



Security in Networked Computing Systems' Project

Home banking login service with One Time Password

Marco Micera, Riccardo Rocchi

A.Y. 2016/2017

Outline

1. Abstract	1
2. Basic infrastructure	1
2.1 Pre-existing physical architecture	1
2.2 remoteServer's database schema.....	2
3. Application logic	2
3.1 Sequence diagram	2
3.2 OTP checking flowchart	4
4. Simple BAN logic analysis	7
4.1 Assumptions	7
4.2 Real protocol	8
4.3 Idealized protocol	8
4.4 Analysis.....	8
5. Conclusions	10

1. Abstract

This project's aim is to understand and develop a basic home-banking login service: additional security is obviously required in those kind of applications, therefore the user will be asked to send a One Time Password during the login phase.

The OTP is generated by a smart card given by his/her bank at registration time:



A typical OTP dongle

This extra security measurement shouldn't of course affect the perceived usability of the service, so we will introduce our protocol proposal that will be using OTP windows to overcome some de-synchronization problems.

2. Basic infrastructure

2.1 Pre-existing physical architecture

We start by presenting a reasonable pre-existing physical architecture sketch for a Bank organization:



Basic physical architecture

Users will be performing their login requests to their closest local application server available in their region: we suppose this kind of servers do not contain any critical information about their users, because they are located in a de-militarized zone (DMZ).

On the other hand, a database server (that could be centralized or distributed) is firewalled properly in order to allow communications only from desired sources: this is also guaranteed by the use of SSL, that exploits both local servers' and database server's certificates.

Attackers will then more likely choose local servers as their target: if one of them gets compromised, an attacker would not find any critical information stored in it.

Upon receiving some credentials by some user who is attempting to login, the attacker would then be able to get those by decrypting them correctly with SSL; the attacker could also receive from the database server the user's dongle counter and key if the previous credentials were correct. To sum up, a compromised local server only affects users who are attempting to login through it.

We will be referring to the local application server as the *localServer* (following the software denotation) and to the database server as the *remoteServer*.

2.2 *remoteServer*'s database schema

For each user, the *remoteServer* stores the following informations in its database:

- *username*
- *password*
- *dongle_key*
- *dongle_counter*
- *large_window_on*
- *large_window_otp*

Almost all of the information above is encrypted by means of AES-128 in ECB mode with PKCS5 padding before it gets stored.

Only the database server must be able to encrypt and decrypt data in its database, so symmetric encryption represents the best choice since it allows faster encrypting and decrypting than asymmetric encryption.

The 128-bit key though is now a system's single point of failure: once it gets compromised data is no longer secret.

The ECB encryption mode avoids us to store different Initialization Vectors for each user, but it doesn't hide data pattern: for this reason, the binary *large_window_on* variable is stored in clear.

This is acceptable since it does not represent critical information, as the *large_window_on* variable is supposed to be deactivated most of time for all users (see Paragraph 3.2: [OTP checking flowchart](#)), and when it gets activated it stays on for a very short period of time due to the fact that after a second login attempt, whatever it is successful or not, the *large_window_on* variable gets reset to zero.

Even if an attacker succeeds to know it and this parameter is set to one at a given time, his/her chances to broke the system are very low.

An attacker that somehow succeeds to send an OTP belonging to the *large_window* must then find a new OTP code (also belonging to the *large_window*) to get authenticated.

