Secure Login without Passwords

T.M.C. Ruiter, $X \oplus R_{\text{key}}$

February 1, 2014

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

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.

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].) 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 14 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. 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 A new login approach

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

This proposed new way of logging in has several advantages over the current practice of websites requiring passwords. 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.

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 A new login scheme

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, or, just when you are starting to remember it, you need to change it again.

As a radical measure, we do away with all this!

Instead of userids, the website holds hashes based on keyring identifiers. Instead of a password for an account, the website holds 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 personal key, which is verified with the key from the website.

We will let the computers (yours, the one running the website, and another one dedicated for logins) do what they do best: compute. Give them some random bits to chew on, and they are in heaven. We will let humans (you, your neighbor and your fellow earthlings) do what they do best: remembering small ordinal numbers.

2.2 Storing your keys

Keys are just long strings of random bits with no information whatsoever. Your keys are personal and are stored locally.

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.

2.3 Computers and links

This proposed way of logging in concentrates on computing values and communication of the results. It is therefore appropriate at this point to elaborate a bit on who is communicating with whom and why.

2.3.1 Computer roles

Four computer roles can be distinguished when logging in.

Webserver This is the front-end server to which the user is communicating. From our point of view, it is the central system, acting as a hub. It communicates with three other servers: the accounts server, the login server, and of course the client computer. We consider it the most vulnerable to 'visits' with malicious intent, so it has been appointed the most simplests of tasks. It does not compute anything remotely valuable, it just compares values provided by others.

Accounts server This server is typically located in a network only accessible by the webserver. It stores all kind of user related account data, but no userid or passwords; it stores hashes and site keys instead. Like the webserver it does not compute anything either.

Login server This server uses an HSM which stores an array of secret keys per website it services. This server computes nothing on its own, it lets the HSM do all the work secretly, using some of its finest cryptographic functions. It receives login requests from the webserver and returns values that the webserver can compare or relay.

Client computer With this device the user communicates with the webserver. It generates random numbers and uses `XOR` and $\texttt{SHA}_{256}$ to compute the values exchanged with the webserver.

These roles can be played by as many computers as there are roles, but that need not always be the case. In fact, each and every role may be combined; a single computer may perform them all, for that matter.

The accounts server and the login server may be combined in a single server in the back-office. The webserver and the accounts server may be combined, with the login server somewhere on the Internet. The combination of webserver, accounts server, and login server is also valid; although this concentration of roles is not highly preferred.

2.3.2 Secure links

Traditionally, the communication channel between the user and the webserver is secured by means of TLS. The user sees the URL that starts with "https://", which means that the webserver has authenticated itself using a digital certificate. The browser will validate the supplied certificate using other trusted certificates. This security measure is sufficient but required.

The communication between the webserver and the accounts server should be conducted over a secure channel as well.

Finally, the communication between the webserver and the login server needs to be encrypted. For this, TLS is a sufficient security measure, although in this case mutual authentication is a must. The webserver needs to know it is talking to a legitimate and trusted login server, and the login server needs to know it is receiving valid requests. The certificate of the websever is used to select which array of secret keys to use, as the login server may service more than one webserver.

2.4 Logging in

2.4.1 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.

2.4.2 Part II

This is all great and very sophisticated, but the website wouldn't dare share your wonder about this, as you may well fake your enthusiasm, and flatly lie about the correctness of the hash it has sent you. To see if your claims hold, 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. Then, using `XOR` again, you combine the two random values you now have, and return this value to the website.

When the website receives a 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 Keys

2.5.1 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 $\texttt{AES}_{256}$.

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 user key is temporarily regenerated by encrypting it again with the Secret Key. Once the user key is available, the random value can easily be decrypted 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, or HSM for short. 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 with full control of the HSM.

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

2.5.2 Key lifetime

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

The keys used in this login scheme do not all have the same length. The Secret keys have 256 bits, for instance. The `RSA` operations use keys of 2048 bits. Keys stored in a keyring have 128 bits, but fewer bits may actually be used. The number of bits in such a key may differ among websites, and is denoted in this article as value n. See 3.5 on page 11 for a discussion of key lengths.

