Secure Login without Passwords

T.M.C. Ruiter

December 22, 2013

Any person can invent a security system so clever that he or she can't imagine a way of breaking it.[1]

Abstract

These are the ingredients for secure logins:

- For login purposes, the webserver uses a separate login server. This server can be internal (in the back-office) or external (somewhere on the Internet).
- The login server maintains a small array with Secret keys in an HSM, which will be renewed regularly (oldest key removed, all keys shifted one position, new key added).
- All cryptographic operations for the login procedure are performed by the login server and take place inside the HSM; no values are remembered afterwards, except Secret keys.
- For each user, a random number acts as the site key for that user. The user gets a different value, which is the $\texttt{AES}_{256}$ encryption of the site key for that user with the newest Secret key.
- Logging in is a process of proving that both the website and the user have the right key by sending $\texttt{SHA}_{256}$ hashes of random numbers encrypted with these keys. Neither the site key nor the user key are ever sent over the line.
- When Secret keys are changed, the user automatically gets a new key. This way, user keys are changed regularly as well.
- All user keys are put on a keyring, which is a set of at least 99 keys, most of them dummies. No other information whatsoever is stored in a keyring, which may be stored in an encrypted file.
- The user selects which key is used for what, which it needs to write down or remember. Although keys themselves are changed regularly, the key number and its purpose will never change during the lifetime of the keyring, which might be forever. This makes remembering key numbers, at least for those keys that are used regularly, doable for most people.
- The keyring itself is something you must have; the key number something you must know, so logging in using a keyring is a basic form of two-factor authentication.

[1] Also known as Schneier's Law.

Contents

1 Introduction

Passwords should be kept in memory—human memory, that is—at all times. This article is about passwords for websites; the passwords many people use on a day to day basis. Passwords are an archaic type of security measure ([2]), compared to the scheme proposed here.

1.1 On human behavior

The rationale amongst security experts is that passwords should not be written down. Ever. And passwords should be unique for each and every website. Oh, and—I almost forgot—you have to change them as well. Each month or so will do nicely.

Yeah, right! I cannot remember all different passwords I am forced to use, although my memory is quite good. I *have* to write them down, otherwise, I am lost. (Fortunately, I have some support for this [3].) I don't trust software that will "remember" my passwords for me, because their "memory banks" might be on a malicious server on the other side of the ocean. Therefore, I resort to a little black booklet, with all my account information. I don't remember passwords any more, I remember where my booklet is. And I confess that I am compelled to reuse passwords, and to keep the ones I am not forced to change, so I can actually remember some of them. So I believe nothing has fundamentally changed in more than 13 years... [1]

I have given up on inventing a scheme for passwords that differ from each other, can be changed individually at different times, and are easy to remember, as not to be forced to write them down. I believe no such scheme exists, because websites have different requirements regarding passwords. Some don't allow spaces, some complain about length, some fuss about dictionary lookup. The interval at which you are forced to change passwords is different for websites; some changes are compulsory, some voluntarily, some just to stop annoying pop-ups. To see what I mean, watch [4].

For the user, the bad thing with passwords is that you have to keep track of it all. Remembering difficult passwords is cumbersome for most, and impossible for some. Tracking things infalliably, and remembering different passwords for each and every site is not something people excel at.

1.2 The keyring system

Using 128 random bits as a key to gain access to a website is far more secure than letting people decide which password they would like to use to do so.

Imagine a keyring, not unlike your own keyring on which you have keys for your car, your house, shed, or locker. This keyring has a label and 99 keys on it, numbered 1 through 99. Instead of brass or steel, these keys are made of 128 random bits each. The label is another 128 bits long, and as random as possible, like the keys. Instead of being on a steel ring, these keys, with the keyring label, are written to a file on disk; a blob of 100 strings of 128 bits.

The use of keys on this keyring is not entirely different from using real physical keys. The bits in a key are comparable with the tooths or holes of a physical key you use to unlock your home. You do not need to remember exactly how far the tooths need to protrude or exactly where and how deep the holes in your key need to be, to be able to unlock the door; you just select the right key (the whole physical thing at once, with all the right tooths or holes) by recognising its form or its label.

The use of this proposed keyring has several advantages over the current practice of websites to use passwords for logging in. Instead of having to remember dozens of passwords for numerous sites, you only need to remember a key number for that site, in the range of 1 to 99. This key number stays the same for that website at all times, so you *can* remember it.

1.3 Assumptions and requirements

The following assumptions are made.

- The Secret keys (see section 3.1) need to be secret, inaccessible and unobtainable. Therefore, these keys are stored in an Hardware Security Module (HSM).
- There are three server roles involved with the login procedure:

  webserver This is the front-end server to which the user is communicating.

  accounts server This is the server that holds all account data. It stores all kind of user related data, but no userid or passwords; it stores hashes and site keys instead.

  login server This server uses an HSM which stores an array of secret keys per website it services.

  Other systems may exist but are not relevant in this article.

- The webserver is protected from the Internet by a firewall. This is standard already and should be continued. Having a firewall does not make a webserver impervious to attack, however, as many successful hacks have shown.

  The accounts server is protected from the (possibly hacked) webserver by another firewall.

- There is a secure channel (like a SSL tunnel) between the webserver and the accounts server.
- There is a secure channel (like a dedicated IPsec VPN) between the webserver and the login server. The login server may be anywhere on the Internet and may be owned by a different organisation than the webserver.

  When the login server is using a network HSM (not part of the physical hardware of the login server itself) a secure channel between these two systems is required.

- The user communicates with the webserver over a secure channel. The most obvious choice for this would be a single sided SSL/TLS connection (HTTPS) with a server certificate signed by one of the many Root CAs. This is current practice and need not be changed.
- The webserver should take countermeasures against attempts to guess login values by delaying each successive attempt with exponentially increasing waiting times. Should some valid login values be supplied (but wrong keys used) the webserver should temporarily lock the account when guessing persist. The user may be noticed by email when such locking has occurred.

2 Login for dummies

Not that you are one if you decide to read this section. It serves as a general description of the login process, without going into technical details too much.

2.1 What are we talking about?

Instead of choosing a traditional userid and a password, we let the computer work things out for us instead. Userids tend to be in short supply for people with first names like John, Peter, and Chris, or surnames like Jones or Smith. This is true for people in almost any country. And the larger the site, the rarer free userids.

Passwords tend to be weak, as people choose short, real-life words, like things or names. Not that people are not willing, but remembering too long, too difficult passwords is not easy. It is very frustrating when you cannot login because you have forgotten your password.

As a radical measure, we do away with all this! We will let the computers (yours, the one running the website, and another one dedicted for logins) do what they do best: compute. Give them some random bits to chew on, and they are in heaven.

2.2 Random values

The website holds no userids, but hashes of keyring identifiers. The website holds no passwords, but one part of a key, of which you have the other part, like the broken locket Annie clings to in the musical that bears her name.

