Token-Based Multiparty Password Authenticated Key Retrieval Process

Pablo Guerrero – siriux@gmail.com – http://greenentropy.com – August 2014

Motivation and previous work

A Password Authenticated Key Retrieval (PAKR) process is any process that allows a client to retrieve a large key, only knowing a password that it's easier to remember than the key. A simple PAKR process would be to authenticate to a server (or any other system) using the password, and receive the key encrypted by the password. This is secure, simple and efficient, as long as you assume that the server has not been compromised by an attacker. Otherwise, the password and the key can be compromised.

Even if the server doesn't store the password in plain text, an attacker with access to the server can easily recover it. The reason is that usual passwords have low entropy, to make them easy to remember, and that allows to easily find the password using offline brute force.

A PAKR process that can resist the attack of one or more servers without compromising the key or the password, would greatly improve the client security in this scenario. At least three different processes, based on the use of multiple independent servers, have been already proposed to solve this problem.

The first one is based on exponentiation, where the client recovers a high entropy key using N different servers and it's password. This system can resist the attack of N – 1 servers while still preserving the password and key secrecy. The problem is that it's covered by at least the patent US7359507 and there have been serious doubts about it's security (“Security Analysis of Password-Authenticated Key Retrieval”, Shin - Kobara).

The second one is based on an equality comparison of a secret by two servers without revealing any additional information to any party. This system is much simpler and requires less work on the client, but it's covered by the patent US7725730 and it's only valid for two servers.

The third one is based on splitting the entropy of the password into pieces to independently authenticate to different servers that have shares of the key. This process is covered by the patent US6959394 and I have serious doubts of it's security in the real world if multiple servers are compromised, as the entropy of each piece is really small.

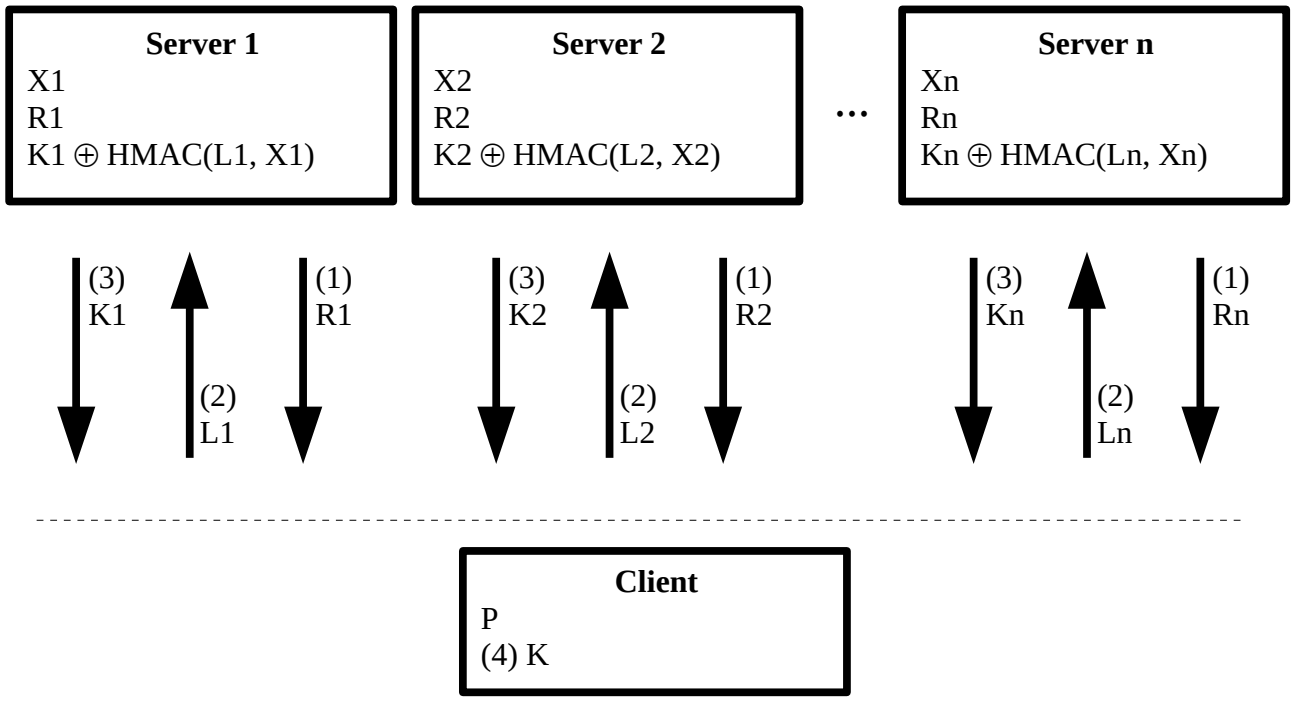
Therefore, a solution based on a different process (not covered by patents), that can securely solve the problem for N servers would be really interesting, specially for the Open Source community.

