of passwords in a single attack, that this is enough to improve the user's security by many degrees of magnitude.

4   Your Password Manager

Consider now a specially tailored password generator application. Stored in the source code somewhere is a random string, created by the programmer getting their 3 year old grandson to bang on the keyboard

```
char r[]="dfgneNVriothjOerjgorigjq[89xu8qu1j)(kdUjSSgj4[jjjr";
```

The program consists of a drop-down list with the names of the services with which the programmer has an account. There is also a box into which they enter their one-and-only Master Password p when the app first starts up, and another box which will require a short 4-digit PIN number every time a password is needed. When the name of a service n is selected from the drop-down list, the associated website URL and the Username for the service are displayed. The Username is copied into the clipboard and can be pasted from there into the

website username field. The URL displayed is a safe place from which you should ideally navigate to the log-in page. If the URL has been entered to the browser from another source, at least check that it is consistent with the one displayed here – otherwise you could be subject to a phishing/man-in-the-middle attack.

Finally on entering the PIN a suitable password for the selected service is calculated as $pw_n = H(r, H(PIN, H(n, H(p))))$, and again copied to the device's clipboard, for pasting into the password field on the website. Observe that the password is generated from a hash of the Master Password, the PIN, the name of the service (like "GitHub"), and the random string. Note that by default the entered password pw_n is never visible, and so probably remains unknown and unremembered by the user (and to any shoulder-surfers). For the hash function $H()$ we use the standard SHA3-512 function.

We are assuming of course that the average user has no problem remembering one reasonably secure password, as required by all password managers, and a PIN number.

The binary of the app can be copied to other computers and devices that the programmer owns. Its source code should be kept safely offline.

5   Policy

To create the passwords we basically convert the output of the final hash to base64 encoding, which gives us a nice mix of upper and lower case letters and digits, and occasional special symbols as dictated by most password policies. Our default policy checks the output to ensure it contains at least one uppercase letter, one lowercase letter, one digit and one special character. If it doesn't we hash it again until it does. Another decision is the length of the password. We decided on 12 characters, giving roughly 72-bits of entropy. So a typical password generated by this program looks like `hSU!pKA09v6Z`. Normally the password is just silently dropped into the device's clipboard, but it can be made visible if desired, for example if it needs to be manually copied to an unsupported device.

One of the main criticisms of stateless password managers, is that the password generated may not fit with the policy of a particular website. Such policies are notorious for their variety. For example a website might set a maximum password length of just 10 characters, or it may (or may not) demand the inclusion of special characters. In practice our default password policy will work in most cases, but on occasion it may have to be tweaked. But for a programmer the solution is quite simple. There is a "policy" function which returns true or false. For every new website, a new policy may be required. However in our experience three or four such policy functions suffice, and a new policy is rarely required. Next to each website in the code is stored an index which identifies the appropriate policy.

6 Security

One of the biggest disincentives for a hacker is to have to go after each service user individually, and hack into their individual device. That is a major hassle, and is probably outside of their skill set. And one component of the password is the random r tucked inside the binary of your very own app, a copy of which is not available to anyone else on the planet. And you can get creative with r, and where it is stored. Although the original source code might be open source, as it is here, as a programmer you can modify it. For example you can spread pieces of r all over the program, and recombine them programmatically. Just do enough to make your binary truly unique.

Another idea might be to muck with the SHA3 hash function. Increase the number of rounds. Tweak some of its constants (admittedly a bit risky). Its your password manager. Own it.

Ideally your one-and-only Master Password should not be in any dictionary, certainly when combined with the PIN. One attack is to steal the password file from a server, then access your computer and try password and PIN guesses until there is a hit on the password file entry. So as for standard password hashing we make it more difficult by aggressively iterating the application of the hash function, and calculate the password as $pw_n = H(r, H(PIN, H(n, H(H(H(...H(p)...))))))$, with p hashed maybe tens of thousands of times, so that it takes milliseconds to calculate. Indeed since you only enter it once during the working day, you might afford to wait even a second or two.

Of course if your app binary is stolen it can be reverse engineered and r extracted. But first they got to get your app binary, and perform a specially tailored attack on it, because there is none other quite like it. If some-one sits down at your computer while the app is active, they cannot generate your passwords, as they don't know the PIN. Note that nothing related to the Master Password or the PIN is stored inside of the app, so reverse engineering your app doesn't compromise either. Importantly, if the wrong Master Password and/or PIN is entered, no warning is given. The passwords generated will simply be the wrong ones. Also the implementation is completely deterministic – no cryptographically secure random number generator is required. Which is a good thing as they are notoriously hard to implement properly.

There is I admit a case for making the PIN 6 digits rather than 4, as PIN guesses can be spread across multiple sites. You decide.

A potentially serious issue is how to replace a compromised password. Again we simply program our way around the problem. For example replace the service handle "GitHub" with "GitHub 2", which is sufficient to generate a new unrelated password. The same idea can be used if you are a victim of one of those horrible password policies which require you to change your password every month.

7   Implementation

OK, so lets do it. To make life easier it makes sense to use a tool like Qt^6, with its nice Qt Creator tool. Now the same source code can be used to generate the app across a range of devices. Pay attention to the open source licensing arrangements, although if I read it right, and since its for personal use, its free.

If you need access to a new service you need to add it to the list of services embedded in the app, recompile it, and download it again to all your devices. And of course you have the one-off but tedious task of updating all of your existing poorly chosen passwords. There is a price to be paid for taking control of your own security.

It is particularly important to note the place in the code where you (or your grandson) enters that long unique random string. Don't forget to change it!

When the app is running it looks as shown below on all devices, just after the master password has been entered. Our user interface is extremely simple, and lacks any artistic merit. You can probably do a lot better.