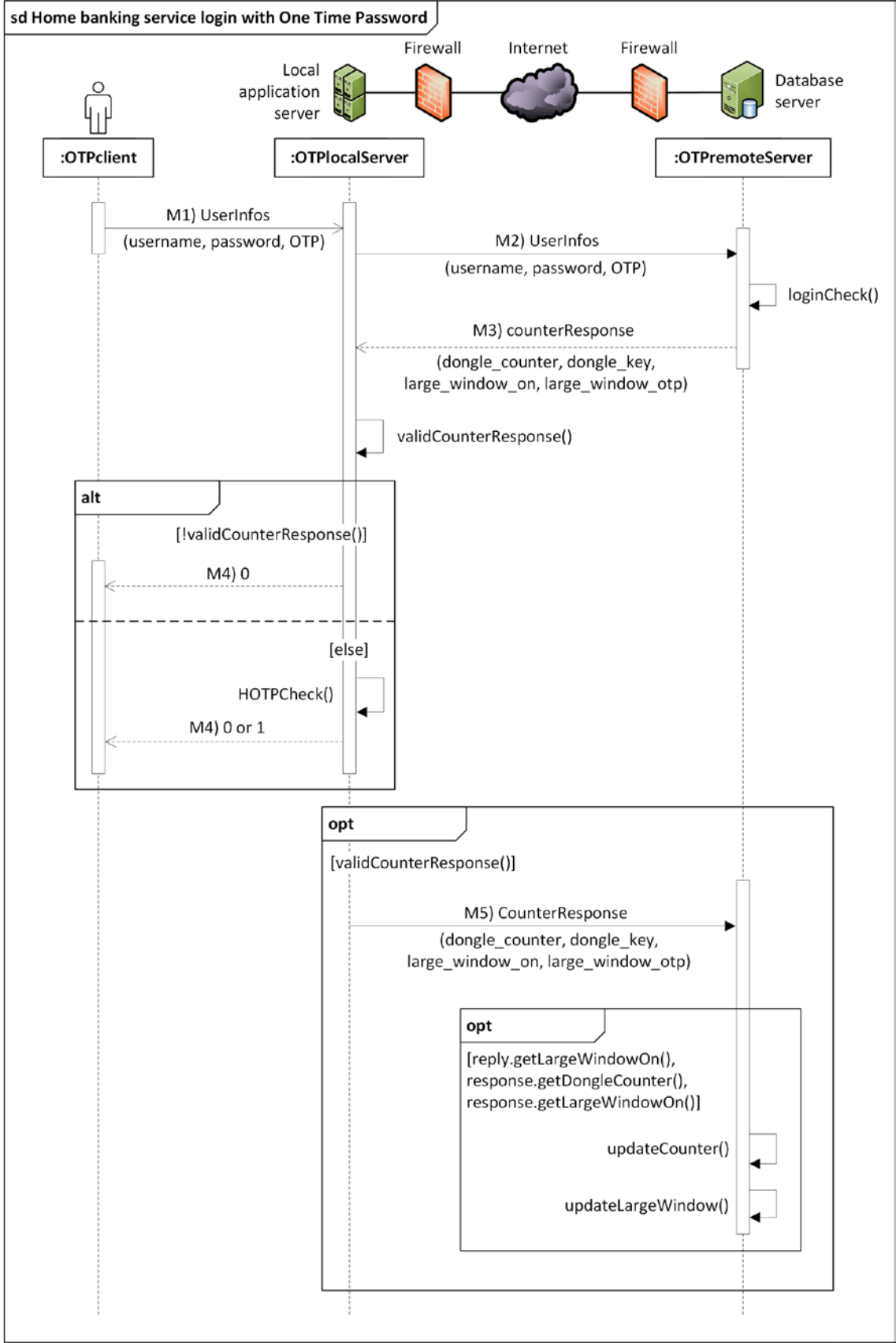
This is very unlikely because the adversary has a probability of $30/10^6$ (where 30 are the number of OTP values valid for the second attempt, and 10^6 is the total number of OTP codes) to authenticate him/herself once he/she knows that the user's *large_window* is on.

This mechanism is better explained in the [Application logic paragraph](#).

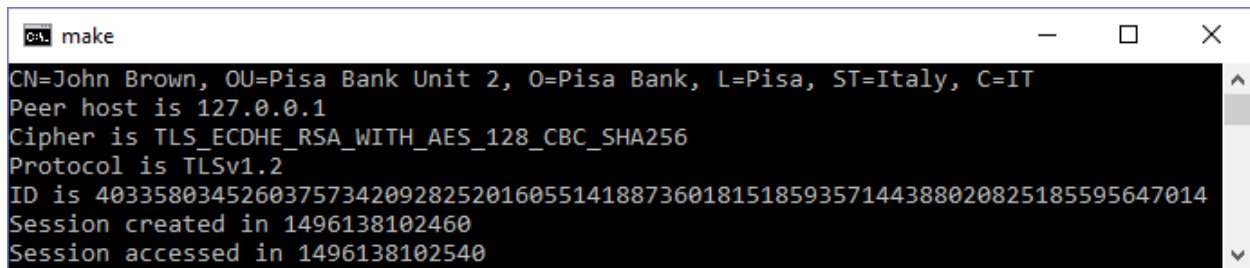
3. Application logic

3.1 Sequence diagram

Now that we have defined a system's basic infrastructure, we can now proceed by showing a simple sequence diagram:



All communications between object instances are protected by SSL (local servers and the database server already have their own certificate).

A screenshot of a terminal window titled 'make'. The terminal output shows the following text:

```
CN=John Brown, OU=Pisa Bank Unit 2, O=Pisa Bank, L=Pisa, ST=Italy, C=IT
Peer host is 127.0.0.1
Cipher is TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
Protocol is TLSv1.2
ID is 40335803452603757342092825201605514188736018151859357144388020825185595647014
Session created in 1496138102460
Session accessed in 1496138102540
```

remoteServer's self-signed certificate example used in this project

Both *remoteServer* and *localServer* are multithreading, so they are obviously able to serve more users at the same time.

3.2 OTP checking flowchart

The user starts by sending his/her credentials, consisting in a username, password and a OTP: the latter is generated by the "OTP dongle", a smart card given to the user by the Bank at registration time, as seen in the [Abstract](#).

This smart card has a secret key in it, called *dongle_key*, which is pre-shared with the Bank, moreover only the *remoteServer* knows it since it is stored encrypted in its database.

Whenever a new client signs up a contract with the Bank, the latter will take care of sending the smart card to the new customer.

This secret key it's used to generate the HOTP (HMAC-Based One-Time Password) using an algorithm better described in the [RFC-4226](#) published in 2005: this algorithm simply outputs an OTP given two values, the secret key mentioned above (*dongle_key*) and the *dongle_counter*.

Every time the dongle button is pressed (in order to generate a new OTP), the counter stored in the smart card is incremented, so that it will generate a different OTP afterwards.

When the *UserInfos* object (composed by username, password and the OTP) finally arrives to the *remoteServer* (M2) through the *localServer* (M1) the process that checks whatever the user had all the correct inputs starts.

Of course, every attempt fail is reported to the client as a "Login error, please try again" message: in this way attackers won't get any further information about the cause that made the authentication fail.

The first step is to make sure that the user is registered to the Bank, by checking the relative entry in the *remoteServer's* database and whether his/her password is correct or not.

Then, the *remoteServer* sends a message (M3) to the *localServer* containing all the informations needed to generate the HOTP code, so it will then be able to match it with the one sent by the user.

The *localServer* implements this task using a mechanism that allows some sort of synchronization between the counter stored in the bank's database and the smart card's one.

This is possible thanks to the usage of two different windows:

- *narrow_window* (used in the normal case scenarios)
- *large_window* (used when the smart card's counter goes out the *narrow_window*)

At first the *localServer* tries to search for a correct OTP inside the *narrow_window*.

If it can't find such OTP value inside this window, the *localServer* tries to search for it inside the *large_window*.

If no valid OTP is found in both *narrow_window* and *large_window*, then the *localServer* transmits the message M4 as 0, indicating that the authentication has failed.

Meanwhile if a valid OTP is found in the *large_window*, it is possible that the smart card's counter has gone out of the *narrow_window*.

A simple scenario of this synchronization error could be a careless user who generates different OTPs without performing any login, incrementing the counter stored in the smart card.

Now the *localServer* has to prove if it is just a coincidence or a case of an attacker's sheer luck.

In order to do that the *localServer* sends a command (M5) to the *remoteServer* imposing it to activate the *large_window* for the user, and to store in its database what OTP caused this event.

After the last login has failed, the client should try to perform a new login operation: now, after checking his/her username and password, the *localServer* knows if the user's *large_window* is activated, thanks to the the *CounterResponse* object in M3: the *localServer* will then search for a valid OTP inside the *large_window*.