Proposed Process

The proposed process is based on single use pre-calculated tokens, that allow a client that knows a password **P**, a single try to retrieve a key (or any other strong secret) **K** with the help of **N** independent servers. **N**-1 servers can be compromised by an attacker without compromising the security of **P** or **K**.

First we will show the process using a single token. Then, we will address the problem of having a single token and how to avoid denial of service attacks.

Description of the process using a single token



Definitions

We have a client **C**, that knows a password **P** and wants to retrieve a key **K**.

We have **N** servers **Si** for i=1..**N** each one storing some information associated with a user. It stores a secret random number **Xi** and a set of tokens (one in this example). A token contains, for each **Si**, a random number **Ri** different for each token, and another number equal to **Ki** ⊕ HMAC(**Li**, **Xi**) where ⊕ means the XOR function.

The term **Li** is defined as the HMAC function of the ⊕ of all **Rj** with j≠i, using as secret **P**. In an example with **N**= 3, we would have **L2** = HMAC(**R1** ⊕ **R3**, **P**).

The term **Ki** is defined such as **K** = **K1** ⊕ **K2** ... ⊕ **Kn**. It's important that different tokens have different **Ki**. One way of achieving this is, for each token, generate **N**-1 random numbers for **Ki** with i=1..**N**-1, then **Kn** = **K** ⊕ **K1** … ⊕ **Kn-1**.

Process

With the definitions in place we can describe the process shown in the previous figure, where **C** retrieves **K**. The first step is to retrieve the **Ri** associated with the current token from all the servers.

The second step is for **C** to calculate all the **Li** using **Ri** and **P**, sending them to the respective servers.

In a third step, the servers will perform the HMAC of what they received in step 2, using **Xi**, and then ⊕ the previous HMAC with **Ki** ⊕ HMAC(**Li**, **Xi**). If the **Li** provided is correct (**C** has the right

password), we have exactly the same term in the ⊕ twice, and by a property of this function, they will cancel, giving **Ki**. This result is provided back to the client.

It's important to note here that **Si** cannot verify if the provided **Li** is correct or if the result of the ⊕ is **Ki**. If **Si** were able to verify any of this numbers, the security of the whole system would lose its properties. Therefore, the servers cannot know if the client has the correct password, and this process cannot be used to authenticate **C** with respect to **P**. If authentication is needed, you can do it in a standard way with respect to a different secret (e.g. user private key encrypted with **K**) after **K** is retrieved.

The final fourth step is performed by the client, who ⊕ all the received **Ki**. If, in fact, the client has the correct password, this will provide **K**, as shown in the definitions before.

Security

The secrecy of **P** and **K** is kept by the system, as long as the connection with the servers is authenticated (the client identifies the server) and secure, the random numbers are large enough to resist an offline brute force attack, at least one of the servers is not compromised by an attacker, and this server prevents an online brute force attack over **P.** For example, allowing a single try of the step two for each token and performing some kind of slow down if multiple tokens are allowed.

