JavaCard Project – Information Sheet
Hardware Security

Greg Alpár and Erik Poll,
Digital Security, ICIS, Radboud University

Updated January 2021

This document collects hints and tips for your JavaCard Project in the Hardware Security course, especially when it comes to documenting your design. It is based on the experiences of previous years when we taught the course. Read it carefully and check it regularly during the project, and use it as a checklist for the project reports before handing them in.

In a nutshell, you have to form a group, choose an application, design a system, write a document about the design, implement it, and write a second, final document. The final report is an update the first report (for instance updated in case any details of the protocols had to be tweaked given the constraints of the smartcard) extended with some documentation of choices made in the implementation phase. Lots more details below.

Contents

1 About the Project

The goal of the JavaCard smart-card project is to build a smart-card system by which you can

- experience the whole process from high-level design, given security requirements and assumptions, down to actual code on real hardware;
- appreciate complexity and interplay of
  - design considerations & constraints,
  - key management & distribution,
  - protocols,
  - silly hardware limitations, weird crypto padding, …
  - and the practicalities of getting all this working.

There are four options for the project application you have to implement:

1. electronic purse;
2. loyalty card;
3. petrol rationing;
4. car rental.

The only design constraints are

- you must use a JavaCard contact smart card – a JavaCard card provide you, to be precise.
- you must store some information other than just key material on this card (e.g. credits, amounts, logs, …).[1]

The focus on the project is on the smart card and terminals it talks too. A real system would have to include some back-end infrastructure, but there is no reason to spent a lot of effort into building that: simply make as little of the back-end system as you can get away with.

1.1 The deliverables for your project

The rest of this document gives more detail about the three deliverables you have to hand in:

- the initial high-level design document (discussed in Section 2),
- the source code of your implementation, and
- the final project report (which is essentially an update of 1).

[1]This is to make the application interesting. Otherwise, your solution could be just be a standard authentication solution, if you assume all terminals are online so that any additional information or functionality can be provided by a central back-end. Note that in our world where everything is increasingly online, this may well be a natural design solution.

Ultimately, the goal is that your documents should be such that security experts (like your fellow students or the people grading this course) can quickly convince themselves of the soundness of your design, and that what you implemented is secure. As part of this, you have to describe what it means for your application to be secure, and against which attacks it is meant to be secure. It should be clear that the security requirements address all the threats to your application given the attacker model, that your protocol design and associated key distribution ensure these security requirements, and finally, that the code correctly implements all this. There is no need to include any "sales pitch" advocating the use of smart cards as e-purse, car key, loyalty card, etc.

In short, your report should explain *how* your solution works and *why* it is secure, and also *what* means for it to be secure. The challenge here is to describe the system at various level of abstractions, namely

- the abstract level of *functional and security requirements*, motivated by a clearly defined *attacker/threat model*,
- the more concrete level of *key and certificate distribution* and the (still abstract) *security protocols*,
- the concrete level of the *source code*,

while making the relations between these levels as clear as possible.

Unfortunately, much of the documentation of real-world systems you will come across in your future professional life will not do this. Much of such documentation will consists of hundreds of pages presenting all sorts of details, without giving any clue about what security requirements all this is meant to guarantee, without any rationale explaining why the system has been designed the way it has been, and without any argumentation about why all this is secure.

1.2 Checklist before for handing in any stuff

- Run a spell checker over documents.
- Make sure pages and chapters/sections are numbered.
- Make sure documents are `pdf` – no `doc`.
- For documents handed in, make sure your group number is both in the filename and in the document itself.

2 First deliverable: High-Level Design Document

The goal is to provide a **concise and clear document** outlining and motivating your design, including stakeholders, assets, security requirements, design decisions, attacker model, and assumptions. The design should also contain details like PIN codes, key distribution, and (the goals of) any security protocols used. The document should be 8 pages max., but try to use as few as possible.

The sections below provide a more detailed description of things to include in your design document, roughly in the order to include them, and a very rough indication of the length. But feel free to diverge from this order and the indicated lengths if you think this makes for a better document.

2.1 Use cases (1 page or less?)

A brief description of the use cases, You can also use this to fix some terminology, e.g., for the different terminals involved.

