PV204 Security Technologies: jpass-smartcard code review

Adam Janovský, Marie William Gabriel Konan, Matěj Plch

Desktop application

As described by the authors in the documentation, the desktop application can be compiled to use simulated or real smart card. When using simulated card the application worked, we were able to create a database with passwords, store it encrypted on a disk, load it from disk and successfully decrypt it. We have also traced the program in debugger and confirmed that the program processes the database through the smart card.

We have also tried to use real smart card, but we have encountered multiple compilation errors. We have contacted the authors and they admitted that they have not tested their applet on a real card. Together we have successfully built the applet for a real card, but the applet have not worked. Reason for that is not known, the applet works in a simulator, but this is not the first time we see such behavior, so we have no illusions about reliability of smart cards. The other team didn't have time to provide us with working applet on a real card, so we have not done more investigation of this problem.

Static code analysis

We have used FindBugs to analyze the project for issues in the code. The original project already contained 28 issues, but another 26 issues were introduced. The findings were most probably not critical, for example unused methods or dead stores. One issue was interesting, it complained about unused variable Applet.SecureChannel.keyMAC, which could indicate that there is something wrong with an integrity of messages. However, authors clearly stated in their documentation that implementation of the secure channel is incomplete and broken.

Java Card applet and secure channel design

Symmetric key derivation

Though not implemented, the authors propose to truncate shared secret obtained by DH key exchange to 256 bits and use it as a symmetric key for AES. Practically, it would be probably safe way to go. Yet, as the shared secret is basically member of group, the entropy within those 256 bits is less then 256 bits. In exact, some bits might be predictable due to algebraical structure of the number. How many bits are leaked depends heavily on choice of primes for the underlying group (safe primes leak less, results in like 247 bits of entropy), etc, etc... Moreover, there are standards that command to use proper key derivation function. On assumption that hash function is properly random, simple hash of the key would be sufficient.

PIN validation

- Selection of the applet is allowed with blocked PIN
- The PIN validated flag is not reset on select nor deselect. The attacker could wait for the applet to be deselected (card not removed), then simply select it again by himself and be authenticated. Then he could establish shared secret and obtain decrypted database

Some (bad) examples of code follow. For instance, in method `processDecrypted` we see the following code:

```
if(ins == INS_VERIFYPIN) {
    if((short)m_pin.getTriesRemaining() > (short)0) {
        VerifyPIN(apdu);
    } else {
        ISOException.throwIt(SW_BAD_PIN);
    }
}
```

The first condition is rendundand, as the PIN validation process would return false anyway. Moreover, in the `VerifyPIN` method we see the code

```
if (m_pin.check(apdubuf, ISO7816.OFFSET_CDATA, (byte) dataLen)
   == false) {
        ISOException.throwIt(SW_BAD_PIN);
    } else {
        m_pin.reset();
        m_pin.check(apdubuf, ISO7816.OFFSET_CDATA, (byte)
           dataLen);
    }
```

where the PIN is checked twice. If it passes, why to check it again? Generally, both mistakes do not probably create any space for the attack, but the style of the code allows exotic constructions which can be misused, as mentioned later.

Padding oracle

Introduction

- Padding oracle already covered in PV079.
- If we force the Java Card applet to tell us about correctness of padding during decryption of the database, we are able (in AES-CBC mode) to decrypt the database.
- Remains impractical under our constraints.
- The Java Card applet actually leaks info about correct/incorrect padding. The guessed plaintext bytes are sadly encrypted to unknown values, hence this cannot be directly misused.
- Most of the findings in the code review are obvious - authors probably know that integrity must be included and so on and so on. This is different, shows how certain parts of the system can be misused.

Constraints

- No freshness, hence the ability of replay attacks.
- Ability to inject arbitrary APDU commands into the communication.
- Ability to stop traffic going to the desktop application.
- The attacker must be able to send encrypted messages (this is the only thing that keeps it impractical).

The attack

- In the applet, there are three methods to decrypt the database. Only one method – `CBC_BULK_Finish` – concerns the padding.
- The method `CBC_BULK_Finish` is designed to decrypt only the last block of the data. Yet, can be used to decrypt any blocks of the data multiple times.
- The method `CBC_BULK_Finish` expects only one block of the data, i.e. 16 bytes at most (but can be forced to accept more). It decrypts those bytes and then checks the padding. If the padding is correct, it returns index where padding starts, i.e. $[0, \ldots, 15]$. Moreover, the data to decrypt are fed to the cipher object from `m_last_block` variable, not from the APDU buffer.
- The variable `m_last_block` should only have length of 16 bytes, yet, it is the byte array of 260 bytes. And data from apduBuffer can be fed into this variable by using the method `CBC_BULK_PROCESS` in encryption mode.
- If the padding is incorrect, the method `CBC_BULK_Finish` returns the length of encrypted data. Since it expects the 16 bytes, this should not be distinguishable from the index where padding starts $[0, \ldots, 16]$.
- Yet, if we send more blocks there, for instance 2 blocks, 32 bytes, then if the padding is incorrect, it returns value 32 - points to incorrect padding.
- Hence the padding oracle occurs.
- The response from padding oracle is sent encrypted to the attacker. But, the length of the APDU data perfectly corresponds to the returned value - if it is 32, padding incorrect. If the lengths is $[0, \ldots, 16]$, the padding is correct.
- So there is no need to decrypt the response, only check for its length.

Simulation of the attack

1. The attacker gains the access to the encrypted database — $e_k(db)$, which is stored on the PC.
2. The attacker waits for the desktop application to get authenticated in order to be able to inject APDU commands that require authentication.

3. Now, the method `CBC_BULK_Finish` takes `m_last_block_length` bytes, which is initially set to 0 and using `MODE_DECRYPT` can be only set to value 16 (the length of the block). Yet, if we use `MODE_ENCRYPT` and send arbitrary long buffer, it will set the variable `m_last_block_length` to any value the attacker wishes, i.e. to 32. After this, the method `CBC_BULK_Finish` will accept buffers of length 32 bytes.

4. The attacker must encrypt the already encrypted database with the secure channel key, i.e. $e_s(e_k(db))$ and start sending questions to the padding oracle.

5. The question to padding oracle cannot be send directly in apdu buffer, but must be fed to `m_last_block` first (from where is fed to `CBC_BULK_Finish`. This can be done by calling `CBC_BULK_PROCESS` with `MODE_ENCRYPT` option.

6. The attacker reads the length of padding oracle response and based on that he decrypts the database .

Conclusions on padding oracle

The attack is obscure, impractical, but shows that many parts of the system can be misused (calling encryption mode to feed variable, etc...) and that one method actually leaks the padding result.

Minor problems with applet

- Instruction codes are sent unencrypted.