PV204 Project: KeePassXC with Java Card

Adam Janovský, Marie William Gabriel Konan, Matěj Plch

Introduction

This document describes design of all the parts of this project. Our work is also published on GitHub in our applet/documentation repository `https://github.com/Afforix/pv204_project` and our KeePassXC fork `https://github.com/Afforix/keepassxc`.

Java Card applet

The JavaCard applet is designed as follows. The password is stored in encrypted array in EEPROM. The encryption key is generated randomly and stored inside Key object. IV is also generated randomly. No-one (except the card itself) has the access to the key. The password is encrypted by 128-bit AES in CBC mode. The reason for encryption is that on the lecture it was noted that sensitive data should not be stored in arrays, but rather in special objects. Yet, there is no safe object for 128-byte long password. The unencrypted password is copied straight from/into incoming/outcoming APDU buffer. The length of the password is stored unencrypted, again, on EEPROM. The applet has only several practical methods, i.e. `verifyPin, setPin, setPassword, sendPassword`. The last three are allowed to run only when the user is authenticated. The methods are called based on the APDU commands, just as done in previous home assignments. We allow selection of applet only if the PIN is not blocked, moreover we reset authentication state on selection. No admin PIN is involved, therefore blocking the PIN (3 tries) results in blocking the applet permanently. Initially, the PIN is set to `0000`. The rest of the documentation is contained straight in the Java Card applet.

KeePassXC

In the KeePassXC we have added a button for requesting the password from Java Card. After successful selection of the applet user is asked for a PIN. The PIN dialog does not allow user to insert other characters than digits and the input length is limited to 4. If the PIN is correct, stored password is requested from the Java Card applet and filled in KeePassXC to a password field. The Java Card button is available when unlocking a database and when setting master password of a database.

Secure channel design

We have decided to use the Secure remote password protocol in version 6, slightly modified for use on Java Card, as precisely described in [1]. This protocol is based on [2] which provides mutually authenticated secure channel with possible encryption, with use of short password - PIN in our case. After the authentication, the Java Card version of the protocol provides both integrity and confidentiality as well as encapsulation of whole APDUs (to provide integrity of the APDU header). Further on, we shortly describe how the protocol works in our enviroment. The card poses as server and the PC app poses as client in the protocol. The initial phase of the protocol has to be done in the secure environment. First, group in which we compute is established, i.e. the modulus n is chosen and the generator g is computed. Further on, the PC app picks a random short salt s and asks user for a PIN P. Then the PC app computes $x = H(s, P)$, where H stands for arbitrary secure hash function. Further $v = g^x$ is computed and (v, s) is stored securely on the javaCard. The mutual authentication and shared secret is then achieved with the following scheme

$$
\begin{array}{lcl}
\text{pcApp} & & \text{Card} \\
A = g^A & \xrightarrow{A} & B = kv + g^b \\
u = H(A,B) & \xleftarrow{B,s} & u = H(A,B) \\
x = H(s,P) & & \\
S = (B - kg^x)^{a+ux} & & S = (Av^u)^b \\
K = H(S) & & K = H(S) \\
M_1 = H(u,S) & \xrightarrow{M_1} & (\text{verify } M_1) \\
(\text{verify } M_2) & \xrightarrow{M_2} & M_2 = H(u, M_1, S).
\end{array}
$$

where all computations are performed modulo n and with respect to the following notation

g	. . .the group generator
s	. . .Random pcApp's salt for the password
a, b	. . .Ephemeral private keys, generated randomly and not publicly revealed
A, B	. . .Corresponding public keys
k	. . .Constant multiplier, computed from the has of the modulo and concatenated with g
x	. . .private key derived from the password and salt
v	. . .The password verifier calculated from g^x
u	. . .Random scrambling parameter, publicly revealed
$H()$	. . .One-way hash function
K	. . .Computed session key

The actual authentication happens in the last two steps and the resulting session secret is K. Upon completion of the protocol, secure communication takes place. It has to be noted that if any error during the protocol happens, whole process has to be stopped and repeated. For the exact thread model, see [1].

Secure channel implementation

Unfortunately there was not enough time and manpower to design and implement the secure channel protocol, so it is not used. We have at least found possible libraries to use/get inspired. First of all, the applet which implements the protocol is part of the article [1] and is fully available on github [3]. The corresponding C++ library (implementing the original SRP-6 protocol) is available from [4].

References

[1] HÖLZL, Michael, Endalkachew ASNAKE, Rene MAYRHOFER a ndMichael ROLAND. A password-authenticated secure channel for App to Java Card applet communication. *International Journal of Pervasive Computing and Communications* [online]. 2015, **11**(4), 374-397 [cit. 2017-05-04]. DOI: 10.1108/IJPCC-09-2015-0032. ISSN 1742-7371. Available from: `http://www.emeraldinsight.com/doi/10.1108/IJPCC-09-2015-0032`

[2] WU, Thomas D., et al. *The Secure Remote Password Protocol.* In: NDSS. 1998. p. 97-111.

[3] HÖLZL, Michael, Endalkachew ASNAKE, Rene MAYRHOFER and Michael ROLAND. *Java Card applet for SRP-6a password-authenticated secure channel to secure elements/smartcards* [online]. Available from: `https://github.com/mobilesec/secure-channel-srp6a-applet`.

[4] SLECHTA, Pavel. *C++ library implementing The Stanford Secure Remote Password Protocol - SRP (SRP6a)* [online]. Available from: `https://github.com/slechta/DragonSRP`.