Don't forget that in addition to the obvious terminal applications that are needed for each project (e.g., cash register, petrol pump, etc.), you will also need functionality to 'personalise' the cards. This typically involves uploading/generating keys, setting card numbers, etc. For this you will need a personalisation terminal, which is used by the card issuer prior to handing out cards to the card holders.

Initialisation is the start of the lifecycle of a smart card. Also think about a possible end: e.g. do cards have to blocked or decommissioned at some stage? How do you deal with cards that are reported lost or stolen?

2.2 Attacker model (half a page?)

An attacker model describes the *capabilities* of adversaries that you try to protect the system from (i.e. what can attackers do? To which parts of the system (incl. communications links) do they have access?) and often also the *goals* of these attackers.

When it comes to the capabilities of the attackers, these are effectively already given

- Your attacker model should include active MITM (Man-in-the-Middle) attacks on all smartcard-terminal interactions. Note that this goes a bit beyond what is considered as the Dolev-Yao attacker model, as a card tear – ripping the card from the terminal and – not only interrupts communication but also reset the card, wiping all its RAM You may not rely on terminals 'swallowing' cards to make card tear attacks impossible.

- Your attacker model can assume that the card is tamper-resistant, so that no physical attacks are possible to read out or change memory contents. (In other words, you may assume that the card is secure in the sense that confidentiality and integrity of software & data on the card is guaranteed, and cannot be observed or altered by side-channel or physical attacks.) However, your design must ensure that one broken card will not bring down the entire system, which would happen if for example all cards contain the same private key.

You will still have to figure out the *goals* or motivations of the attackers. These goals are typically in one-to-one correspondence with your security objectives, so you might end up repeating yourself.

Still, it can be useful to think of goals of an attacker, if only to make sure you do not overlook some security requirements. It can be useful to consider some of the insiders as a special category of attacker, e.g. malicious shops or petrol pump owners, as sometimes these parties have special goals or special access. E.g. a petrol pump owner might be interested in selling petrol without collecting rations, to effectively sell on the black market. A shop keeper may want to claim money from a bank for fake payments that did not happen.

There will always be some things you have to exclude from you attacker model, i.e. some components or people for which you have to assume that they cannot be compromised. These components are then in the Trusted Computing Base (TCB) for some specific security property. State such assumptions *explicitly.* Also, it may be useful to explicitly state assumptions on what an attacker can *not* do, rather than just saying what an attacker can do.

Tips and pitfalls

- Don't confuse *threats* and *attacks.* In security, the convention is:
  - A threat is something bad that may happen.
  - An attack is a way an attacker may try to make something bad happen.

  So a Man-in-the-Middle (MITM) is not a threat, but an attack. An attacker getting free rations, money, loyalty points, etc. is a threat.

- Dolev–Yao is a well-known attacker model, but be aware that when it comes to attacker models, there is more to life than Dolev–Yao. Dolev–Yao is very narrow in focus: it only considers the security of some of the communication channel (here, between smart card and terminal), whereas in real life that are there are some attacks outside the scope of Dolev–Yao, namely at either end of the communication pipeline, e.g., a smart card being stolen, people claiming to have lost their cards, people interrupting transactions by card tears, corrupt employees.

- Make all important assumptions you are making – esp. about attacks or attackers that are considered 'out of scope' – *explicit.* This may include assumptions on various any component of the system (e.g., petrol pumps, point-of-sale terminals, cars, the odometers in cars, the presence of reliable clock for time stamps, ...

2.3 Security Requirements

Ideally, you want to provide a numbered list of security requirements.

Security people always talk about CIA – *confidentiality*, *integrity*, and *availability* – but thinking in terms of CIA, or trying to classify security requirements in these three categories, is typically *NOT* a useful approach. Instead of thinking about the rather abstract notions of confidentiality and integrity, it is much better to begin brainstorming about security requirements for this project in terms in the more concrete notions of authentication and non-repudiation. (These two properties can be considered as variants of integrity, of course.)

The more obvious security requirements are often about *preventing* security problems, but there may be sensible but less obvious security requirements about *detecting* of problems or *reacting* to problems, for example having logs to detect problems or simply to do forensics in case a problem has occurred.

Numbering or naming the security requirements (e.g. SR1, SR2, . . .) is convenient to make it easy to refer to them in other parts of the document and more generally to keep track of things.

Tips and pitfalls

- Beware of possible confusion between, and the relation between:
  - *'entity' authentication*: authenticating that some entity – someone or something – is who they says they is. Often you want mutual authentication instead of just one-way authentication.
  - *'data' authentication*: authenticating that some data as really been sent by some party, and has not been altered.

    When we want some message to be authenticated, we often also want the message to be fresh, meaning that it is not a replay of a message sent earlier.
- When describing security objectives, don't just say 'authentication' or 'non-repudiation', but always say
  - authentication of WHOM to WHOM;
  - non-repudiation of WHAT by WHOM to WHOM.

  Rather than talking about 'non-repudiation of X by A to B', it is often clearer to say something like 'A cannot deny having done X to B' or 'B can prove to C that A has done X'. Note that non-repudation properties can involve three (or more) parties.
- Clearly distinguish between integrity of an individual message and integrity of a whole transaction consisting of a sequence of messages (aka session or transaction integrity).
- Don't forget about non-repudiation: does the terminal (or card) need some verifiable proof of a transaction later? Eg., should petrol pumps, e-purse payment terminals, etc. want to obtain some verifiable proof that they collected rations, money, etc.
- It may be useful to think about **abuse cases** when thinking about the attacker model or coming up with security requirements. Often, abuse cases are in one-to-one correspondence with security requirements, in that a security requirement typically tries to prevent some form of abuse.
- When coming up with security requirements, you often immediately start thinking of ways in which to guarantee this. But in your report make sure there is a clear distinction between WHAT you want to guarantee and HOW you guarantee it, so that you separate security requirements and design decisions. The easiest way to do this is that the section listing the security requirements does not mention things like passwords or cryptographic keys, let alone any details of security protocols.

  Conversely, when designing a system or coming up with security protocols, you sometimes notice that your forget some (obvious) security requirement, and then you design your protocol to take care of this. Don't leave such requirements implicit, but go back and explicitly add them to the list of security requirements.

2.4 Key distribution and protocols (this will run into several pages)

Describe the high level design of your system, including details such as use of passwords or PIN codes, the key distribution, and the security protocols, as explained in more detail below.

2.4.1 Key and certificate distribution

Before you give the details of any security protocols give a clear overview (e.g. in a table) of the key and certificate distribution (i.e. who has which keys and certificates) and also the purpose of keys and the meaning of certificates.

Explain any notation you use for signing, MAC-ing, or encryption (for which you can e.g. use $\{\ldots\}_K$, $MAC_K(\ldots)$, etc.)

If you use certificates, where you sign (the hash of) some public key with a master key to prove the public key is genuine, think about some other information to include in the hash value that you sign. For instance, you'll typically want to include the ID of the owner of this key in the hash. Or, if you use certificates for say both terminals and cards, you should include some boolean in the hash to indicate whether the certificate is for a terminal or a card. If there are different type of terminals, instead of a boolean you could use some byte to distinguish the different types of terminals. (All this limits the impact if any private key ever leaks. You typically have to assume keys do not leak for the security requirements to be met, but still, it is nice to reduce the impact in the case that does happen.) Using X.509 certificates is a terrible idea, as these are way too long and verbose to be practical for smartcard communication, and the specification of X.509 is a horrible mess, but it might provide some inspiration. (There is a specification for Card Verifiable Certificates (CVCs), which are used in e-passport applications, as a certificate format that is better suited to the constraints of smartcards, but even these CVCs are much more complex than what you will need.)

2.4.2 Security protocols

Give a description of all the protocols used as Message Sequence Chart (MSC) or in Alice-Bob style, i.e.

```
1. A -> B: {bla}key
2. B -> A: ...
3. A decreases its balance by n
4. A-> B: ...
```

Numbering the steps is useful. In the implementation every message will have to include some header byte (the so-called ISO7816 instruction byte) to identify which message type it is; it can be useful to already include such unique identifiers in each of these messages in this MSC or Alice-Bob presentation.

Do not only specify the communication steps, but also security-critical operations such as

- increasing or decreasing balances,
- writing some data to log files,

- incrementing counters,
- displaying message ”Payment Done” on cash register
- starting/stopping petrol pumps,
- etc.

as the precise order in these happen may matter: it may be important that you first increment some value and only then send a message rather than the other way around.

Before describing security protocols in all their detail, give concise descriptions of their goals, i.e. of WHAT they are meant to achieve, for example

- authentication of card by terminal,
- mutual authentication between card and terminal,
- proving authenticity (or authenticity and freshness) of some data,
- providing some signed transaction digest that can be used for non-repudiation,
- ensuring confidentiality of some data, etc.

This really helps the reader – and yourselves – to keep track of the bigger picture.

Before describing security protocols in all their gory details, it may be useful to first briefly explain the essential idea behind them, for example

- a challenge-response protocol to prove knowledge of key K, or
- a key-exchange to establish a shared secret key S, etc.

Again, this can really help the reader to keep an overview. Ideally it should be trivial to check that all the security requirements are met by the protocols.

Tips and pitfalls for the design document

- Always try to have a clear separation between:

  1. the *security requirements* together with the rationale for them, and
  2. the *design* of your system, given the security requirements.

  In other words, distinguish the *what* you want to ensure from the *how* you want to ensure it. It is easy for this things to become mixed.

  For example, if you include a log, there is (or should be) some security objective that this is meant to guarantee. The log is the HOW, this objective is the WHY. If you choose to log some things (which often makes sense) but don't mention what this is meant to achieve, you are leaving objectives implicit (or you do not have a clue why you are doing it ⌣̈) (By the way, note that logs typically bring their own security requirements, as integrity of log files can be important).

  Don't include discussion (or repeat discussion) of the rationale for security requirements when presenting the design of the system. And, conversely, don't start presenting the design in the section on security requirements.

- Ideally, you want simple checklists of requirements, attacks, assumptions, etc. This then makes it easy for the reader (and yourselves!) to check that all security requirements are guaranteed by your design, given your attacker model and assumptions you made. This should make it simple to check if you have not overlooked any possibilities for attacks, left important security requirements implicit, or made some unrealistic or questionable undocumented assumptions. If there is a lot of prose mentioning requirements, attacks and design decisions in different places, it can get very hard to keep an overview.

Tips and pitfalls for your protocols

- **ENCRYPTION DOES NOT ENSURE INTEGRITY**.

  Encrypting data and then assuming that this encryption ensures integrity is a very common mistake in protocol designs. In fact, it is embarrassing to see how many students who are doing a master in security still make this basic mistake!

  The simple and robust way to authenticate a message M – or more generally, ensure some form of integrity – is to

  - first take the hash $h(M)$ of that message;
  - then encrypt that hash, either with a symmetric key, to produce a so-called MAC (Message Authentication Code), or with a private asymmetric key, to produce a digital signature.

  If you're extremely careful (or very lucky) you might get away with using encryption to ensure integrity (namely, if part of data that is encrypted has some known, fixed value, so that an arbitrary value won't decrypt to a sensible value and is rejected), but better not to rely on that. [2]

- More generally, any use of encryption (to provide confidentiality) WITHOUT also using MACs or digital signatures (to provide integrity) is VERY SUSPICIOUS, because if confidentiality of some data is important, then integrity is usually also important.

- In you introduce some formal notation for signing (or MACing) messages, for example $sign_K(M)$, make it very clear if this means (i) the message M appended with the digital signature (or MAC) or (ii) just the digital signature.

- **Double check that your protocol does not have replay possibilities.** This is a common and easy mistake to make.

  To avoid replays, it is best to let *both* sides in the protocol contribute some nonce (either a random number or some transaction counter), and then involve both these nonces in all subsequent communication.

  Note that there are different variants of replay attacks: an attacker could replay an entire session, but an attacker could also replay individual messages within a session. To avoid the latter, you can include a message

[2]There are authenticated encryption schemes of course, but these are not readily available on smartcards. Also, for non-repudiation you typically do have to consider encryption and signing as separate constructs.

counter that increases at each step, or increase nonces at each protocol step. Alternatively, you can also make sure your implementations are very strict in only accepting messages in one fixed order and rejecting any messages arriving out of order.

- Related to the point above: beware of the difference between 'message integrity' of an individual message and 'session integrity' of a whole session, i.e. a sequence of messages.

  Having MACs or digital signatures on individual messages provides message integrity. If you include some sequence numbers in messages you can obtain session integrity and reduce the scope for MITM attacks.

- A nonce (number used once) included in protocols to prevent replays does not have to be a random number. It can also be a simple incrementing counter or a time-stamp. Using a counter rather than a random number can have advantages both when it comes functionality and security: it provides a unique sequence number for transactions, which is useful anyway, it establishes a clear ordering of transactions, and it may make it easier to detect replays than a random nonce.

- The smartcards sued can do standard crypto (at least AES and RSA; not all have ECC) and hashing algorithms (e.g. SHA2), but don't get distracted by such boring details too soon. Just choosing where to use symmetric or asymmetric crypto and some hash functions is good enough for the initial design. Remember that block ciphers come in different modes.

- If you choose to have a PIN code, consider

  - if it is really worthwhile for people to remember a PIN code for this.
  - for which type of transactions a PIN is required (maybe not at all?).

  If your smart cards use PIN codes, you typically also want functionality for people to change them, for convenience's sake. (Except for high-end security systems where it may be acceptable to impose an unchangeable PIN on users.)

- Esp. for those of you who did the Verification of Security Protocols course: it is a nice idea to use ProVerif of some similar tool to verify your protocol, but be careful, as it is dangerous to assume a protocol is secure just because it passes ProVerif. We have seen many security protocols that were verified with ProVerif but which were fatally flawed.

  Root cause of the problem here that while it is simple to use ProVerif to verify secrecy of the keys, it is more complicated to specify and verify security requirement such as (mutual) authentication or non-repudiation. Using ProVerif can be DANGEROUS if you only verify the secrecy of keys and then assume your protocol is OK, because this misses the trickiest aspects of the protocol...

  Many protocols will consist of mutual (or one-way) authentication followed by some 'transaction' (i.e. increasing balances, paying rations, etc.). The tricky bit to get right is the transaction here. It is easy to do the mutual authentication right, but the transaction data should also include some nonces, transaction counters, etc. to ensure they cannot be replayed. That is easy to miss in ProVerif verification, unless you are very careful specifying the right properties to verify.

We don't want to discourage people from using security protocol analysers – they are great tools – but maybe they should carry an obligatory government health warning:

> *Using ProVerif can seriously damage your security if you just assume it will automatically check that protocols are secure without you still having to think REALLY REALLY HARD about expressing ALL the relevant security requirements.*

- Don't forget to explicitly mention the blocking of issuance functionality (e.g., initialising keys, card numbers etc.) as explicit step in the protocol(s) for initialisation.

- More generally, it is good practice to make a state diagram of the life cycle of a card. Note that there are some associated security requirements, e.g., 'keys on an already personalised card can never be changed' or 'revoked cards cannot be reactivated'. All rather obvious maybe, but good to mention such things explicitly so they not forgotten later on.

  In a real system, the state diagram for the life cycle would also involve steps to lock down the card in order to disable applets to be removed or additional ones to be installed. The GlobalPlatform API implemented on the smartcard provides functionality to do this. However, you should NEVER attempt to this, as it will render our limited supply of smartcards useless.

3 Implementation

After you've handed in the initial design, go ahead and build your design.

- Keep track of any design decisions you make along the way, and record where you deviate from the original high level design. This may happen because of
  - technical restrictions,
  - you run out of time, or
  - you thought of better ways to do things.

- You may find out that when you implement your protocols you have to slightly tweak things to fit with the ISO7816 protocol, e.g. because the terminal always has to start any protocol, or because there is a limit in the amount of data you can send with each command. If so, update your protocol descriptions.

- Implementing the crypto can be time-consuming and nasty to debug. It may be wise to make a first implementation without any of the cryptographic checks in place, and then add these one by one to make debugging easier.

- Nearly all JavaCards, including the ones we give you, do not have a garbage collector. This means that memory leaks will cause the card to block (with a strange error message) eventually because it runs out of RAM or EEPROM. Therefore, make sure that after the applet is installed, initialised and personalised, there are never any additional calls of `new` or invocations of `makeTransient...Array`.

  Also be careful with factory methods that return objects, such as `getInstance` to create `Cipher` or `RandomData` objects. If you call such methods after initialisation/personalisation, carefully read the Java Card API specs to check that these calls do not use leak memory. The easiest way to browse the API specs is of course using the HTML generated from the APIs JavaDocs; see links on the course webpage. When in doubt, it is better to create instances of `Cipher` or `RandomData` objects only once to avoid memory leaks.

- Don't use recursion in any of the smartcard code.

  Recursion is an elegant solution to many problems and you have probably been encouraged to use recursion in programming courses. However, on a device such as a smart card, with very little RAM, recursion is dangerous because there is a real risk that you run out of stack space. Running out of stack space will generate weird and informative error messages and hence be a pain to debug.

- When you make choices (e.g., to use RSA with 1024 bits, SHA-1 etc.), give a reason or simply admit (as may be the case here) that you have no real reason or simply have chosen it because the card supports it or you thought is was fun to try out. For this project this may not seem really necessary. In real life, however, there will be serious consequences to consider, e.g.,

  - Is 1024 bit enough? (Or over the top?)
  - Are smart cards that can do RSA not too expensive? etc.

  In systems that will be around for a long time, you might have to consider how you are going to cope with upgrading to 2048-bit keys in 10 years' time – or worse, with migrating to quantum-resistant crypto. For this project you don't have to worry about that.

- Use some minimal JavaDoc in your code. At least, include `@author` tags to keep track of which group members wrote or changed which code.

4 Second Document: Final Report

The final report should be an update of the first design document, updated to take into account our earlier feedback and any changes that happened along the way: mention

- any shortcuts you made in the implementation – either because of technical problems or simply to meet deadlines;
- any further improvements you thought of only; or

- things that you didn't have time to implement in the end.

You final design document should include:

a) Tables or lists which give an overview of

  - notations you use, like $\{\ldots\}_K$, $MAC_K(\ldots)$, K_M, PK_S, SK_S for encryption, signing, the various keys etc.. Ideally program variables for keys should match the notation used in the design document, so that it's easy for any code reviewer to see the correspondence of the design and the code.
  - which keys, certificates and other crucial data are known by the various parties (incl. the card)

b) Your protocols as message sequence charts, or in "Alice $\rightarrow$ Bob: ..." style. In addition to the messages exchanged, ALSO include the important (security critical) actions that the parties take. This can be things like incrementing balances, setting or unsetting flags, etc.

c) A state diagram of the persistent state of the card. This will inevitably include an 'initial' state and an 'issued/personalised' state, but maybe additional state, e.g., an state to flag the card is in the middle of some protocol to be able to detect card tears, or a 'blocked' state to block cards that are reported lost or stolen.

government Both a) and b) should have already be included in the design document (see Section 2). Make these tables and protocols specs before you start implementing anything.

For c) you should explain how this state diagram is realised in the code. This will typically involve one or more variables in EEPROM that record in which lifecycle state the applet is.

If your implementation has to keep track of some session state, to keep track of where it is in a protocol so that it can reject messages coming out of sequence, you may also want to specify that state diagram, and explain how that state is recorded. This will typically involve one of more variables in RAM. Note that the only way to allocate such variables in RAM will be as transient arrays.

Keep the report short and to the point: 10 pages max. or 8 pages if you use 2-column style. To get in the right frame of mind when writing the document, you may imagine you're trying to present your application to a technical expert working for the organisation that commissioned the application, and that you have 10 pages to explain the security requirements you identified and to explain how implementation in the end ensures these.

Tips and pitfalls

- Be frank about any limitations and shortcomings of your solution!
- At some stage you will have to decide about how long your data fields have to be; e.g., if things fit in a `byte` or you need a `short` etc. These are design decisions that should not just be implicit in the code. You don't have to decide all this at the beginning, – in fact, you probably shouldn't – but you should remember to document such decisions in your final report.

- Have you thought about which steps or parts of protocols need to be atomic, and do you need to use JavaCard transactions for this in the smart-card code?