Security Technologies

Project assignment № 2

Štěpánka Gennertová 451432, Marek Šanta 433772, Michal Pavúk 487550 *2020-04-23*

In this report we present our implementation of secure channel communication protocol between the client application and the applet. It is based on a standardised J-PAKE. In addition to the standard Diffie-Hellman key exchange, whose functionality is contained within the protocol, J-PAKE provides mutual authentication of both parties.
The Password Authenticated Key Exchange by Juggling (*J-PAKE*) is a password-authenticated key agreement protocol, that allows two parties to establish private and authenticated communication, based on their shared low-entropy password (in our case PIN).
Theoretical description of J-PAKE protocol defines a completely symmetrical two-round key exchange. Such an exchange is impractical in scenarios where the communication is initiated by one of the parties. Fortunately the authors of the protocol recognize this issue and provide a three-pass variant of the protocol. However due to technical limitations of the JavaCard platform, namely missing support for extended APDUs in many environments (including JCardSim) we split the data that would be transferred in a single response into 2 responses. This has no effect on the overall security of the protocol.

Protocol

For the purposes of this description we refer to the two communicating parties as an application (running on host computer) and an applet (running on the card).
The application initiates the communication with the card by selecting the installed applet for secure data transmission. It then sends the `INIT` command (INS: 01) with the required data.
$E(F_p)$ is an elliptic curve over a finite field F_p where p is a sufficiently large prime of order n. Our implementation uses the prime256v1 curve. We use s to denote the shared secret (PIN).

Zero-knowledge proof

Authentication of this protocol relies on a well standardised Schnorr zero-knowledge proof scheme. The proof allows one to prove the knowledge of a discrete logarithm without leaking any information about its value. For brevity description of this scheme is omitted, but can be found in RFC8235. The main takeaway is that the proof of knowledge of an exponent a is comprised of:

- UserID - a string identifying a communicating party
- $V = G^v \mod p$ - G is a point on an elliptic curve (same curve as used for J-PAKE), v is a randomly selected value from $\langle 1, n-1 \rangle$
- $r = v - a \cdot c \mod q$ - a is a value for which the proof is computed where c is a challenge (a hash) computed from the point V, the curve G, the value a and the user identity.

The parties can verify the proofs by checking if their private knowledge of the exponent lies on the same curve as the proof says. The secret value is never transferred.

Phase 1

The application generates ephemeral private keys x_1 and x_2 from $\langle 1, n-1 \rangle$. It then selects the applet and sends the `INIT` APDU.
The data contained within the `INIT` command are:

- $G_1 = G^{x_1} \mod p$ and $G_2 = G^{x_2} \mod p$
- V_{x_1}, V_{x_2}, r_{x_1} and r_{x_2} are components for zero-knowledge proofs for x_1 and x_2 respectively

Phase 2

The applet verifies the ZKPs for x_1 and x_2. If the proofs are **NOT** valid the applet responds with error code and the session initiation is terminated. Otherwise it will, similarly to the application, randomly select an ephemeral private keys x_3 and x_4 from $\langle 1, n-1 \rangle$. It then responds with:

- $G_3 = G^{x_3} \mod p$, $G_4 = G^{x_4} \mod p$
- V_{x_3}, V_{x_4}, r_{x_3} and r_{x_4} as ZKPs for x_3 and x_4

Phase 3

The application verifies ZKPs for x_3 and x_4 in case they are **NOT** valid, the application terminates the session by deselecting the applet (and clearing card's RAM in the process). Otherwise the application will send the `FINISH` command, containing:

- $A = (G_1 \cdot G_2 \cdot G_4)^{x_2 \cdot s} \mod p$
- A ZKP for $x_2 \cdot s$

Phase 4

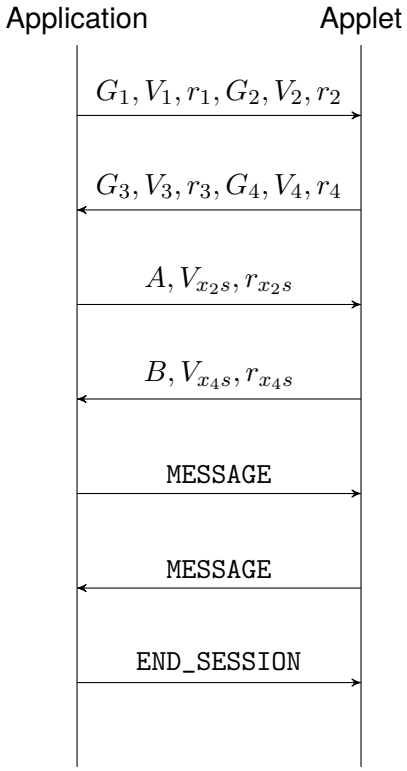
Upon receiving the second APDU command, the applet verifies the proof for $x_2 \cdot s$. If the proof is valid, it responds with its generator B:

- $B = (G_1 \cdot G_2 \cdot G_3)^{x_4 \cdot s} \mod p$
- $V_{x_4 \cdot s}$ and $r_{x_4 \cdot s}$ as a ZKP for $x_4 \cdot s$

Now both parties will compute the common key material as follows:

- Application: $K_a = (B - (g_4 \cdot [x_2 \cdot s])) \cdot x_2$
- Applet: $K_b = (A - (g_2 \cdot [x_4 \cdot s])) \cdot x_4$

At this point, the following equality holds true: $K_a = K_b = G \cdot [(x_1 + x_3) \cdot (x_2 \cdot x_4 \cdot s)]$ and both parties will use key derivation function to derive a common session key k. All subsequent communication is then encrypted using the derived session key k.



Commands, responses and messages

Message	CLA / Code	INS	P1	P2	L_c	Data field	L_e
SELECT	00	04	04	00		AID	
INIT J-Pake 1	b0	01	00	00	C4	$G_1, V_1, r_1, G_2, V_2, r_2$	C4
I-RESPONSE J-Pake 2	90 00	-	-	-	-	$G_3, V_3, r_3, G_4, V_4, r_4$	
FINISH J-Pake 3	b0	02	00	00	63	$A, Vx2s, rx2s$	63
F-RESPONSE J-Pake 4	90 00	-	-	-	-	B, V_{x_4s}, r_{x_4s}	
END_SESSION	b0	03	00	00	-	-	-
MESSAGE	b0	04	00	00	variable	encrypted message data	

The `MESSAGE` APDUs contain data encrypted with the derived session key k and have a following format:

Offset	Size	Value	Description
0	1	counter 00-FF	
1	2	checksum	ALG_ISO3309_CRC16
3	1	length 00-FD	remaining length of the data
4	length 00-FD	any	message contents
4+length	240-4-length	padding	random bytes

Implementation

After establishing the channel all exchanged messages are symmetrically encrypted using 128-bit AES cipher in CBC mode with random padding with the derived session key. Channel initialization is an atomic operation. Should one of the steps on either side fail, the whole handshake is aborted and needs to be started anew.

The purpose of the counter is to detect and prevent dropping, and replaying of the encrypted messages. The counter is valid for the duration of the session and starts at the value of 0. It increments with each successfully received command (as indicated by the response code). Should either party receive a message with the counter value lower than its internal session counter, the session is immediately terminated. On the other hand, should the counter be larger than the internal one, some packets were dropped by the adversary and the session is terminated.

While the attacker has knowledge of the counter value, they cannot modify the value of the counter due to the presence of encryption. The semantics regarding the checksum mismatch are set by the application utilizing the channel.

The `END_SESSION` message does not require any kind of verification. While we are aware that this simplification allows the attacker to finish a session prematurely, we decide to leave it as-is, since the attacker can simply cut the connection entirely.