We have implemented the password manager on all the devices available to us – a windows PC, an Apple Mini Mac, an iPad and several iPhones of varying vintage. Implementation on the Apple devices, which requires the Xcode tool, was not entirely straightforward. One issue that arose was the necessity to pay for an Apple Developer license ($99 per year) so that we could load the application onto our iPhones and iPads. We were frankly disappointed to have to pay anything for this project – but maybe you already hold such a license. The rendering on the devices was also not perfect – while it should look OK in portrait mode, it was often not usable in landscape. Maybe we just need more experience with Qt. As future work, given the simplicity of the program, we intend to implement it natively in Xcode using the Swift language.

8   Operation

Operation is as simple and as user-friendly as we could make it. On first running after the device is powered up, enter the Master password, and press enter. Thereafter select a service, paste the username, enter the PIN, and paste the password.

8.1   Co-existing with the browser

Browsers go to great lengths to make password access as painless as possible (for an example see the appendix). Typically they all support their own in-built password managers. One big idea is to allow "syncing" of passwords across devices, using the browser's own cloud resource. Syncing also means that if (when?) you lose your device, a replacement device can have its passwords restored from the cloud. But of course each different browser is going to do its own thing, each

---

[6] `www.qt.io`

maintaining its own independent cloud resource. Third party password managers provide a similar browser-independent functionality, but since you are renting space in the cloud, they are going to charge you for that. Its their basic business model. Often they also store unencrypted meta-data associated with your accounts, like maybe even your browsing history.
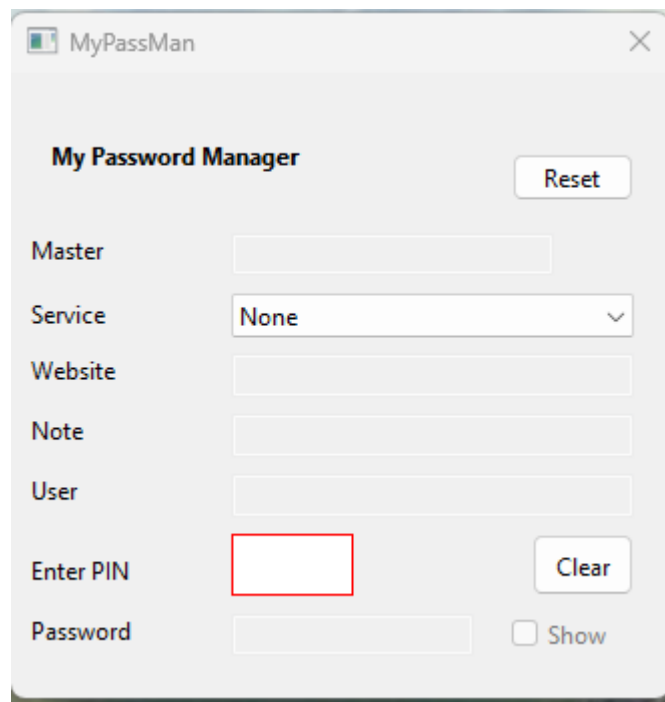
But this does mean that your passwords (and any associated meta-data) get stored up in the cloud. And we don't want that. We are going to "sync" the old-fashioned way, by installing our own individual password manager on all of our devices.

Having done that, you may now want to completely disable those inbuilt browser password managers. Loss of a device is still not critical, as the password manager can be restored if necessary from your source code. Or you may decide to continue to trust your browser, in which case the passwords generated only need to be entered once, and thereafter the browser will remember them for you. A sensible compromise, that keeps most of the nice advantages of browser syncing, would be to insist that the browser keeps passwords locally and does not propagate them to any cloud, by tuning the sync settings such that everything *except* passwords continues to be synced across devices.

9   Objection responses

Finally we give our response to the objections listed above.

1. Simply program in a new policy
2. Compromise of a password manager's master password will always be bad news. But you can make password guesses prohibitively expensive. However there is no ducking the fact that you do have a responsibility to come up with one good strong memorised password.
3. Simply add a counter to the website's handle and thus create a brand new utterly different password. Its a hassle, but any website that insists on rotating passwords is always going to be a hassle to use.
4. Such passwords should be handled externally to this password manager. Its a corner case we don't need to support.
5. Again make master password guesses expensive. And the attacker also needs to guess the PIN
6. True. But this password manager is very simple, using just one cryptographic function. The code size is only about 300 lines (excluding the standard SHA3 code). But do think long and hard before deciding on and finalizing your individualization code tweaks.
7. Now an attacker needs 3 things, the app itself, the master password, and the PIN.
8. Not a problem as the URL is not included in the hash, only the website handle that you choose.

Fig. 1. The Password Manager app.

10   Conclusion

Having proposed this idea, there is really no excuse for me not to implement it for my own use. This I have done. Gradually I have migrated more services, requiring only a new entry into the app's internal database, and the occasional implementation of support for a new policy. I continue to allow browser syncing of data across my devices, but I have disabled password syncing. So far the experience has not been too painful, and my own previously rather chaotic and stress inducing approach to password managment is in the process of being being phased out. I have reason to be confident that my interaction with internet services is now just a private arrangement between myself and the service, with the browser's only role being to establish a secure TLS-backed link that authenticates the service to me and encrypts our transaction.

Accessing passwords from the browser

Let's take Google Chrome as an example. Your passwords are stored both locally and in the cloud. To access what is stored in the cloud enter `passwords.google.com` into the address bar. To see your passwords you will need to *authenticate yourself to your browser*. To access what is stored locally go to settings and select "Autofill and Passwords", then "Password Manager". This time to see your passwords you will need to *authenticate yourself to your device*. Note that different passwords or biometrics are typically needed to authenticate for each of these scenarios.