If no matching OTPs are found, the *localServer* sends a message M5 to the *remoteServer*, imposing it to deactivate the *large_window*.

When instead a new matching OTP is found, (different from the one stored in the *remoteServer*'s database and still belonging to the *large_window*) it means that most likely a synchronization error has occurred.

The *localServer* now proceeds to send the new counter's correct value to the *remoteServer* (M5), that simply deactivates the *large_window* and updates the counter stored in its database associated to that specific user.



Our protocol proposal's entire flowchart

4. Simple BAN logic analysis

We are now going to analyze our protocol proposal by means of the BAN logic, an important tool for analyzing cryptographic protocols.

4.1 Assumptions

1. LS' certificate:

a. $C| \equiv \overset{K_{LS}^+}{\mapsto} LS$

b. $RS| \equiv \overset{K_{LS}^+}{\mapsto} LS$

2. RS' certificate:

a. $LS| \equiv \overset{K_{RS}^+}{\mapsto} RS$

3. SSL session key establishment between C and LS:

a. $C| \equiv C \overset{K_{C,LS}}{\longleftrightarrow} LS$

b. $LS| \equiv C \overset{K_{C,LS}}{\longleftrightarrow} LS$

c. $C| \equiv LS| \equiv C \overset{K_{C,LS}}{\longleftrightarrow} LS$

d. $LS| \equiv C| \equiv C \overset{K_{C,LS}}{\longleftrightarrow} LS$

4. SSL session key establishment between LS and RS:

a. $RS| \equiv LS \overset{K_{LS,RS}}{\longleftrightarrow} RS$

b. $LS| \equiv LS \overset{K_{LS,RS}}{\longleftrightarrow} RS$

c. $LS| \equiv RS| \equiv LS \overset{K_{LS,RS}}{\longleftrightarrow} RS$

d. $RS| \equiv LS| \equiv LS \overset{K_{LS,RS}}{\longleftrightarrow} RS$

5. *dongle_key* establishment at registration time

a. $C| \equiv C \overset{dongle_key}{\longleftrightarrow} RS$

b. $RS| \equiv C \overset{dongle_key}{\longleftrightarrow} RS$

6. The *remoteServer* RS is an authority on generating shared keys:

a. $RS \Rightarrow C \overset{dongle_key}{\longleftrightarrow} RS$

b. $LS| \equiv RS \Rightarrow C \overset{dongle_key}{\longleftrightarrow} RS$

Every *localServer* LS believes the previous statement because they belong the to same Bank organization.

4.2 Real protocol

In message M1, the client C sends the OTP to the *localServer*.

The OTP is actually a HOTP, an HMAC-based One-Time-Password: this is basically an hash function digest on the user's *dongle_counter* concatenated with its *dongle_key*. Also other operations are performed on the digest, but they are irrelevant to this purpose.

In order to follow the BAN logic notation, we can represent the HOTP (a keyed-hash) as the *dongle_counter* encrypted by means of the *dongle_key*, that is $\{dongle_counter\}_{dongle_key}$.

M1) $C \rightarrow LS: \{username, password, \{dongle_counter\}_{dongle_key}\}_{K_{C,LS}}$

M2) $LS \rightarrow RS: \{username, password, \{dongle_counter\}_{dongle_key}\}_{K_{LS,RS}}$

M3) $RS \rightarrow LS: \{dongle_counter, dongle_key, large_window_on, large_window_otp\}_{K_{LS,RS}}$

M4) $LS \rightarrow C: \{0 \text{ or } 1\}_{K_{C,LS}}$

M5) $LS \rightarrow RS: \{dongle_counter, dongle_key, large_window_on, large_window_otp\}_{K_{LS,RS}}$

4.3 Idealized protocol

Only M3 changes as follows:

M3) $RS \rightarrow LS:$

$$\left\{ C \xleftrightarrow{dongle_counter} RS, C \xleftrightarrow{dongle_key} RS, large_window_on, large_window_otp \right\}_{K_{LS,RS}}$$

4.4 Analysis

Every message can be correctly de-crypted from each receiver, thanks to assumptions 3a, 3b, 4a and 4b.