Your key is different from the key the website has, but they are related to each other. It is not possible to login with the key that is stored on the website; you can only login with your key, which is verified with the key from the website.

And oh. . . , keys are created using a random generator. Keys are just long strings of random bits with no information whatsoever. Keys are kept on a keyring and have ordinal key numbers from 1 to 99 for easy recollection.

2.3 Loging in part I

Before starting the login process you select a key from your keyring; the one you know to belong to the website you are trying to login to.

To login you don't send your key to the server but generate a secret random value, which you encrypt with your key using a simple XOR operation. Your key is totally random and the secret random value you encrypt with it is also, well. . . totally random. Mixing these random bits gives another totally random value. The website will receive this value and will try to decrypt this with the key that belongs to your account (we will keep you in the dark for now, about how this works). When it has decrypted the random login value, it will know which secret random value you generated. Instead of telling you the secret random number, the website sends a hash value of it, because otherwise someone able to see the network traffic will instantly know your key.

If the website returns a hash value matching your secret random number, you know two things. First, you know that the website you are talking to can successfully decrypt your random value. It can only do this if it has a matching key. Second, you know you are talking to the same site that sent you your key when you applied for an

account. This implies that if you trusted that site then, you can trust this site now, as it can only be the same site.

2.4 Loging in part II

This is all great and very sophisticated, but the website cannot share your wonder about this, as you may well fake your enthusiasm, and lie about the correctness of the hash it has sent you. To put your money where your mouth is, the website will do the same as you and send you an encrypted secret random value. This cryptogram can only be decrypted by someone owning the right key for the account. Using XOR with the key you have originally selected from your keyring, you decrypt it and return a hash of this value to the website.

When the website receives a hash value matching its secret random number, it knows two things. First, the user on the other side can successfully decrypt the secret random number. It can only do this if it has a matching key. Second, the website knows that this key is from the same keyring that was used when applying for an account. This implies that if you trusted this user then, you can trust this user now, as it can only be the same user.

2.5 On site keys and user keys

Beware, the icky parts start here (a bit).

When applying for an account, a site key is created by a random generator. The user key is then computed as the encryption of the site key with a Secret Key, with a block cypher like AES.

When the user encrypts its secret random value with its key it uses XOR. This is a very simple and reversable encryption method. But as both key and value are utterly random, no information is stored in the encrypted value. The website needs to decrypt the value, and this can only be done with the user key. But the website does not store user keys, only site keys...

The solution to this may be a bit of a dissappointment and a cheat: the website temporarily regenerates the user key by encrypting it again with the Secret Key. Once it has the user key, it can decrypt the random value easily by XOR-ing it. Also, the cryptogram that is sent to the user is a random value XOR-red with this regenereated user key.

The clever part, however, is that all the crpytographic functions the website is required to perform take place inside a Hardware Security Module. Secret Keys can be put in the HSM and made to good use there, but can never be retrieved from it. The HSM computes all values needed for the login process, without ever revealing the keys that are used, not even to a hacker.

In the end, only random bits enter the HSM, and only random bits leave it.

2.6 More about keys

Every key has its lifetime, so Secret Keys need changing every so often, as a good security measure. The same goes for keys on the user's keyring. The login protocol is capable of changing keys on the website and keys on the keyring, without any effort on the user side. Well..., a little program will do it for you, without you noticing.

Logins can be granted, but new keys may not be, which results in the expiration of an account. A website may deny a user new keys when payment is due, giving users limited login rights. At any time new keys can be provided again.

To learn how this all works, how keys are used, what a website can do with logins, and more, can all be read in the next chapters.

3 Keys

3.1 Secret keys

The keys used to encrypt other keys—called Secret keys in this article—are used to perform $\texttt{AES}_{256}$ encryptions of site keys, and are 256 bits long.

Several Secret keys are used and are considered to be stored in array S, in this article. With this array, three values are defined.

1. The number of stored Secret keys is represented by m.

2. The constant[2] `MAX_KEYS`, which is the maximum number of keys that will be kept. The value of m starts at 0 and will reach `MAX_KEYS` eventually and grow no more. The value `MAX_KEYS` has a minimum of 2 (a new key and at least 1 old key). All values in the array S are shifted up one position when a new key is stored, which will always be inserted at $S[0]$. This way, with at least one key loaded ($m > 0$), it always holds that $S[0]$ is the newest key, and $S[m-1]$ is the oldest key. With $m > 1$, $S[1]$ is the jongest old key.

3. The constant `MAX_ACTIVE_KEYS`, which is the maximum number of keys that will be used for logging users in. This value lies in the range $[2, \texttt{MAX\_KEYS}]$.

   Subtracting `MAX_ACTIVE_KEYS` from `MAX_KEYS` gives the number of inactive keys which can be used to reactivate expired user keys. Users using a user key that is older than the oldest active Secret key cannot login, but their key can be restored with an inactive Secret key. If a key is used that is older than the oldest inactive key, the key cannot be restored and is lost forever.

3.2 Site keys

For each user, there exists a site key and a derived user key (see section 3.3). The site keys are stored in a table of a database, where the $\texttt{SHA}_{256}$ hash of the userid will act as the primary key for that table. Since the user only sends the $\texttt{SHA}_{256}$ hash of its userid, no actual userids will be stored.

Site keys are generated once by a pseudo-random generator, and need never be replaced.

Each time a user tries to login, the database is queried with the $\texttt{SHA}_{256}$ hash of the userid, and the site key for that user is returned for further processing.

3.3 User keys

A user key is computed by encrypting the site key for the user with the joungest (or current) Secret key, using the $\texttt{AES}_{256}$ algorithm.

The keys users get from the website are stored on a keyring (see section 3.5). These keys are automatically changed for new keys by the website when the Secret key is changed.

[2] Although declared a constant, the value of `MAX_KEYS` may change over time. When the login policy regarding the use of old keys is changed, more or fewer old keys will be stored. For this, `MAX_KEYS` needs to be changed.

3.4 Key lifetime and old keys

For enhanced security, Secret keys need to be replaced at regular intervals. This change of keys is initiated by the website, without the possibility to make individual arrangements with any of its users.

After changing the Secret key, it is no longer possible to login with any user key. Therefore, an array of Secret keys needs to be kept, allowing users to login using an old(er) Secret key. At least one old Secret key needs to be stored; otherwise, the Secret key cannot be changed without rendering all user keys permanently useless.

Upon successful login with an old key, the user will be provided with a new user key, with which it can login using the newest Secret key, from that moment on. Changing of user keys is seamless and the user might, just as well, be kept unaware of this.

The website's security policy should prescribe with what frequency Secret keys are to be changed, and how many old keys should be kept. If, for instance, the Secret key is changed every month, and 11 old keys are kept, users can still login using a key that is a year old. If the user key is older than that, the site is not able to verify the user's key any longer.

3.5 Storing user keys in a keyring

Keys for the user are collected and stored in a keyring. The keyring is a block of (at least) 100 random numbers with 128 bits, most of them dummy keys. The keyring will be considered an array Z in the rest of the article. Since all bits are entirely random in a keyring—for dummies and real keys—a keyring stores no information whatsoever.