Three sort of keys are used: user keys (on a keyring), site keys (in a database), and Secret keys (in an array in an HSM). A user key is the result of encrypting the site key for that user with a Secret key.

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 to get user keys, and are 256 bits long. They reside in the HSM of a login server.

3.1.1 Array of Secret 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.

If there were only one Secret key, changing it would render all user keys permanently useless. Therefore, an array of Secret keys needs to be kept (with room for at least one old Secret key), allowing users to login using an old(er) key. Several Secret keys are used this way and are considered to be stored in array S, in this article.

1	2	...	MAX_ACTIVE_KEYS	...	MAX_KEYS

With this array, three values are defined.

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

2. The constant[1] `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 youngest 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 $[1, \texttt{MAX\_KEYS}]$.

   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.

[1] 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.

Upon successful login with an old key, the user will be provided with a new user key automatically, with which it can login using the newest Secret key, from that moment on. Changing of user keys is seamless (see section 3.3) 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.

3.1.2 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.

3.1.3 Expired accounts

In some situations user access may be barred until some condition is met (see also 4.3.6 on page 18). 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.

3.2 Site keys

For each user, a site key is generated once by a pseudo-random generator of the HSM of the login server (see section 4.2.3 on page 14), and need never be replaced.

The site keys are stored in a table of a database of the accounts server, where a hash value will act as the primary key for that table. Each time a user tries to login, the database is queried with the hash value the user supplies, and the site key for that user is returned for further processing.

3.3 User keys and keyrings

Together with the site key, a user key is computed by encrypting the site key for the user with the youngest (or current) Secret key, using the $\texttt{AES}_{256}$ algorithm (also see section 4.2.3 on page 14).

The keys users get from the website are stored on a keyring (see section 3.3.1 on the next page). The keyring will be considered an array Z in the rest of the article. A key in a keyring is automatically replaced with a new key by the webserver whenever the Secret key of the login server is changed (see section 4.3.6 on page 18).

3.3.1 Storing user keys in a keyring

A keyring is created by using a random generator of sufficient quality, generating a set of random numbers of 128 bits, stored in a file. Key number zero is used to identify the keyring and will never change after its creation. The other keys are dummies (random bits with no meaning whatsoever), some of which will be overwritten with real keys over time. Real keys in the keyring should optimally be put randomly between the dummy keys.

Typically, 100 random numbers are generated this way, so all keys are conveniently numbered with a one or two digit number (1 through 99), which can easily be remembered. However, you can put any number of keys in a keyring file. From zero keys (which leaves only the keyring identifier $Z[0]$, 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.

3.3.2 Copying keyrings

A keyring can be freely copied to other devices, so all keys are available there as well.

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.

3.3.3 Irretrievably lost keys and keyrings

When a key number is forgotten, or a key inadvertently overwritten, and no record or backup 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.

3.4 Certificates and key pairs

As part of the security of the login process, keys need to be exchanged in encrypted form to and from the login server. Furthermore, the website may want to signify to its users that it is using a trusted login server.

To accomodate for this requirement, the login server needs a certificate that holds a public encryption key. The private key that belongs to it is stored in the HSM. With this certificate the users of the login system can validate the trustworthyness of the used login server by validating the chain of trust. The certificate is signed by a Certificate Authority (CA). This CA's certificate is also signed, etcetera, up to a Root CA. A list of all trusted Root CA's is stored in the browser of the user and is already used to validate the certificates of websites.

3.4.1 Login server keys

The login server uses two types of $\texttt{RSA}_{2048}$ keys: an encryption key and a signing key; both private parts tucked away in its HSM. The public counterparts of these keys are presented to the user with a certificate, signed by a Certificate Authority so the user can validate them.

The encryption keys of the login server are called K_{le} for the public key ('l' for 'login'; 'e' for 'encryption'), and K_{LE} for the private key. The signing keys of the login server are called K_{ls} for the public key ('s' for 'signing'), and K_{LS} for the private key.

3.4.2 Accounts server keys

Like the login server, the accounts server needs its own set of keys for encryption of other keys. No signing is neccesary, so we have K_{ae} as the public key ('a' for 'account') and K_{AE} as the private key.

3.5 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.

3.5.1 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 32 bits should be more than adequate.

In a setup of 12 Secret keys, changed monthly, you cannot login after a year's worth of trying, since your key has expired by then. To crack a key you will need to try each one individually. Statistically, the average time to find it is half the time to test them all, so you must be kept busy for at least two years to regard a key safe 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), you can do 3600 test in one hour. Two years have 17520 hours in total, so $17520 \times 3600 = 63,072,000$ attempts could be done within that time. A 26-bit key would require $2^{26} = 67,108,864$ attempts.

But having 12 Secret keys would also give you 12 posibile hits with each attempt, dividing the time to find it by twelve. Adding another 4 bits to the key (30 bits) would give room for 16 Secret keys. A 32-bit key would enable you to have 64 Secret keys and still need more than 2 years of continuous effort to test all of them.

Keys with fewer than 32 bits are still viable, as long as other measures are taken against brute force attemtps. If that is the case, keys can be a lot shorter, without making a brute force attempt an attractive option.

3.5.2 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.

4 Logging in

Having unencrypted its keyring, selected a key number (k), and given its means of identification, the login process can commence.

4.1 User Identification

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

The user will still need something as a means of identification. The most basic form of this is to give a name (real or imaginary); in this case in the form of a string of letters. Another form may include biometric features, like an iris scan, a fingerprint, or hand geometry. Or, in contrast, a hardware token may provide a certain value.

The identification value is chosen once per website and can remain the same as long as the keyring exists. There are no restrictions or constraints as to the contents of this identification value, apart from it to be empty. This means that it may be the same and reused for each and every website.

The hash that is sent to the website to identify the user, will be constructed using a combination of the keyring identifyer $Z[0]$ and the identification value (`ID`), and is constructed as follows. Compute the $\texttt{SHA}_{256}$ hash of the keyring identifier and the ID:

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

$$H_1 = \texttt{SHA}_{256}(\texttt{ID})$$

Then, compute the final 256-bit value U_h by means of an `XOR` operation:

$$U_h = H_0 \oplus H_1$$

This hash value is a combination of something the user has, or has access to (the keyring) and something the user knows (its chosen name) or uniquely is (some biometric value) or phisically posesses (a hardware token). 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

See section 5.1 on page 19 for an accompanying flow diagram.

4.2.1 Step 1: Excuse me...

When a new user applies for an account, apart from any personal details the website is interested in, it presents the webserver with its hash value U_h. Furthermore, the user should select the index (d) of a hitherto unused key and use n bits of it, at the discretion of, and indicated by, the webserver.

$$K_d = Z[d] \bmod 2^n$$

Although this is a dummy key that is not used for logins (it will even be replaced after a successful application for an account) it still needs to be secured, since it is

used in the application process. Eventually, this key needs to find its way to the login server, so we will encrypt it with its public encryption key K_{le}:

$$K_h = E_{\texttt{RSA}}(K_d, K_{le})$$

The user will then send U_h, K_h, and its age, shoe size, and gender to the webserver.

4.2.2 Step 2: Howdy partner!

The users' particulars and its identification hash value U_h are relayed to the accounts server, which will create the account. The webserver will request values for a new user from the login server, by forwarding the dummy key (K_h), and indicating how long the keys should be (n bits long).

4.2.3 Step 3: What have we here?

The login server will do the following:

1. Generate a n-bit pseudo-random site key K_s for the user, which is then encrypted with the public key K_{ae} of the accounts server:

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

   $$K_y = E_{\texttt{RSA}}(K_s, K_{ae})$$

2. Encrypt the K_s with the current Secret key $S[0]$ to yield the new user key K_u.

   $$K_u = \texttt{AES}_{256}(K_s, S[0]) \textbf{ mod } 2^n$$

3. The (2048 bit) K_h value is decrypted with the private key K_{LE} to yield K_d:

   $$K_d = D_{\texttt{RSA}}(K_h, K_{LE})$$

   which is subsequently used to encrypt K_u:

   $$K_x = K_u \oplus K_d$$

   Both values K_x and K_y are returned to the webserver.