Freshness is guaranteed by the SSL handshake protocol: in fact, nonces contained in *client_hello* and *server_hello* messages allow both parties to generate a fresh master secret in order to avoid replay attacks.

Let us see an example in BAN logic terms for LS upon receiving message M1.

Let:

$$M1 = (username, password, \{dongle_counter\}_{dongle_key})$$

Using assumption 3b and the message meaning rule (1st postulate):

$$\frac{LS | \equiv C \xleftrightarrow{K_{C,LS}} LS, LS \triangleleft \{M1\}_{K_{C,LS}}}{LS | \equiv C | \sim M1}$$

Also, freshness is guaranteed by the SSL handshake protocol, so thanks to the nonce verification rule (2nd postulate):

$$\frac{LS | \equiv \#(M1), LS | \equiv C | \sim M1}{LS | \equiv C | \equiv M1}$$

This of course applies to each message exchanged in our protocol proposal.

Following the same reasoning, we could also state that LS, upon receiving message M3:

$$LS| \equiv RS| \equiv \left(C \xleftrightarrow{dongle_counter} RS, C \xleftrightarrow{dongle_key} RS, large_window_on, large_window_otp \right)$$

This of course implies:

$$\frac{LS| \equiv RS| \equiv \left(C \xleftrightarrow{dongle_counter} RS, C \xleftrightarrow{dongle_key} RS, large_window_on, large_window_otp \right)}{LS| \equiv RS| \equiv C \xleftrightarrow{dongle_key} RS}$$

By applying the jurisdiction rule (3rd postulate) with assumption 6b, we obtain:

$$\frac{LS| \equiv RS| \equiv C \xleftrightarrow{dongle_key} RS, LS| \equiv RS \Rightarrow C \xleftrightarrow{dongle_key} RS}{LS| \equiv C \xleftrightarrow{dongle_key} RS}$$

After M1, this result can be applied to the message meaning rule (1st postulate):

$$\frac{LS| \equiv C \xleftrightarrow{dongle_key} RS, LS \triangleleft \{dongle_counter\}_{dongle_key}}{LS| \equiv C | \sim (dongle_counter)}$$

Where $LS \triangleleft \{dongle_counter\}_{dongle_key}$ is due to the fact that the *localServer* has received the HOTP code $\{dongle_counter\}_{dongle_key}$ in the first message, encrypted with the symmetric session key $K_{C,LS}$ established by SSL between the *client* and the *localServer*.

The *localServer* now does not have to believe that the *dongle_counter* quantity is fresh: it could have been sent previously, even by a legitimate user (e.g.: by distraction).

It is enough for the *localServer* to believe that the *client* C “once said” (sent) its *dongle_counter* value to proceed with the OTP check. The *localServer* will then be in charge of determining whether it is a valid OTP code or not.

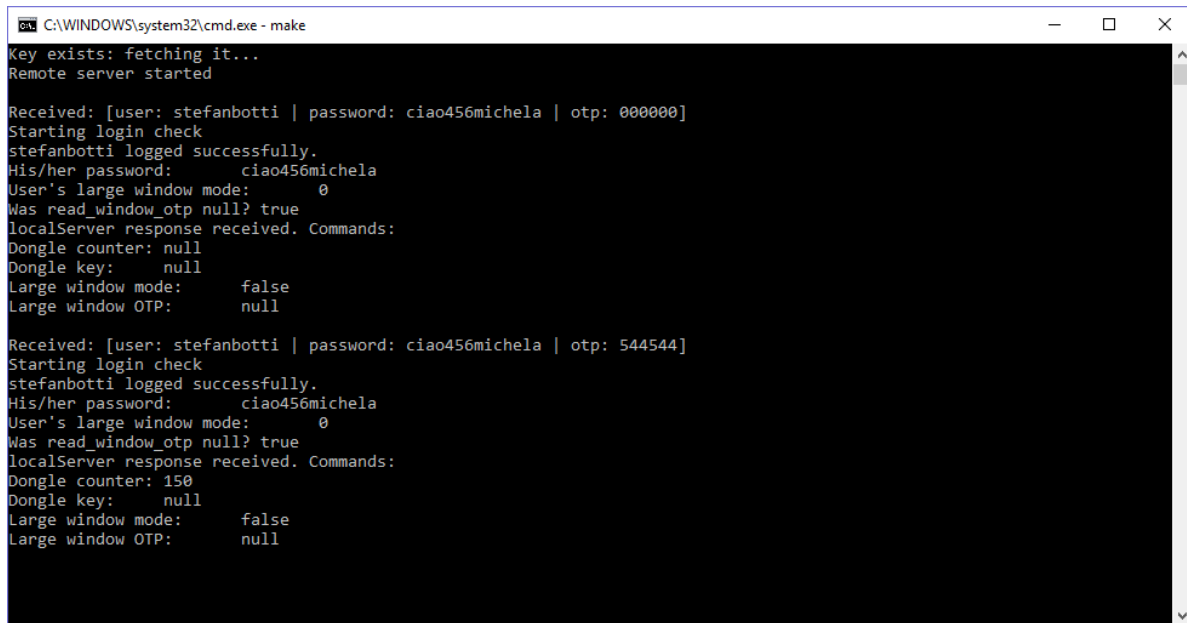
Replay attack involving M1 are avoided thanks to the freshness assumption guaranteed by the SSL handshake protocol as we said before.