Real keys in the keyring should be randomly put between the dummy keys.

3.5.1 Creating a keyring

A keyring is created by using a random generator of sufficient quality, which generates 100 random numbers of 128 bits. Key number zero is used to identify the keyring and will never change after its creation. The other 99 keys are dummies (random bits with no meaning whatsoever). These random numbers are then encrypted and written to a file[3] on a storage device. Unencrypted values should never be written to a storage device and only be kept in volatile memory. It is up to the user to remember how to decrypt its keyring.

3.5.2 Copying keys and keyrings

A keyring can be freely copied to other devices, so all keys are available there as well. The keys of one keyring can be copied to other keyrings, in different locations. It is up to the user to keep a registration of this.

Encrypted keyrings can be stored in a public place, for easy access, and act as a backup. A dedicated webserver can be employed for storing encrypted keyrings, which can be used to restore lost keyrings. They can also be used to login when away from home with no access to your own keyring. These temporary keyrings should be discarded when logging out.

[3] A password protected PKCS#12 file would do nicely.

3.5.3 Irretrievably lost keys and keyrings

When a key number is forgotten and no record of it can be found, the key must be considered lost.

As the user has supplied websites with personal information, it may be easy to regain login capabilities by just asking for a new key, provided a username and maybe some other information can be given. Selecting a free keynumber on the keyring and replacing that key with the new one should restore the ability to login.

Keyrings can become unusable or lost in two situations: the device holding it is defective (like a harddisk crash) and no backup has been made, or the encryption cannot be reversed (PIN forgotten). In both cases, the user has to start over and create a new keyring, containing only dummy keys.

4 Logging in

Having unencrypted its keyring, selected a key number (k), and entered the userid, the login process can commence. Keys used in this scheme may vary in length per website. We will use the value n to indicate the length of the site and user keys. See 6.2 on page 21 for a discussion of key lengths.

4.1 User IDs

Instead of sending a traditional alphanumeric userid, we send a hash. This hash will be used by the webserver to identify the user, and the webserver will only store hashes in its user database. This hash is specially crafted, so guessing will be hard.

The user will still need a userid, but this userid may be the same[4] for each and every webserver. The userid is chosen once and can remain the same as long as the keyring exists. The hash will be constructed using a combination of the userid and the keyring identifyer $Z[0]$. The hash value that is sent is constructed as follows. Compute the hash of the keyring identifier:

$$H_0 = \texttt{SHA}_{256}(Z[0])$$

Then replace the most significant bits of H_0 with the bit representation of the userid (a simple string like 'John Doe') to get H'_0. Then, compute the final 256-bit hash:

$$U_h = \texttt{SHA}_{256}(H'_0)$$

This hash value is a combination of something the user has (the keyring) and something the user knows (its chosen userid). This makes it hard to match a stolen set of hashes from a webserver with a set of keyrings from a keyring server.

4.2 Applying for an account

When a new user applies for an account, it presents the webserver with its hash value U_h. Furthermore, the user should select the index (d) of a hitherto unused key (therefore, a dummy).

$$K_d = Z[d] \textbf{ mod } 2^n$$

The user will send U_h, along with its dummy key K_d to the webserver.

The webserver will request values for a new user from the login server, by forwarding the dummy (K_d) and specifying for which website (W) the user will be granted access. The login server will do the following:

1. First, lookup a pre-shared key K_w, which belongs to the website W.

2. Generate a n-bit pseudo-random site key K_s for the user:

$$K_s = \texttt{Random}(n)$$

[4] Different userid's for different websites are still possible, but not necessary. The chosen userid acts like a default userid, or as a password for the keyring, if you like. The browser may remember the (default) userid for the user.

3. Encrypt the K_s with the current Secret key $S[0]$ to yield the new user key K_u, which is then encrypted with the users dummy key K_d:

$$K_u = \texttt{AES}_{256}(K_s, S[0])$$

$$K_x = K_u \oplus K_d$$

4. Finally, the two new values are encrypted with the pre-shared key K_w:

$$E_s = \texttt{AES}_{256}(K_s, K_w)$$

$$E_x = \texttt{AES}_{256}(K_x, K_w)$$

   And both are returned to the webserver.

The webserver will then relay the new keys E_s and E_u, and U_h to the accounts server. The accounts server will decrypt those values with it's own pre-shared key K_w:

$$K_s = \texttt{AES}_{256}(E_s, K_w)$$

$$K_x = \texttt{AES}_{256}(E_x, K_w)$$

It will then store the combination of K_s and U_h with the other data it already has for the user, and the encrypted user key K_x is sent to the user through a separate channel, like email.

The user can import the new key K_u into the keyring by calculating

$$K_u = K_x \oplus K_d$$

Finally, with something like

$$Z[d] = (\texttt{Random}(128 - n) << n) \oplus K_u$$

the dummy key K_d at $Z[d]$ is overwritten with 128 random bits, of which the last n bits contain the new key K_u.

4.3 Basic login scheme

The procedure described below to login is the basic login scheme.

4.3.1 Step 1: Knock, knock...

The user's login program has to compute the folowing:

1. An n-bit pseudo-random number R_u which hash will be returned by the webserver:

$$R_u = \texttt{Random}(n)$$

2. The user key K_u is taken from the keyring Z at index k. As this is 128-bit value, take the least significant n bits of it. The pseudo-random number is encrypted with it to get the value A_u:

$$K_u = Z[k] \textbf{ mod } 2^n$$

$$A_u = R_u \oplus K_u$$

3. The webserver will return a hash value of the user's pseudo-random value. To be able to verify this hash, we compute our own hash B_u to verify it with:

$$B_u = \texttt{SHA}_{256}(R_u) \textbf{ mod } 2^n$$

The special userid hash U_h and the encrypted pseudo-random number A_u are sent to the webserver.

4.3.2 Step 2: Site key lookup and key matching attempts

The webserver runs a login procedure, see Algorithm 4 on page 24.

After receiving U_h from the user, the site key K_s needs to be looked up. A query with U_h is sent by the webserver to the accounts server. If a match is found, K_s is returned; otherwise, K_s is set to zero, indicating that no such record exists. In the latter case, the values B_s and P_s are both set to zero as well, and returned to the user. The user needs to rethink its actions and start over.

If a match is found, an iterative process starts, trying to find the right Secret key to log the user in. The webserver will send three values to the login server: A_u, K_s, and a Secret key index i, starting at 0. This will be repeated (increasing index i), until a match is found, or no more keys can be tried.

4.3.3 Step 3: Login server actions

The login server is a processor of login values and calls a function of the HSM to compute them (see Algorithm 5 on page 25).

In case i is less than m (the number of stored Secret keys, see section 3.1 on page 8) the login server calculates the following, all within the HSM:

1. Temporarily, regenerate a user key, using the same algorithm as when the key was originally generated:

$$K_u = \texttt{AES}_{256}(K_s, S[i])$$

   with $S[i]$ the i-th Secret key stored in the HSM.

2. With the user key K_u, decrypt the pseudo-random the user has sent:

$$R_u = A_u \oplus K_u$$

3. Calculate a hash with which the user can verify we own the site key K_s and a corresponding Secret key $S[i]$:

$$B_s = \texttt{SHA}_{256}(R_u) \textbf{ mod } 2^n$$

4. To be able to verify that the user owns and knows its user key K_u, is not sending a replay of some earlier succesfull login sequence, and will be unable to ly about the correctness of the hash of the pseudo-random the webserver will send, we calculate a challenge for the user.

   If the index i equals zero, generate a pseudo-random value

$$R_s = \texttt{Random}(n)$$

otherwise, compute

$$R_s = \texttt{AES}_{256}(K_s, S[0])$$

which will be the new key for the user.

We then encrypt R_s to a value P_s the user can decrypt using its key:

$$P_s = R_s \oplus K_u$$

This value will be different for each time the loop is executed, even if the same new key is transmitted, since K_u will change each time.

5. Calculate the expected response also:

$$Q_s = \texttt{SHA}_{256}(R_s) \textbf{ mod } 2^n$$

The HSM will produce the values B_s, P_s, and Q_s as a result of the calculations and the login server will send them back to the webserver, as an answer to the three values it was given (A_u, K_s, and i). The HSM will delete all temporary values from memory directly afterwards, and remember only its Secret keys.

Should index i equal m, then the array of Secret keys is exhausted. If we reach this situation, we cannot log the user in since all possible attempts have failed. Return zero values for B_s, P_s, and Q_s to indicate this.

4.3.4 Step 4: user verifies site key

The webserver sends B_s and P_s to the user. If both are zero, the login has failed and the user should return to step 1. It is advisable for the user to choose different values for the next try.

If B_s contains a nonzero value, the user verifies whether this value equals B_u. If the two values do not match, i.e.:

$$B_u \neq B_s$$

the user either has selected the wrong key or uses an old key. A key can be old if it comes from a keyring that was not used recently. Another possible reason for a key to be old is when the Secret key of the site has changed recently.

To indicate that no match has been found, we send

$$Q_u = \texttt{0x0}$$

(n zeroes) to the webserver. The webserver will need to start over, and send values A_u, K_s, and $i + 1$ to the login server. So, back to step 3.

4.3.5 Step 5: site verifies user key

If B_s matches B_u, then the webserver has found a Secret key for the user. The user can now prove it has control over its user key by computing

$$R_s = P_s \oplus K_u$$

and

$$Q_u = \texttt{SHA}_{256}(R_s) \textbf{ mod } 2^n$$

which is sent to the webserver as proof. If the webserver accepts Q_u as correct it will log the user in.

If the webserver receives a value Q_u that does *not* match, something fishy is going on. Further attempts for this account, or from that source hould be scrutinized.

4.3.6 Step 6: key replacement

If this attempt to login was not the first with this key, the webserver has sent the user a new key in R_s (instead of a pseudo-random value). The user should store this new key in the keyring, overwriting the old key the user has just used:

$$Z[k] = (\texttt{Random}(128 - n) << n) \oplus R_s$$

where all bits of $Z[k]$ are replaced.

4.4 Optimizations and alternatives

With the basic login scheme several optimizations and alternatives are possible. Here are some obvious ones.

4.4.1 Aborting the login procedure

Logging in with an old key every now and then is inherent in this scheme, as most Secret keys will be changed at regular intervals. Therefore, it is likely to sometimes receive a hash value B_s that does not match B_u. An unsuccessful second attempt may be the result of using a seldomly used key or a wrong key. So, after at least two unsuccesful attempts the user may be presented a question whether it likes to select a different key from the ring, or continue trying with this key. If the user opts to select a different key, the login procedure has to be aborted. This may be done by setting

$$Q_u = \texttt{0x1}$$

and send this to the webserver (instead of `0x0`). We return to step 1.

4.4.2 Using last login date

Instead of always starting with Secret key $S[0]$, the date of the last login of the user can be taken into account. If the query with U_h as key, that is sent to the accounts database, would also return the last login date, a starting key index i could be selected. It is no use trying new keys when you know beforehand that these keys were added after the last successful login.

4.4.3 Conditional key replacement

For most websites, restoring user keys that refer to the most recent Secret key $S[0]$ will be done without hesitation. For some, refreshing keys will be done only when certain conditions are met, like payment of monthly fees, a certain number of reviews written, or some amount of data uploaded. Until then, logging in with a valid (but in this context a typically 'old') key is granted. But if, for instance, payment is overdue, the key will expire and logging in is no longer possible.

Instead of incrementing index i in step 2, the index is decremented. The HSM will just be sending random values for P_s in that case, for all values of index i. To indicate to the user that no new key is sent, the website will replace it with A_u

$$P_s = A_u$$

and return this value to the user. In this case, no keys should be decrypted or overwritten.

4.4.4 Certifying the login server

Allowing the user to verify it recieves trustworthy responses from the webserver, the webserver may present the user with a certificate, stating that it is using a bona fide login server. This certificate—signed by a regular Root CA—contains a public encryption/decryption key K_D. The private key for the login server (K_E) will be stored in its HSM.

Before sending B_s in step 3, encrypt it with K_E:

$$B'_s = \texttt{RSA}_{1024}(B_s, K_E)$$

In step 4, before comparing B_s with B_u, this value has to be decrypted using the public key K_D from the login server's certificate. So, if B_u equals $\texttt{RSA}_{1024}(B'_s, K_D)$ we proceed to step 5.

4.5 Changing Secret keys

At regular intervals a new Secret key is introduced and, at the same time, an old Secret key may be sent to the Eternal Hunting Grounds.

4.5.1 Terminated accounts

When deleting the oldest Secret key from memory, all accounts with a 'last login' date older than the installation date of the second oldest Secret key should be marked 'terminated'. Removal of the oldest key renders all those keys unusable. Those accounts can be purged, as they cannot be used again.

4.5.2 Expired accounts

In some situations user access must be barred until some condition is met. Accounts can be made temporarily inaccessible for this purpose by letting keys expire. Expiration can happen automatically when users do not login in time to get their keys replaced, or intently by denying key updates. Expired user keys are associated with Secret keys with an index in the range $[\texttt{MAX\_ACTIVE\_KEYS}, \texttt{MAX\_KEYS})$.

Expired accounts can easily be made active again by using inactive Secret keys for the login process (and then renew the key).

Expired accounts can be reinstated, but only before the associated Secret key is erased from memory. After that, there is no way to regenerate the user key for logging in. The user should apply for a new account if it wants to regain access.

5 Security proof

All exchange of data between user and webserver should be transported over a secure channel. Not that the login sequence would be directly vulnerable, but for the other data that is transported. Requiring a login implies the subsequent exchange of private, valuable, or secret data in almost all cases.