Optimization

You can save some storage space and one network round trip, by replacing all the uses of **R1** with **K1**, if both have the same size. This can be specially interesting for **N**=2 because it reduces the round trips from 4 to 3. The security of this optimization has not been reviewed with the same care as the original one.

Multiple tries

The previously described process allows a single try per token, as a token can only be used once, even if you fail to retrieve **K** for any reason. This can easily be extended to multiple tries by providing multiple tokens with different random numbers.

More tokens can be generated by anyone that knows **K** and a signing key (associated with the user) that it's registered with the PAKR servers, this way, the tokens can be verified. Each part of the token and it's signature must be encrypted with a public key associated with the corresponding server, as this data must remain secret to anyone but the server. It must also include some kind of sequence number that allows the servers to identify an already used token.

In particular, a client that correctly retrieves **K**, can use it to decrypt the signing key stored in a public place. Also, a device that is considered secure by the user (personal computer, pen drive, ...) can store this key.

Once generated, there are two alternatives to make them usable by a client. The first one is to just store them on the PAKR servers, allowing clients to perform a key retrieval try. The second option is to store them in a server not participating in the PARK (maybe public, cheaper or already available) from where the client retrieves it. Then, it can send each piece of the token to the corresponding server and the process can be executed as before.

Avoiding DoS attacks

Even if this system is secure with respect to the password and key secrecy, it's vulnerable to a denial of service attack (DoS) by anyone that consumes all the available tokens, even if it doesn't have the correct password.

A solution can be to store a large pool of tokens, as explained before, and add some kind of limitation on the speed an attacker can try with a new token (by IP, global, proof of work …).

If pool size limitations would require an inadmissible slowdown, you can extend the previous solution using a second very low entropy password **P2**. Before allowing the use of a token, whoever is storing it authenticates the client with respect to **P2** (this means that **P2** can be compromised if a single server is compromised). This greatly reduces the possibility of a DoS attack as a token will only be consumed when the provided secondary password is correct, allowing smaller pools and slowdowns.

It's also important that the store servers (PAKR or alternative servers) respond in exactly the same way even if the authentication fails, for example returning random numbers of the expected size, so that the attacker cannot guess **P2** using an online brute force attack and then use it for a faster DoS attack. When using an alternative server, the PAKR servers that receive an invalid token from a client (wrong **P2** on the alternative server) will answer with a new random **Ri** and **Ki** to avoid giving extra information to an attacker.

As **P2** can be easily compromised, we can only use **P2** authentication to provide tokens, any other use would be unsafe. And, as before, the server cannot know if the client has a valid **P**, because it doesn't obtain enough information to authenticate the client with respect to **P**.

Remembering two passwords can be difficult for a user, but we can use alternative systems that simplify this task. A simple way can be to randomize 100 numbers and some colors on the client using the previously introduced **P** as seed. Then, the client presents a table with 100 icons to the user, ordered by the random numbers and using the random colors. Finally, the user picks 3 memorized icons, which provides enough entropy to generate **P2.**

The user has 3 different clues for each pick (color, position, icon), so this will probably make really easy for the user to remember **P2**. This method also allows the user to identify a mistyped **P** early, based on the colors and the positions of the icons.

Conclusions

The proposed process has the advantages of being much more simple to prove secure with respect to the first previous process and of being extensible to **N** servers with respect to the second one. It has a reasonable number of round trips that grows linearly with **N**, and it's not covered by any patents I'm aware of, allowing anyone to use it freely.

For all these reasons, many real world use cases could benefit of the newly proposed process.

There is still room for improvements, like reducing the storage cost for the tokens or simplifying the measures against DoS attacks. It's easy to imagine some of these improvements, as some parts of the process could be replaced with equivalent ones shared in a larger system.