5. Conclusions

The applications we developed (one for each entity) is available at:

- <https://github.com/marcomicera/OTPclient>
- <https://github.com/marcomicera/OTPlocalServer>
- <https://github.com/marcomicera/OTPrmoteServer>

The *remoteServer* application just outputs some useful informations on its terminal:

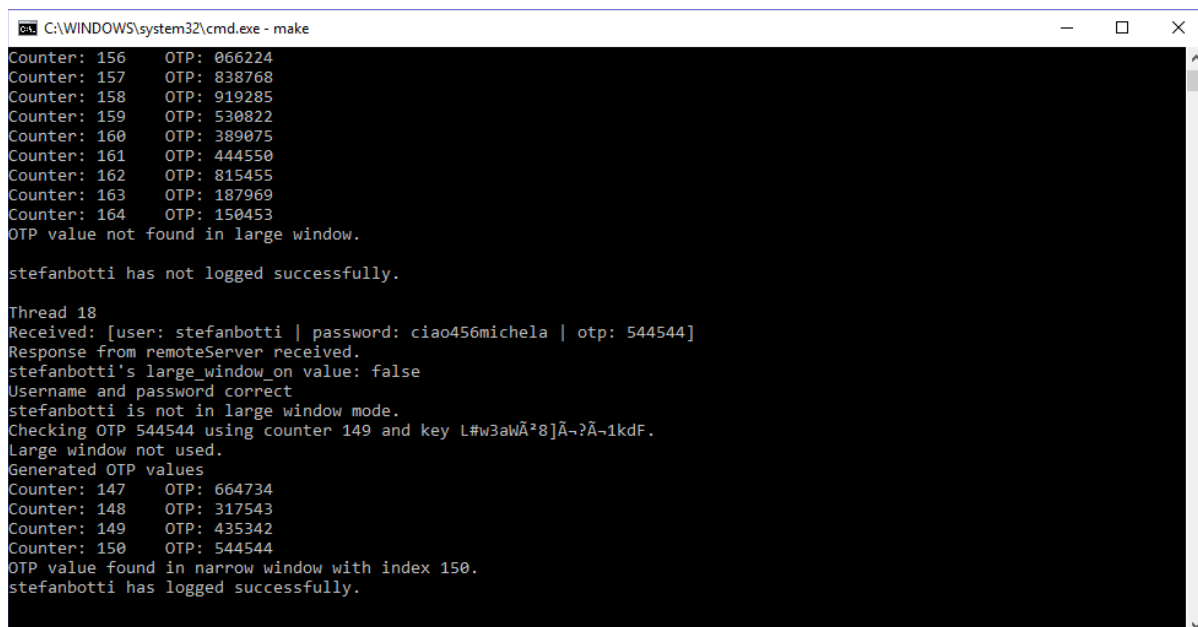


```
C:\WINDOWS\system32\cmd.exe - make
Key exists: fetching it...
Remote server started

Received: [user: stefanbotti | password: ciao456michela | otp: 000000]
Starting login check
stefanbotti logged successfully.
His/her password:      ciao456michela
User's large window mode: 0
Was read_window_otp null? true
localServer response received. Commands:
Dongle counter: null
Dongle key:      null
Large window mode:      false
Large window OTP:      null

Received: [user: stefanbotti | password: ciao456michela | otp: 544544]
Starting login check
stefanbotti logged successfully.
His/her password:      ciao456michela
User's large window mode: 0
Was read_window_otp null? true
localServer response received. Commands:
Dongle counter: 150
Dongle key:      null
Large window mode:      false
Large window OTP:      null
```