5.1 On random numbers

The login sequence is an exchange of random data. The random numbers used in this exchange are generated by two sources: the pseudo-random generator of the system running the webbrowser and that of the login server.

When encrypting valuable data, using pseudo-random numbers that are generated with weak algorithms, or are weak themselves, eases decryption. In this case, however, there is nothing valuable to encrypt; only random bits. Cryptanalysis of random data is very hard.

Having a poor pseudo-random generator on the system running the webbrowser, as is typically the case for home-use equipment like PCs, tablets, or smartphones, does not really hurt, because this is the "valuable data" that is encrypted. It does not really matter which value is used for this, in this login scheme (but see section 5.3).

The random numbers generated by the login server are of good quality, for they are created by the pseudo-random generator of the HSM. These random numbers are used for site keys and can be considered strong. User keys are directly dependent of these keys, so they can be considered strong as well.

5.2 Eavesdropping the connections

Somebody able to eavesdrop on the exchange of values between the user and the webserver will see several values being transmitted. An attempt is made to prove that these values are of little use to a hacker.

5.2.1 Applying for an account

To prevent having sets of keys in plain-text in one location, there is a pre-shared key, which is shared between the accounts server in the back-office and the login server somewhere on the Internet. The encrypted values pass the webserver, but the webserver has no copy of this key.

The webserver receives a dummy key of the user, which should not be sent to the accounts server. This way, no set of site key and user key will arrive at the accounts server.

5.2.1.1 Values passing the webserver A user fills in a form on a webpage to supply enough information for the creation of a new account. It also must send a dummy key and a hash. The webserver sends values to the login server and relays the results to the accounts server.

K_d A dummy key from the keyring. This dummy key has no information and cannot be used to decrypt anything at the webserver.

U_h A special $\texttt{SHA}_{256}$ hash (see section 4.1) dependent of the userid. Relayed as-is to the accounts server.

Email address To which the new key will be sent. Relayed as-is to the accounts server. Although this data has privacy aspects they are considered of no value in this context.

User details To fill the accounts database with. Relayed as-is to the accounts server. Although this data has privacy aspects they are considered of no value in this context.

E_s New site key (encrypted with pre-shared key) returned by the login server. The webserver has no knowledge of the pre-shared key, so K_s cannot be decrypted without obtaining this from the login server or the acccounts server.

E_x Encrypted key (encrypted with pre-shared key) returned by the login server. The webserver has the user dummy key, but with that key only the value $K_u \oplus K_w$ can be obtained.

5.2.1.2 Values passing the login server

K_d The dummy key the user will replace with the new key later on.

W An indication to which website the request pertains. Used to select the right pre-shared key for the exchange of values with the accounts server.

E_s New site key (encrypted with pre-shared key) returned to the webserver.

E_x Encrypted key (encrypted with pre-shared key) returned to the webserver.

5.2.1.3 Values passing the accounts server

U_h The $\texttt{SHA}_{256}$ hash value from the user.

E_s New site key (encrypted with pre-shared key) as obtained from the login server.

E_x Encrypted user key (encrypted with pre-shared key) as obtained from the login server.

K_x Encrypted user key is sent out over a separate channel.

Email address To which the new key will be sent.

User details To fill the accounts database with.

5.2.2 Logging in

5.2.2.1 Values passing the webserver

U_h A special $\texttt{SHA}_{256}$ hash dependent of $Z[0]$ and the userid is sent to the webserver. This user value is sent once per login attempt and passed to the accounts server.

K_s The site key belonging to U_h, sent by the accounts server. Eavesdropping on this traffic will give the hacker a set of combinations of values. Having K_s for each user is of no value, since the hacker does not have the means (array S in the HSM of the login server) to turning this into K_u which is needed to login.

A_u A random number XOR-ed with the user key K_u is sent to the webserver. The random number is different for each login attempt. This random number is most likely generated by a suboptimal pseudo-random generator, namely the generator of a PC, tablet, or smart phone. Even so, you cannot easily determine K_u from this value, since this value is sent once per login attempt. Harvesting large quantities is practially infeasible, so statistical analysis will fail.

B_s The login server tries to decrypt the random number R_u from the user by regenerating the user key K_u. It then returns the least significant 128 bits of the $\texttt{SHA}_{256}$ hash of the found random value to the webserver, which passes it to the user. This value is sent repeatedly until the right user key has been found. Since different user keys will be tried, the hash value will differ each time. None of the hashes returned this way give any hint to R_u nor K_u.

P_s The webserver also receives a random number XOR-ed with the regenerated user key K_u from the login server. If the login server chooses to change keys, the random number contained in P_s will be the new user key K_u but further undistinghuishable from any other random value. Since P_s depends on R_s (which is a good quality pseudo-random number from an HSM and different for each P_s) K_u cannot be calculated from a single or a series of P_s values.

Q_s The response of the user to the P_s challenge sent by the login server. Used for comparison with Q_u sent by the user.

Q_u The user returns the the least significant 128 bits of the $\texttt{SHA}_{256}$ of the random number from the webserver. The lower half of the $\texttt{SHA}_{256}$ value of R_s can never be used to calculate R_s, so therefore K_u can never be calculated from a single or a series of Q_u values.

The only practical data present at the webserver would be the set of all U_h values, since these values are a direct link to an account for the website. Logging in will not be possible; the only harm that can come from this is a denial-of-service attack, by trying to login with bogus keys, so that accounts are locked out for some time.

5.2.2.2 Values passing the login server

A_u The webserver sends this value to the login server: a random encrypted with the user key. The value B_s will be calculated in response to this.

K_s The site key belonging to the user. The login server will temporarily regenerate the user key K_u from this within the HSM.

i The index to use when selecting Secret keys. Increments with each attempt.

B_s The response to challenge A_u. This value is returned to the webserver.

P_s The challenge the webserver will send the user.

Q_s The answer to this challenge.

The values exchanged here (except i) are random values or hashes thereof. The random value R_s in the HSM used to create P_s and Q_s cannot be guessed from these two values, since only half the value of the $\texttt{SHA}_{256}$ hash is returned with Q_s. Therefore, the user key K_u is also secured. Since the Secret keys are kept in an

HSM, K_u cannot be derived from K_s. The K_s value cannot be related to any account from the webserver, as only the webserver knows to which login attempt these values belong and cannot be derived from any value exchanged here.

5.3 Manipulating values

The hacker has control over values U_h, A_u, and Q_u, which he or she can change to any bit pattern. Values U_h and A_u are sent once during a login.

Userid harvesting malware must replace the system function of generating random numbers and be able to intercept network packets before they are encrypted by the SSL/TLS software. That would mean replacing a function of the SSL library as well. Only then can they calculate the user key K_u, using the known random values, and filter out the userid hash U_h.

Sending a random value for U_h always gives you a response. In most cases a zero value for B_s and P_s are returned, indicating that no record exists belonging to U_h. Given the fact that U_k depends on $Z[0]$ and a userid, finding a valid U_h will only be possible when the keyring has been successfully decrypted. Generating specific values for U_h by guessing userid's and sending those to a webserver (along with a random value for A_u) might give rise to non-zero (but bogus) values for B_s and P_s. In that case a userid has been harvested. From that moment on each key in the keyring can be tried to see if it fits.

Suppose a valid U_h has been found. All the webserver will do with any value of A_u is decrypt it with a key dependent on U_h, and return the least significant 128 bits of the $\texttt{SHA}_{256}$ hash of this. Should $\texttt{SHA}_{256}$ somehow be totally reversible, having only half the value leaves 2^{128} possible values for the random value, so no user key or site key can be obtained this way.

6 Scalability

There are several degrees of scalability with this login scheme.

6.1 Keyring

Instead of a nice and round 100, you can put any number of keys in a keyring file. The value of 100 is just convenient; all keys are numbered with a two-digit number, which can easily be remembered.

Any number will do, from zero (which leaves only the keyring identifier, and allows you to login to zero websites) up to any amount of keys you bother to carry around with you. Having more keys than sites you want to login to is just a way to make things a bit harder for those that do not own the keyring to choose keys. If you are not happy with this, you can just add new keys as you acquire them, just as with real keys. (Nothing stops you from adding bogus brass and steel keys to the keyring that holds your carkeys. But I don't believe it is common practice to add BMW and Ford keys to a keyring of a Toyota owner.)

6.2 Key lengths

The length of keys is generally considered as an important aspect. The more bits, the better the key.

That is true if such a key is used to encrypt data that is in any way predictable, like, for example, a piece of text. If the encrypted data has patterns of any kind, you can directly work with intermediate decryption attempts. Statistical analysis of resulting bit patterns can reveil if a certain key or method is getting close or closer.

But all values encrypted with our keys are comprised of random bits only. This implies that any result of decryption has to be tried, to establish if the decryption was in any way successful. That would mean many login attempts (millions, billions, or more) which is infeasable. And that is just to crack a user key, which only gives you one login.

6.2.1 Minimum key length

With a Secret key fixed on 256 bits to accommodate the $\texttt{AES}_{256}$ encryption, all other keys can be a lot smaller than that. Theoretically, a 1-bit key could suffice, but this obviously is not strong enough, as it would take only two attempts to test all possible values.

To rule out any feasable brute force attempts (supposing that a site does not stop countless consecutive failed attempts) a key length of 24 bits should be enough. Assuming that a single login attempt, exhausting all Secret keys each time, could be done in a second (taking into account the network traffic to download and upload webpages and values), a 16 bit key would be cracked in at most 18 hours. 365 days have 8760 hours, and if Secret keys are replaced every half year, then, if an attempt takes more than 4380 hours, it must be considered futile. Doubling the time with each bit, a 24 bit key would take 4608 hours of continuous effort, which is well over half a year.

6.2.2 Maximum key length

There is no limit to the number of bits in a key, but using more and more bits for keys has its trade-offs. Keys longer than 64 bits require more processing, as they do not fit in CPU registers common today. But even keys longer than 32 bits may be subobtimal in some programming languages that are used to make client side login pages, or server side login programs.

Part of the exchanged values are least significant parts of $\texttt{SHA}_{256}$ hashes. These give ample proof of having the right key, but reveal nothing of the hashed value—even if the hash function should be reversible. Therefore, the number of bits of the user and site keys must not equal the number of bits of the hash result.

There are of course hash functions that produce longer hashes, like $\texttt{AES}_{512}$, but putting more than, say, 128 bits into a key brings no more security.

If this scheme is somehow flawed, it most likely will not because of insufficient key lengths. As such, 128-bit keys should be considered the maximum practical key length.

6.3 Old Secret keys

At least one old Secret key is required, if Secret keys are to be replaced every now and then. An HSM can store many keys, so there is no real practical limit.

6.4 Encryption algorithms

In this article, the basis for generating the keypairs for the site and the user is $\texttt{AES}_{256}$. All site keys and user keys are related using this algorithm, so it has to be secure. The encryption algorithm has to be a block cypher, using a secret key. A hashing function does not do the trick, as this would directly reveil user keys once you have the site keys. But any other block cypher may be employed for this task.

Hashes are computed using $\texttt{SHA}_{256}$ in this article. The hash values that are exchanged between website and user are incomplete by design, and just contain some of the least significant bits of the hash. The key length of the site and user keys should not exceed the lenght of the hash value, nor have equal length, as that would weaken the login algorithm. Using 128-bit keys, a hash function like `MD5` is not recommended, as this produces a 128-bit hash.

7 Implementation

7.1 Algorithms

The several algorithms explained in Section 4 are represented here in a consise manner.

7.1.1 Userid hashes

The hash that is used in the login procedure is composed of something you have (the keyring) and something you know (the userid). Together, they are one part of the values needed to login. It is calculated with Algorithm 1.

Algorithm 1 Computing the hash of the userid.

1: **procedure** UserIDHash($Z, userid$)
2: $H_0 \leftarrow \texttt{SHA}_{256}(Z[0])$ ▷ Hash the keyring identifier.
3: $H'_0 \leftarrow$ ReplaceMSB($H_0, userid$) ▷ Pepper it with the userid.
4: **return** $\texttt{SHA}_{256}(H'_0)$ ▷ This will be value U_h.

7.1.2 New account

Algorithm 2 is run by the login server to get the login values for a new user. It is called by the webserver when the user has provided all necessary data. The values returned will be relayed to the accounts server.

Algorithm 2 Generate values for a new account.

1: **procedure** NewAccount(K_d,W)
2: $K_w \leftarrow$ GetWebsiteKey(W) ▷ Get pre-shared key for website W.
3: $K_s \leftarrow$ Random(n) ▷ Generate n-bit pseudo-random number.
4: $K_u \leftarrow \texttt{AES}_{256}(K_s, S[0])$ ▷ This is the new user key.
5: $K_x \leftarrow K_u \oplus K_d$ ▷ Encrypt this with the dummy key.
6: **return** $\texttt{AES}_{256}(K_s, K_w), \texttt{AES}_{256}(K_x, K_w)$ ▷ Encrypt both values with K_w.

7.1.3 User login program

A user must complete Algorithm 3 on the following page successfully to login. It is called with the hash, the keyring, an index, and the decryption key from the certificate of the login server. The function AskToContinue is called with the attempts counter as parameter. Only after two attempts should the user be asked if further attempts should be tried. It is up to the implementer if this question is asked once, at every further attempt, or at some other interval. If no actual question is asked, the function can return `0` directly.

7.1.4 Webserver program

With Algorithm 4 on the next page the webserver determines whether a user should be granted access. This simple algorithm does not calculate anything, it just compares values and sends data around.

The accounts server can indicate to the webserver (through account status s, and the site key K_s) what is required of the user; either now or in the near future.

Algorithm 3 The login program of the user.

procedure UserLogin(U_h, Z, k, K_D)
  $R_u \leftarrow$ Random(n) ▷ Generate n-bit pseudo-random number.
  $K_u \leftarrow Z[k]$ ▷ Take key from keyring.
  $A_u \leftarrow R_u \oplus K_u$ ▷ Compute a challenge.
  $B_u \leftarrow \texttt{SHA}_{256}(R_u)$ **mod** 2^n ▷ And the response also.
  $Q_u \leftarrow 0$
  $j \leftarrow 0$ ▷ Attempts counter.
  $B_s, P_s \leftarrow$SendToWebserver(U_h, A_u) ▷ Wait for B_s and P_s.
  while $B_s \neq 0$ **do** ▷ Website is trying keys for us.
    if $B_u = \texttt{RSA}_{1024}(B_s, K_D)$ **then** ▷ Website found the right key!
      $R_s \leftarrow P_s \oplus K_u$
      $Q_u \leftarrow \texttt{SHA}_{256}(R_s)$ **mod** 2^n ▷ Compute response to P_s.
    else
      $Q_u \leftarrow$ AskToContinue(j) ▷ Return 0 to continue; 1 to stop.
    $B_s, P_s \leftarrow$SendToWebserver(Q_u) ▷ Answer to million dollar question.
    $j \leftarrow j + 1$
  if $Q_u > 1$ **and** $P_s > 1$ **then** ▷ Login succeeded.
    if $j > 1$ **then** ▷ Not the first attempt with this key.
      if $P_s \neq A_u$ **then** ▷ We are not denied a new key.
        UpdateKeyring(Z, k, R_s) ▷ New key is sent with R_s.

Algorithm 4 The login program of the webserver.

procedure WebserverLogin(U_h, A_u)
  $F \leftarrow 0$ ▷ Login state.
  $K_s, s \leftarrow$ GetAccountInfo(U_h) ▷ Query the accounts server.
  if $K_s \neq 0$ **then**
    $i \leftarrow 0$
    repeat
      $B_s, P_s, Q_s \leftarrow$HSM(A_u, K_s, W, i, n) ▷ Call this function on login server.
      if $Q_s \neq 0$ **then**
        $Q_u \leftarrow$SendToUser(B_s, P_s) ▷ Wait for Q_u.
      else ▷ Array S exhausted.
        $Q_u \leftarrow Q_s$ ▷ No point going on: terminate loop.
      if $Q_u = 1$ **then** ▷ User aborted login.
        $Q_s \leftarrow Q_u$ ▷ Terminate loop at user's request.
      if $s > 0$ **then** ▷ Something required from the user.
        $i \leftarrow i - 1$ ▷ Deny user new keys.
      else ▷ All is OK.
        $i \leftarrow i + 1$ ▷ Normal key index.
    until $Q_u = Q_s$
    $F \leftarrow Q_s$ ▷ Return Q_s by default.
    if $Q_s > 1$ **and** $s > 0$ **then** ▷ Login succeeded but something required.
      $F \leftarrow A_u$ ▷ Login granted for now.
  SendToUser($0, F$) ▷ Indicate login state.

7.1.5 Login attempts

Algorithm 5 computes values for the webserver to check.

Algorithm 5 The program of the login server, running inside the HSM.

```
1:  procedure HSM(A_u, K_s, W, i, n)
2:      I ← ABS(i)                          ▷ Index i may be negative to deny new keys.
3:      if I < MAX_ACTIVE_KEYS then         ▷ Use only active keys from array S.
4:          K_E ← GETSITEKEY(W)             ▷ Search RSA key for this site.
5:          K_u ← AES_256(K_s, S[I])        ▷ Temporarily regenerate user key.
6:          B_s ← RSA_1024(SHA_256(A_u ⊕ K_u) mod 2^n, K_E)   ▷ Compute response to A_u.
7:          if i > 0 then                   ▷ Not the first attempt with K_s.
8:              R_s ← AES_256(K_s, S[0])    ▷ Send user a new key.
9:          else                            ▷ First attempt or not allowed a new key.
10:             R_s ← RANDOM(n)             ▷ Generate n-bit pseudo-random number.
11:         P_s ← R_s ⊕ K_u                 ▷ Compute a challenge.
12:         Q_s ← SHA_256(R_s) mod 2^n       ▷ And the response also.
13:     else                                ▷ Array S is exhausted.
14:         B_s, P_s, Q_s ← 0, 0, 0         ▷ It's game over.
15:     return B_s, P_s, Q_s                ▷ Return these to the webserver.
```

7.2 Login webpages

The login algorithms for both the user and the server are presented as contiguous programs. Since there are several exchanges of values, and the fact that the user has no login program at its disposal, the algorithms need to be broken apart.

The website can present login code to the user through the inclusion of JavaScript in the HTML login pages. At the server side, PHP can be used to generate HTML with JavaScript. In this example implementation we use 32-bit random values.

7.2.1 First page

This can be a normal HTML page. It must contain JavaScript code to start the login process, by calculating Au, Uh, and the key index k, which the user must select. It also calculates Bu, and its value is stored in the sessionStorage of the browser. See Listing 1 on page 26.

7.2.2 Initial PHP page

The initial PHP page queries the accounts database for a user with hash U_h. If such user hash exists the site key K_s is returned and stored in the session array. Otherwise a page is displayed to inform the user the login process has failed due to a mismatch.

From that moment on, the webserver sends webpages to the browser that calculate the value Q_u. These pages are identical, except for the values of B_s and P_s that are sent along. After computation of Q_s, the form, containing an input element that will hold the Q_u value, is automatically submitted.

The initial PHP page sends the first of these (almost identical) webpages; the action option in the form will call the second PHP page for all subsequent calculations. See Listing 2 on page 26.

Listing 1: Initial Login Page

```
<!DOCTYPE html>
<html>
  <head>
    <title>Login 32-bit implementation</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="style/login.css" />
    <script src="http://localhost:8888/slwp/sha256.js"></script>
    <script src="http://localhost:8888/slwp/keyring.js"></script>
    <script>
      function StartLogin(keynumber)
      {
        var Ru = new Number(Math.floor(Math.random()*Math.pow(2, 31)));
        var H0 = SHA256(Z[0]);
        document.forms['login']['k'].value = keynumber;
        document.forms['login']['Au'].value = XorKu(Ru, keynumber);
        document.forms['login']['Uh'].value = parseInt(SHA256(UserId +
            H0.substr(UserId.length - 1)).substr(-8), 16);
        sessionStorage.Bu = parseInt(SHA256(Ru.toString()).substr(-8),
            16);
        sessionStorage.j = Number(0);
        document.login.submit();
      }
    </script>
  </head>
  <body>
    <h1>Secure Login without Passwords</h1>
    <input name="keyring" type="file">
    <form name="login" method="post" action="login2.php">
      <input id="k"  name="k"  type="password" hidden>
      <input id="Uh" name="Uh" type="password" hidden>
      <input id="Au" name="Au" type="password" hidden>
      <input id="ID" name="ID" type="text" value="default␣userid"
          readonly>
    </form>
    <div class=keypad>
      <script>DisplayKeys(true);</script>
    </div>
  </body>
</html>
```

Listing 2: Second Login Page

```
<?php
  session_start();
  require 'user_pages.php';
  $_SESSION["Ks"] = getAccountInfo($_POST["Uh"]);
  $_SESSION["Au"] = $_POST["Au"];
  $_SESSION["k"] = $_POST["k"];
  $_SESSION["i"] = 0;
  $Bs = $Qs = $Ps = 0;
  if ($_SESSION["Ks"] != 0)
    send_bs_ps();
  else
    login_failed("No␣such␣user");
?>
```

7.2.3 Second PHP page

The second, and only other, PHP page will compare values and send one of three possible webpages as a result of this: login succeeded, login failed, or another copy of the calculation page for Q_s. See Listing 3 on page 27.

