[PV204] Project Report - SatoChipApplet

Martin Knotek, Lenka Svetlovská, Jiří Týma

1 Introduction

SatoChipApplet is open source javacard applet implementing a secure hardware wallet that can be used to safely store and spend Bitcoins and other digital currencies. Based on javacard technologies, it can be combined with a Yubikey Neo to provide an easy and secure user experience.

2 Analysis

2.1 Overall

Pros
+ sensitive receive and temporary buffers cleared on deselect
+ good comments on some methods

Cons
- unclear and complicated API on others
- not all allocations are done in constructor, but in setup() instead, we will definitely change that
- difficult and unclear setup method, we will try to refactor
- very long methods, we might refactor slightly and split those to multiple shorter functions
- multiple functions and parts of code commented out without explanation, we might remove them altogether

2.2 Functionality

The applet provides the following main functionality
- generate keypair
- generate symmetric key
- de/encrypt data - DES, AES, RSA
- derive public from private key
- signature of short,long message and transaction
- change/create/unblock/verify/list pins
- BIP32 import key
- BIP32 set/get extended key

2.3 APDU format

Apdu buffer contains instruction, class, and in the data there is additional information bout the operation that is to be executed - its data, algorithm to use, en/de-cryption, additional operands. The additional info above the standard APDU packet is dependent on the function which is being called.

For instance, in function *GenerateKeyPairRSA()* the buffer data at offset CDATA+1 specifies some options to use when generation keypair.

2.4 Sensitive data

What sensitive values are protected:

Sensitive values, such as *Key*, *Cipher*, *Signature* adn *OwnerPIN* objects are set as private.

They are, however, created through calling new, such as

```
keys = new Key[MAX_NUM_KEYS];
```

inside function Setup. Although the function *setup()* is intended to be run only once, through setting a boolean variable setupDone to true, creating these variables outside of constructor might have severe consequences.

The initial running of this function is ensured through condition in the process function. If the *setupDone* variable isn't set to true value, or the instruction itself is not *INS_SETUP*, an exception is thrown. Another condition follows, that is if the setup has been already done and the instruction is *INS_SETUP*, again, an exception is thrown.

```
        if (!setupDone && (ins != INS_SETUP))
            ISOException.throwIt(SW_SETUP_NOT_DONE);

        if (setupDone && (ins == INS_SETUP))
ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
```

In constructor, the initial value of PIN which is hardcoded is set with

```
pins[0] = new OwnerPIN((byte) 3, (byte) PIN_INIT_VALUE.length);
pins[0].update(PIN_INIT_VALUE, (short) 0, (byte) PIN_INIT_VALUE.length);
```

The value of pin is again later changed in function Setup.

In the *setup()* function the *recvBuffer* is allocated on *RAM*, it this call fails, the buffer will be allocated on *EEPROM* and as the comment says, it might destroy the memory after a number of writes to it. The same situation is with the *tmpBuffer*. It would be better if we moved most of *setup()* into constructor and if all memory cannot be allocated, we would not let the app run.

2.5 Implementation details

RandomData generator is constructed upon first usage, this might add execution time, we will construct the generator in the c'tor to avoid time overhead and to make the allocation similar to the allocation of other resources.

In function *signMessage()*, the switch statement has no default case, which at least is a bad programming habit, let alone a possible source of unexpected behavior.

In function *getData()*, if the APDU packet is extended, its length is not checked or restricted, thus allowing for a very large packet which might not fit in the memory.

In the applet there is *ALG_AES_BLOCK_128_ECB_NOPAD* algorithm version used, which might not be the best choice for cryptographic use because it does not hide data patterns very well and is "only" 128b long.

In *MemoryManager*, a large piece of memory is allocated using *new*, without checking its size or return value. This should not happen, we should check for any memory allocation return value or at least set some maximum size.

BigInteger class provides multiple functions for handling BigInts, most of these functions use very low level bit shifting and bitwise logical operations and are not tested at all, meaning might contain an error which is difficult to spot but might have consequances.

2.6 Cryptographic algorithms and protocols

Almost all allocations are provided in `setup()` method by algorithms:

- `ALG_EC_SVDP_DH_PLAIN` to instance `KeyAgreement`,
- `ALG_ECDSA_SHA_256` to instance `Signature`,
- `ALG_AES_BLOCK_128_ECB_NOPAD` to instance `Cipher`,
- BIP32 keys are type AES of the length 128 or 256 bits or
- type ECDSA curve (`TYPE_EC_FP_PUBLIC`) of the length 256 bits,

- `ALG_SHA_256` to message signing and
- `HMAC_SHA160` to transaction signing.

The method `ComputeCrypt()` performs encryption/decryption on provided data and uses various algorithms. The algorithm depends on key type - in case the key is **RSA** (public/private or CRT private), the method uses `RSA_NOPAD` or `RSA_PKCS1` to encryption/decryption. If key is **DES** type it uses `DES_EBC(CBC)_NOPAD` and if key is **AES** type is uses 128 bits `AES_CBC(ECB)_NOPAD`. The method also uses `ALG_SECURE_RANDOM` to create random data.

- The method to compute signature uses `ALG_RSA_MD5_PKCS1`.
- The project contains own implementations of `HMAC-SHA160` and `HMAC-SHA512`.
- The class `secp256k1` refers to the parameters of the ECDSA curve used in Bitcoin.

2.7 Relevant attacker model, possible attacks

The software implementation of HMAC-SHA512 could have an potential impact on the physical security against **side-channel attacks** (for attackers with physical access to the chip).

Based on details stated in 2.5, the function *SignMessage()* might be an attacker's target when trying to manipulate or somehow change the input for the function resulting in unexpected behavior.

Taking the *setup()* function implementation in consideration, it might be possible to tweak the app into thinking the setup was not completed even if it was, calling the setup when not appropriate and thus causing data erase.

As stated in 2.4, if there is not enough RAM the EEPROM is used. If the attacker used the whole RAM, which is not too large, thus using the EEPROM and sending a large amount of APDU packets, while all of them are stored into the EEPROM might lead to problems of running out of memory or damaging the cells. The chance is even higher when the tmpBuffer is allocated in the same manner.

Maybe the simplest attack would be to send a very long extended APDU packet. According to 2.5, the whole packet is stored into a giant array, which very much sounds like a potential security hole which is extremely easy to violate.

The encryption/decryption methods use algorithms with no pad so it supposed to be protected against **padding oracle attacks**.

Try to convince setup() to allocated more memory than actually possible.

3 Conclusion

We have performed a security analysis on the SatoChipApplet and presented the results in this paper. In 2.1 we mentioned a few code style related notes, which we will try to address. In 2.5 and 2.4 we mentioned more issues, these are connected somehow with the security of the applet. These will be our priority as well, as we will try to improve the applet in our best effort.

We also mentioned algorithms used by our applet as well as possible attacks against the applet. When describing any issue, we always tried to suggest a possible solution to that issue.