4.2.4 Step 4: Here are the results

The webserver will then relay the new encrypted key K_y and U_h to the accounts server. The accounts server will decrypt K_y with its private key K_{AE} to get the site key, but encrypt it again on the spot with the public key of the login server:

$$K_s = D_{\texttt{RSA}}(K_y, K_{AE})$$

$$K_a = E_{\texttt{RSA}}(K_s, K_{le})$$

and update the new account with it. These encrypted keys can be sent to the login server by the webserver during logins, without further processing.

4.2.5 Step 5: Thanks mate!

The encrypted user key K_x is presented to the user, so it can store it in its keyring. The user can decrypt its new key with by using K_d it used originally:

$$K_u = K_x \oplus K_d$$

Finally, K_u is imported in the keyring 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 Login scheme

The procedure described below to login is the full login scheme. See sections 5.2.1 on page 20, 5.2.2 on page 20, and 5.2.3 on page 22 for explanatory flow diagrams.

4.3.1 Step 1: Knock, knock...

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

1. The userid hash U_h (see section 4.1).
2. An n-bit pseudo-random number R_u:

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

3. 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:

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

$$A_u = R_u \oplus K_u$$

4. 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 28.

After receiving U_h from the user, the encrypted site key K_a 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_a is returned; otherwise, K_a 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.

Along with K_a the account status s may also be returned. This status may indicate whether we want to allow the user to login or not. If something is required from the user (payment, or otherwise) we might want to expire the account until the requirement is met. We use the value `MAX_ACTIVE_KEYS` to limit the number of keys we want to try.

If `MAX_ACTIVE_KEYS` is set to 1 then accounts will expire immetiately when a new Secret key is inserted. When set to 3, there will be a grace period of 2 times the Secret keys replacement interval. This means that login is granted for this time, in which the user can fulfill its debts. If that does not happen, the account will expire.

Users that have not logged in for a while, but have payed their monthly fees, can still login without problems because all Secret keys will be tried for such logins.

Now, an iterative process starts, trying to find the right Secret key to log the user in. The webserver will send five values to the login server: A_u; K_a; the number of bits in the user and site keys (n); a boolean value indicating if the user is capable of performing a verify operation using a public signing key (v); and a Secret key index i. This will be repeated (increasing index i), until a match is found, or no more keys can or will be tried.

Finally, we could consider reducing the number of login attempts. Instead of always starting with index 0 for 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 computed.

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 29).

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. Decrypt the site key, using the private encryption key K_{LE}:

$$K_s = D_{\texttt{RSA}}(K_a, K_{LE}) \textbf{ mod } 2^n$$

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

$$K_u = \texttt{AES}_{256}(K_s, S[i]) \textbf{ mod } 2^n$$

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

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

$$R_u = A_u \oplus K_u$$

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

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

5. If the user is able and wants to verify the signature of the login server, then the value B_s will be replaced with a signature thereof with the signing key K_{LS}:

   $$B_s = S_{\texttt{RSA}}(B_s, K_{LS})$$

   We use and send only the signature, not the B_s value also.

6. To be able to verify that the user owns and knows its user key K_u, is not sending a replay of some earlier successfull 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

   $$R_s = \texttt{AES}_{256}(K_s, S[0]) \textbf{ mod } 2^n$$

   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. This is because K_u will change each time, as it is the encryption of K_s with a different secret key $S[i]$.

7. We now know both random values. If the user knows them also (by successfully decrypting P_s) it can prove this by sending the `XOR` of both values:

   $$Q_s = R_s \oplus R_u$$

   This is the value that the webserver should compare.

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 five values it was given (A_u, K_a, n, v, 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 is nonzero, the user verifies whether this value matches with B_u.

If the user is not able to verify a digital signature, just compare B_u and B_s. Otherwise, it should try to verify the combination of B_u (the hash over R_u) and the signature thereof, which is in B_s:

$$V_{\texttt{RSA}}(B_u + B_s, K_{ls})$$

If the two values do not match, the user either has selected the wrong key or uses an old key. 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.

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_a, n, v, and $i + 1$ to the login server. So, back to step 3.

After two unsuccesful attempts the user may be presented a question whether it likes to abort the login procedure or continue trying with this key. To abort, use:

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

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

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 and Q_u:

$$R_s = P_s \oplus K_u$$

$$Q_u = R_s \oplus R_u$$

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 should be scrutinized.

4.3.6 Step 6: User key replacement

If this attempt to login was not the first with this key, the webserver may have 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.

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.

5 Schemes

Here are some example flows for logins with $n = 13$ (random values from 0 to 8191). Values between parentheses are calculated but not sent.

5.1 Applying for an account

Table 5.1 shows what happens when applying for a new account (see section 4.2 on page 13).

Table 1: Applying for a new account

Step	User	Value	Webserver	Value	Login server
1		$\Longleftarrow n$	$n = 13$		
2	$(K_d = 4444)$ $K_h = 7707$	$K_h \Longrightarrow$			
3			Create account	$K_h; n \Longrightarrow$	
4				$\Longleftarrow K_x; K_y$	$(K_s = 4289)$ $K_y = 1299$ $(K_u = 1093)$ $(K_d = 4444)$ $K_x = 5401$
5		$\Longleftarrow K_x$	$K_s = 4289$		
6	$K_u = 1093$				

Steps:

1. The website indicates how many bits are used for keys.
2. The user selects a dummy key K_d. This value is encrypted with public key K_{le} (a block of 2048 bits).
3. The webserver creates a new account. It forwards K_h unaltered to the login server, along with n.
4. The login server calculates a new site key K_s, and encrypts this with private key K_{LE} to get K_y (a block of 2048 bits). It decrypts K_h to get K_d. Finally, it calculates K_x from K_d and K_u. Values K_y and K_x are sent to the website.
5. The website decrypts K_y to get K_s, which it stores as key for the new account. Value K_x is sent to the user.
6. The user decrypts K_x with K_d and stores it in $Z[d]$.

5.2 Logging in

These are flow diagrams for successful and unsuccessful logins, as described in section 4.3 on page 15.

5.2.1 Simplest login scheme

Table 5.2.1 shows a login sequence with a valid and new key.

Table 2: Login with a new key

Step	User	Value	Webserver	Value	Login server
1		$\Longleftarrow n$	$n = 13$		
2	$U_h = xyz$ $(R_u = 8021)$ $A_u = 1234$ $(B_u = 5432)$	$A_u; U_h \Longrightarrow$	$i = -1$ $U_h \Rightarrow K_s$		
3			$i = i + 1$	$A_u; K_s; i; n \Longrightarrow$	$i < m$
4			$Q_s \neq 0$	$\Longleftarrow B_s; P_s; Q_s$	$B_s = 5432$ $(R_u = 8021)$ $P_s = 8172$ $(R_s = 2776)$ $Q_s = 5517$
5	$B_s \neq 0$	$\Longleftarrow B_s; P_s$	$B_s = 5432$ $P_s = 8172$ $(Q_s = 5517)$		
6	$B_u = B_s \Rightarrow$ $(R_s = 2776)$ $Q_u = 5517$	$Q_u \Longrightarrow$	$Q_u = Q_s$		
7	$B_s = 0$ Login OK	$\Longleftarrow B_s; P_s$	$B_s = 0$ $P_s = Q_s$		

Steps:

1. The website indicates how many bits are used for keys.
2. Calculate A_u from R_u. Send A_u and U_h to the webserver. Webserver has entry for U_h and finds K_s.
3. Start with $i = 0$ and send A_u, K_s, and i to the login server.
4. Login server calculates B_s, P_s, and Q_s, and sends them back to the webserver. The webserver finds that values are valid for a login.
5. The webserver takes B_s and P_s and sends them to the user. The user sees a nonzero value for B_s.
6. Since $B_u = B_s$ the user calculates R_s and from this Q_u. This value is sent to the webserver, which concludes that $Q_u = Q_s$.
7. To indicate that login is granted, the webserver returns $B_s = 0$.

5.2.2 Login scheme with old key

The login depicted in table 5.2.2 on the facing page succeeds but takes some more steps because an old key is used.

Table 3: Login with old key

Step	User	Value	Webserver	Value	Login server
1		$\Longleftarrow n$	$n = 13$		
2	$U_h = xyz$ $(R_u = 8021)$ $A_u = 1234$ $(B_u = 5432)$	 $A_u; U_h \Longrightarrow$	$n = 13$ $i = -1$ $U_h \Rightarrow K_s$		
3'			$i = i + 1$	$A_u; K_s; i; n \Longrightarrow$	$i < m$
4'			 $Q_s \neq 0$	 $\Longleftarrow B_s; P_s; Q_s$	$B_s = 1902$ $(R_u = 922)$ $P_s = 6300$ $(R_s = 4994)$ $Q_s = 4120$
5'	 $B_s \neq 0$	 $\Longleftarrow B_s; P_s$	$B_s = 1902$ $P_s = 6300$ $(Q_s = 4120)$		
6'	$B_u \neq B_s$ $Q_u = 0$	 $Q_u \Longrightarrow$	 $Q_u \neq Q_s$		
3'-6'*	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
3			$i = i + 1$	$A_u; K_s; i; n \Longrightarrow$	$i < m$
4			 $Q_s \neq 0$	 $\Longleftarrow B_s; P_s; Q_s$	$B_s = 5432$ $(R_u = 8021)$ $P_s = 8172$ $(R_s = 2776)$ $Q_s = 5517$
5	 $B_s \neq 0$	 $\Longleftarrow B_s; P_s$	$B_s = 5432$ $P_s = 8712$ $(Q_s = 5517)$		
6	$B_u = B_s \Rightarrow$ $(R_s = 5678)$ $Q_u = 5517$	 $Q_u \Longrightarrow$	 $Q_u = Q_s$		
7	 $B_s = 0$ Login OK	 $\Longleftarrow B_s; P_s$	$B_s = 0$ $P_s = Q_s$		
8	$P_s \neq A_u$ Update keyring				

Steps 1 and 2 are the same as in the diagram in section 5.2.1 on the preceding page. Steps 3' through 6' are repeated one or more times, but result in "wrong" answers from the webserver.

When the webserver has found the right index, the login server calculates values with the "right" Secret key. Steps 3 through 7 are then identical to those in section 5.2.1 on the facing page.

Finally, we know it took multiple attempts and additionally find that $P_s \neq A_u$, so we need to update the keyring in step 8.

5.2.3 Login scheme with wrong key

Using a wrong key won't get you in...

Table 4: Login with wrong key

Step	User	Value	Webserver	Value	Login server
1		$\Longleftarrow n$	$n = 13$		
2	$U_h = xyz$ $(R_u = 8021)$ $A_u = 1234$ $(B_u = 5432)$	$A_u; U_h \Longrightarrow$	$n = 13$ $i = -1$ $U_h \Rightarrow K_s$		
3'			$i = i + 1$	$A_u; K_s; i; n \Longrightarrow$	$i < m$
4'			$Q_s \neq 0$	$\Longleftarrow B_s; P_s; Q_s$	$B_s = 1902$ $(R_u = 922)$ $P_s = 6300$ $(R_s = 4994)$ $Q_s = 4120$
5'	$B_s \neq 0$	$\Longleftarrow B_s; P_s$	$B_s = 1902$ $P_s = 6300$ $(Q_s = 4120)$		
6'	$B_u \neq B_s$ $Q_u = 0$	$Q_u \Longrightarrow$	$Q_u \neq Q_s$		
3'-6'*	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
3			$i = i + 1$	$A_u; K_s; i; n \Longrightarrow$	$i \geq m$
4			$Q_s = 0$	$\Longleftarrow B_s; P_s; Q_s$	$B_s = 0$ $P_s = 0$ $Q_s = 0$
5	$B_s = 0$	$\Longleftarrow B_s; P_s$	$B_s = 0$ $P_s = 0$ $(Q_s = 0)$		
6	$P_s = Q_s = 0 \Rightarrow$ Login Failed				

The loop $3' - 6'*$ is repeated so many times that eventually $i \geq m$ (the number of keys in S).

6 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.

6.1 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 6.3).

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.

6.2 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.

6.2.1 Applying for an account

The dummy key that is used in the application procedure for a new account is encrypted with the key from a certificate of the login server.

6.2.1.1 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_h An encrypted dummy key from the keyring. Only the login server can decrypt this.

U_h A special value consisting of the `XOR` of two $\texttt{SHA}_{256}$ hashes (see section 4.1). Relayed as-is to the accounts server.

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.

K_y New encrypted site key returned by the login server. Only the accounts server can decrypt this.

K_x Encrypted user key returned by the login server. Only the user can decrypt this value.

6.2.1.2 Values passing the login server

K_h The encrypted dummy key which the login server decrypts using its private encryption key.

K_y New site key (encrypted with public key from accounts server). Returned to the webserver.

K_x New user key (encrypted with dummy key). Returned to the webserver.

6.2.1.3 Values passing the accounts server

U_h The hash value from the user.

K_y New site key which the accounts server decrypts with it private key.

User details To fill the accounts database with.

6.2.2 Logging in

6.2.2.1 Values passing the webserver

U_h The hash value is sent once per login attempt and passed to the accounts server.

K_a The encrypted site key belonging to U_h, as returned by the accounts server. Only the login server can decrypt this. Eavesdropping on this traffic will give the hacker a set of combinations of values. Having K_a for each user is of no value, however, since the hacker does not have the means (array S and the private key 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 either the least significant n bits of the $\texttt{SHA}_{256}$ hash of the found random value, or the signature thereof, to the webserver.
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 or signatures 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 `XOR` of both random values. Nothing can be deduced from this value.

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.

6.2.2.2 Values passing the login server The values A_u, K_a, n, v, and i are sent by the webserver to the login server.

A_u This is the user's cryptogram, as received by the webserver. It is a random value encrypted with the user key, which makes this also a random value. It is sent repeatedly (and unaltered) with each step in the login process. Nothing can be learned from this, as with the next login this value is changed completely.

K_a The encrypted site key belonging to the user. This value is encrypted with the public key K_{le} of a $\texttt{RSA}_{2048}$ keypair. The private key K_{LE} resides in the HSM. After decrypting it to the real site key K_s, the login server will temporarily regenerate the user key K_u from this. All decrypted values stay within the HSM.

n The number of bits there are in values K_s, P_s, and Q_s.

v A boolean indicating whether the user will verify signatures of B_s.

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

Values B_s, P_s, and Q_s are returned. See section 6.2.2.1 on the preceding page for a discussion of these values.

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 the least significant part 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_a. The K_a 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.

6.3 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 n bits of the $\texttt{SHA}_{256}$ hash of this. Should $\texttt{SHA}_{256}$ somehow be totally reversible, having only half the value leaves 2^n possible values for the random value, so no user key or site key can be obtained this way.

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: $H_1 \leftarrow \texttt{SHA}_{256}(\texttt{userid})$ ▷ Hash the userid.
4: **return** $H_0 \oplus H_1$ ▷ 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_h,n)
2: $K_s \leftarrow$ Random(n) ▷ Generate n-bit pseudo-random number.
3: $K_y \leftarrow \mathcal{E}_{\texttt{RSA}}(K_s, K_{ae})$ ▷ Encrypt K_s with accounts server public key.
4: $K_u \leftarrow \texttt{AES}_{256}(K_s, S[0])$ **mod** 2^n ▷ This is the new user key.
5: $K_x \leftarrow K_u \oplus \mathcal{D}_{\texttt{RSA}}(K_h, K_{LE})$ ▷ Encrypt this with the decrypted dummy key.
6: **return** K_x, K_y

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_{ls})
  $R_u \leftarrow$ 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.
  $B_u \leftarrow \mathtt{SHA}_{256}(R_u)$ **mod** 2^n ▷ And the response also.
  $Q_u, j \leftarrow 0, 0$ ▷ Initialize.
  $B_s, P_s \leftarrow$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 $\mathcal{V}_{\mathtt{RSA}}(B_u + B_s, K_{ls})$ **then** ▷ Verify B_s. Website found the right key!
      $R_s \leftarrow P_s \oplus K_u$
      $Q_u \leftarrow R_s \oplus R_u$ ▷ 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.
    $j \leftarrow j + 1$ ▷ One more attempt.
  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, v)
  $F, Q_s, Q_u \leftarrow 0, 41, 43$ ▷ Initialize.
  $k \leftarrow$ `MAX_KEYS` ▷ Use all keys.
  $K_a, s, D \leftarrow$ GetAccountInfo(U_h) ▷ Query the accounts server.
  if $s > 0$ **then** ▷ Something required from the user.
    $k \leftarrow$ `MAX_ACTIVE_KEYS` ▷ Use only some keys.
  if $K_a \neq 0$ **then**
    $i \leftarrow$ GetIndex(s, D) ▷ Use account status and last login date to get i.
    while $i < k$ **and** $Q_u \neq Q_s$ **do**
      $B_s, P_s, Q_s \leftarrow$HSM(A_u, K_a, n, v, i) ▷ 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.
      $i \leftarrow i + 1$
    $F \leftarrow Q_s$ ▷ 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 server program

Algorithm 5 computes values for the webserver to check. It uses private key K_{LE} for decryption of the site key. Private key K_{LS} is used to sign the user random R_u and yield B_s.

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

1: **procedure** HSM(A_u, K_a, n, v, i)
2: **if** $i < \texttt{MAX\_KEYS}$ **then** ▷ Use all keys from array S.
3: $K_s \leftarrow \mathcal{D}_{\texttt{RSA}}(K_a, K_{LE})$ **mod** 2^n ▷ Decrypt the site key K_s.
4: $K_u \leftarrow \texttt{AES}_{256}(K_s, S[i])$ **mod** 2^n ▷ Temporarily regenerate user key.
5: $R_u \leftarrow A_u \oplus K_u$ ▷ Calculate the user random.
6: $B_s \leftarrow \texttt{SHA}_{256}(R_u)$ **mod** 2^n ▷ Compute the hash over the random.
7: **if** $v = \texttt{true}$ **then** ▷ User can verify using certifcate.
8: $B_s \leftarrow \mathcal{S}_{\texttt{RSA}}(B_s, K_{LS})$ ▷ Replace hash with signature.
9: **if** $i > 0$ **then** ▷ Not the first attempt with K_s.
10: $R_s \leftarrow \texttt{AES}_{256}(K_s, S[0])$ **mod** 2^n ▷ Send user a new key.
11: **else** ▷ First attempt or not allowed a new key.
12: $R_s \leftarrow \textsc{Random}(n)$ ▷ Generate n-bit pseudo-random number.
13: $P_s \leftarrow R_s \oplus K_u$ ▷ Compute a challenge.
14: $Q_s \leftarrow R_s \oplus R_u$ ▷ And the response also.
15: **else** ▷ Array S is exhausted.
16: $B_s, P_s, Q_s \leftarrow 0, 0, 0$ ▷ It's game over.
17: **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 webserver 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.

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.

Algorithm 6 The function to start the login, included in the initial HTML page.

```
1: procedure StartLogin(k, userid)
2:     Ru ← Random(n)
3:     H0 ← SHA256(Z[0])
4:     H1 ← SHA256(userid)
5:     Au ← Ru ⊕ Z[k]
6:     Uh ← H0 ⊕ H1
7:     sessionStorage.Bu ← SHA256(Ru) mod 2^n
8:     sessionStorage.j ← 0
9:     Submit(Au,Uh)
```

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_a 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.

Algorithm 7 First PHP page.

```
1: procedure PHPlogin1(Uh, Au)
2:     $_SESSION[Ks] ← getAccountInfo(Uh)
3:     $_SESSION[Au] ← Au
4:     $_SESSION[i] ← 0
5:     Bs, Qs, Ps ← 0, 0, 0
6:     if $_SESSION[Ks] ≠ 0 then
7:         SendBsPs
8:     else
9:         LoginFailed("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.

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.

Acknowledgements

Thanks to Martijn Donders for his cryptographic support, and Rob Bloemer for reviewing the first draft of this article. The review sessions with Jannes Smitskamp were pleaseant and intense, and helped a lot to expose some hidden flaws; which were all remedied elegantly.

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].

List of Tables