Listing 3: Final Login Page

```
<?php
  session_start();
  require 'user_pages.php';
  if ($_POST["Qu"] == 1)
    $_SESSION["Qs"] = $_POST["Qu"];
  if ($_SESSION["s"] > 0)
    $_SESSION["i"] -= 1;
  else
    $_SESSION["i"] += 1;
  if ($_POST["Qu"] == $_SESSION["Qs"])
  {
    $F = $_SESSION["Qs"];
    if ($F > 1 && $_SESSION["s"] > 0)
      $F = $_SESSION["Au"];
    if ($_POST["Qu"] > 1 && $F > 1)
    {
      $_SESSION["login_ok"] = $F;
      login_succeeded("welcome.php");
    }
    else
      login_failed("cancelled");
  }
  else
    send_bs_ps();
?>
```

8 Conclusions

The security of the login process for websites can be greatly improved by using a keyring at the user side and an HSM employed by the website. Instead of sending relatively short, easy to guess strings (passwords) over the line, the use of encrypted random values, up to 128 bits each, is a big improvement. No keys are sent, just random values, which will be different each time a user logs in.

For hackers, getting login data in huge numbers will be very difficult, since this data is no longer stored centrally, but split between website and user. Each part alone has no value, and all values stored at the website render no valid login data without Secret keys. These keys are kept in very secure hardware: an HSM. Sniffing network traffic or collecting keystrokes with Trojans will not help. Keyrings have very high entropy and are encrypted, so to no direct use to hackers as well; they may be stored on the web for easy access and backup.

Several important security measures are automatically implemented: keys are changed frequently (as frequent as Secret keys are changed), they differ for each website, and keys on a keyring and the knowledge which key is used for what site constitutes two-factor authentication.

For the user, the way to logon to websites will change, but it will be an improvement over the burden of keeping track of all passwords. One userid for all sites and a single key number for a website is all you have to remember.

8.1 Advantages

In the following cases a keyring is superior to the use of passwords.

- Websites have all login information stored centrally. If an hacker can obtain this data and decrypt it, it has access to all—possibly millions—accounts at once [5].

  Using the keyring system there is no usable login information whatsoever at the server hosting the website. There is no way that the user information that is *present yield any usable login data. Hacking a website to obtain logins is useless. Other reasons to hack websites will remain, however, and using keyrings does not prevent hacking; it just eliminates one of the major attractions.*

- Users tend to have the same password and the same login name for several websites. A hack of an insufficiently protected site could yield valid usernames and passwords of perfectly protected sites. (Hack of www.babydump.nl yields at least 500 valid logins for www.kpn.nl.)

  Even if all user keys for a website were obtained in a hack, these would be useless for any other website, since they differ by definition.

- Websites require the user to change passwords. As more websites do this, more and more passwords a user has to remember, change. Ideally, no password for a website should be the same as for another website, but that is impractical. This would mean that each and every password needs to be written down, because the number of passwords is too much to remember for most. This thwarts the principle that passwords need to be remembered and never written down. The requirements to change passwords frequently and that they should differ from any other password is an inhuman task.

Using the keyring system, keys will differ for each website by definition and change regularly and automatically. Key numbers (the ordinal numbers in the keyring) don't change, so most of them can be remembered by the average human.

- Sometimes, getting unauthorized access to an account is as simple as just looking at the keyboard to see what the password is. The userid is always displayed when logging in, so shoulder surfing is very effective.

  Using a keyring, shoulder surfing cannot be used directly to login. Since a keyring is something you have to have, you cannot login using only the userid and the key number. You need to have access to the (unencrypted) keyring as well. Therefore, using a keyring is a basic form of 2-factor authentication.

- People tend to use weak passwords (unless a website specifically enforces the use of strong passwords) which can be guessed using specialized tools. If that yields no success, brute force attacks can be launched; to just try all possible passwords with limited length.

  Password guessing nor brute force attacks are an option when trying to login, since no passwords of any kind are exchanged. Even if the keys themselves would be used as an old-fashioned password, the search area would encompass 2^{128} *or* $3.4 \cdot 10^{38}$ *equally likely possibilities. Trying 1 million possibilities per second it would still take* 10^{32} *seconds to try them all.*

- The validity of the connection to websites is built on trust. HTTPS connections are protected using certificates. Sometimes trust only goes so far, and bogus but valid website certificates are used (Dorifel virus) or even the Root CA certificates are forged (see the DigiNotar hack). In that case, the user's trust is betrayed and the user left helpless.

  With the keyring solution no standalone substitute websites can exist; login data must be redirected to the real website. A substitute website does not have the right Secret keys. A user will notice this by wrong answers from the website and the login will abort from the user side.

- Visitors of websites are lured to other, well built fraudulent websites, mimicking websites of banks and such (phishing). Here, a simple mail can give a lot of trouble, redirecting users to an unsecure copy of a website, without the user suspecting anything.

  All communication to and from the malicious website can be passed on to the real website, to give the user the sense it is talking to the real site. Obtaining valid login data this way (as a man in the middle) is useless, since no keys are sent over the line. The data that is used to validate a user is meaningless for the next login.

- People are sometimes called by other people, claiming to be employees of banks. In order to "help" solve a problem, users are asked to give their login credentials. Some ignorent users are willing to oblige.

  With a keyring the only thing slightly useful to a hacker would be the userid. The keynumber is of no use, since the hacker has no access to the keyring itself; telling him which key index is used to login has no value.

8.2 Acknowledgements

Special thanks to Martijn Donders for his cryptographic support, and Rob Bloemer for reviewing the first draft of this article.

References

[1] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, December 1999.

[2] Mat Honan. Kill the password: Why a string of characters can't protect us anymore. *Wired Magazine*, 11 2012.

[3] Bruce Schneier. Write Down Your Password. Cryptogram Newsletter, June 2005.

[4] Toby Turner. Password rant, December 2012. http://www.youtube.com/watch?v=jQ7DBG3ISRY.

[5] Wikipedia. 2012 linkedin hack — wikipedia, the free encyclopedia, 2012. [Online; accessed 17-September-2012].

Listings