As the *localServer* does:



```
C:\WINDOWS\system32\cmd.exe - make
Counter: 156      OTP: 066224
Counter: 157      OTP: 838768
Counter: 158      OTP: 919285
Counter: 159      OTP: 530822
Counter: 160      OTP: 389075
Counter: 161      OTP: 444550
Counter: 162      OTP: 815455
Counter: 163      OTP: 187969
Counter: 164      OTP: 150453
OTP value not found in large window.

stefanbotti has not logged successfully.

Thread 18
Received: [user: stefanbotti | password: ciao456michela | otp: 544544]
Response from remoteServer received.
stefanbotti's large_window_on value: false
Username and password correct
stefanbotti is not in large window mode.
Checking OTP 544544 using counter 149 and key L#w3awÃ²8]Ã~?Ã-1kdF.
Large window not used.
Generated OTP values
Counter: 147      OTP: 664734
Counter: 148      OTP: 317543
Counter: 149      OTP: 435342
Counter: 150      OTP: 544544
OTP value found in narrow window with index 150.
stefanbotti has logged successfully.
```

A typical *remoteServer*'s encrypted database instance could be the following:

username	password	dongle_key	dongle_counter	large_window_on	large_window_otp
-SH6Ä□□3l+eSÅø	tr6ixT o²½□Ic	□ø:í+C7□Äk>!:□ÖM*¼6xÉR3□□PAeY	ÖëÜ(v/oa)c ½	0	NULL
:`DRÉ!;□□Á:²zø	# aEÁte²Á14/Ea²B	9;cs ³+2>9!Rfè-□ `dhs@3@	V:FèöA□öf1□□	0	NULL
e<VÁUoÚÝo-«Ä	(7±äí□-ÖÄ`sÄ□r	ÄÄwÜ×Vw` s-u`EÉ:üO×□éK^~äí:×□	í v□/vaø.¾	0	NULL
éí-tOfa□CZ=□□è	lTeOL`%MCV□□KQ	□□□k*nüäÄ!sh□G:²kdo-«ü lIs	□-äÄIÄ8e a\□8ø°	0	NULL
hÜ□ C6¼a«ñ 8uu...	±□Ü□□äü) `i.=UNB	r¼²»B `9PO`YOBaü!+²øÖo□Ön9` äö&	¼□□B¼F9ø»Xè`ö	0	NULL
o#K8-□`!□Öm	üÄ`² 4VJ□4	×ñw□□□Ö¿A□□«.ñ□□+²Lè□²aRÄ	8«b03k□□×:zX	0	NULL
Ö1ukl□□□Ö%□□□5	b□æZø×□Y(2âèk.NI+²øÖo□Ön9` äö&	ñÜÜH□□)É□ñÄX□HÜüY²èö!k<n²	□□²GKÖÉ!VÖV□□ö	0	NULL
lW□`Eiu7=²/Ü&I+²...	6Zz+ÄÄCd8ÖIb#bI	86O7<²IÜ□(RÄd«`F²EÉpc²ö` Ñ	65è□□8ÖI=806é	0	NULL
lwl□xü²!\$æö□Ö8«d...	»F l&` 3Ää«	F`ñ□²Äiü□IaUsDÖ uðJVJ8ä²ÜbÉëü	l6mwíu~□□cAvö	0	NULL

The *client* application has got a minimal UI to let the user insert his/her credentials. It also has a simulated OTP dongle interface in it:

