622.755 – Crypto Engineering Winter Term, 2023/24

Project: SuperSAM Security Analysis

Sebastiano Morson 12246599

1 Executive Summary

The proposed project highlights numerous security flaws from both hardware and software perspectives. Analyzing the provided implementation revealed a general tendency to follow standardized approaches, but with variations that, although seemingly trivial, represented significant vulnerabilities. In particular, the implementation of AES did not faithfully adhere to the design outlined in the implementation notes, and the encryption function was not adequately designed to protect the confidentiality of the used seeds.

The implementation notes demonstrated shallow and unmotivated design choices, such as the use of a cyclic group now considered insecure and the Diffie-Hellman protocol lacking measures to ensure the integrity, confidentiality, and availability of exchanged data, exposing the entire system to severe security flaws. Additionally, the implementation notes did not sufficiently address the hardware protection measures applied to RFID tokens, which should be well-defined in a context where users have direct access to this component of the system.

In general, the main vulnerabilities I highlighted include: - Issues related to the use of Diffie-Hellman - Issues related to the lack of evident countermeasures to physical attacks on the device - Issues related to an incorrect choice of parameters to initialize the AES encryption algorithm

Regarding the proposed improvements, the most constraining constraint is given by the logical and storage capabilities of RFID systems, posing more pronounced security challenges than systems with greater computational resources. Mechanisms deemed more secure in traditional contexts, such as asymmetric encryption, lose effectiveness when inserted into a context like this, where limited computing resources necessitate a lightweight view of the utilized protocols. The major challenge was therefore to balance security and performance efficiency, with a proper management of the available resources. In doing so, I attempted to offer solutions that were as interconnected as possible and took into account the issues that had emerged up to that point, avoiding the presentation of discordant solutions or those that did not integrate well together.

2 Specific points

2.1 Implementation Note: Wrong Generator, Page 3, Section 2.3

Problem Description: Incorrect Generator Value. Within the implementation notes on page 3, it is stated: *"We follow RFC 3526 https://tools.ietf.org/html/rfc3526"*. However, the generator used by DH does not match the one specified in RFC 3526 [1], page 3, section 2, "1536-bit MODP Group," which is fixed at 2.

Demonstrating this as a vulnerability: The chosen generator g must possess the property of generating the entire prime group of p by raising g to a value x. If g is a correct generator, all $p-1$ elements can be represented by g^x for different values of x. It is crucial that this property holds since the discrete logarithm problem relies on this characteristic. If g is not a generator of p, there is a risk of generating only a much smaller subgroup within which the DHP (Diffie-Hellman Protocol) no longer poses a computational challenge, thus undermining the security of the key exchange used in the system.

Proposed Improvement: To use 3 as the generator, it is necessary to verify that it is indeed a primitive root of the considered group. To do this, it is sufficient to check that $0 \leq g < p$ and $(g, p) = 1$ (g and p are coprime). Therefore, I consider the value p as indicated in RFC 3526, while $g = 3$.
I calculate $\phi(p) = p - 1 = phi$ and then obtain the prime factors of phi, considered as the set composed of all $q_i^{r_i}$, where each q_i is a prime factor, and r_i is the power of that prime factor. At this point, I verify that

$$g^{\phi(p)/q_i} \not\equiv (1 \mod p) \; \forall q_i$$

By performing this test using the tool SageMath, I have confirmed that $g = 3$ is a generator for the value p and therefore does not pose security issues. Without this verification, there was a risk of using an invalid generator. It would be advisable to assess any performance improvements provided by the standard generator 2 proposed in the documentation.

2.2 Implementation Note: Insecure Group,Page 3, Section 2.3

Problem Description: The DH protocol uses an insecure group. Within the implementation notes on page 3, section 2.3, the use of the RFC 3526[1] MODP-1536 group is specified, which, however, does not guarantee an adequate level of security.

Demonstrating this as a vulnerability: The Diffie-Hellman group with a 1536-bit modulus, also known as DH-1536 or Group 5, has been deemed vulnerable mainly due to the increased computational power available, making it easier to execute attacks based on modulus factorization. In December 2015, the LogJam research group revealed that DH-1024 (1024-bit modulus) groups were vulnerable to an attack known as "Logjam." This attack is based on an attacker's ability to precompute and store a set of parameters that can be used to reduce the time needed to attack the shared secret key. The DH-1536 group, although more robust than DH-1024, is still considered not sufficiently strong [2][3][4].

Furthermore, within RFC 8247 [5], on page 9, it is stated that this group has been downgraded to a "SHOULD NOT" (be used) as it is anticipated to be easily computable in the coming years.

Proposed Improvement: The general idea is to use a large MODP group to make solving the discrete logarithm problem more challenging and thus enhance the security of Diffie-Hellman. Research conducted by [6] regarding the insecurity of groups smaller than DH-1024 recommends using groups of at least 2048 bits. Considering the hardware capabilities of the devices used, parameters should be chosen starting from the 2048-bit MODP Group, as indicated from page 3 in RFC 3526[1], and following updated security recommendations from reliable sources such as the Cisco community.

2.3 Implementation Note: Ticket Manipulation

Problem Description: The communication scheme used does not ensure the integrity of the ticket during the execution of the $SendTicket()$ function. This allows a user to gain access to any building by modifying the ticket sent through the $SendTicket()$ function.

Demonstrating this as a vulnerability: During a normal interaction, a user performs the following process:

1. $Token \xrightarrow{Hello} Reader$
2. $Reader \xrightarrow{Hello, Authenticate(g^R)} Token$
3. $Token \xrightarrow{SecureId, g^T} Reader$
4. $Reader$ calculates sk =CBC-MAC($g^{T \cdot R}$)
5. $Token$ calculates sk =CBC-MAC($g^{R \cdot T}$)
6. $Token \xrightarrow{Enc_{sk}(Ticket)} Reader$
7. $Reader$ executes $Ticket = Dec_{sk}(Enc_{sk}(Ticket))$ and authorizes access if the building is present in $Ticket$.

Since the card does not implement memory protection mechanisms preventing reading, the user could:

1. Read the value of sk calculated in step 5
2. Calculate $ticket = Dec_{sk}(Enc_{sk}(Ticket))$
3. Forge a $newTicket$ by modifying $ticket$ and inserting arbitrary building IDs
4. Send $Reader$ the value $Enc_{sk}(newTicket)$

At this point, $Reader$ would grant access to the new buildings specified by the user in step 3.

Proposed Improvement: At the core of this attack is the user's ability to read the memory of the card through tampering. To address this issue, I suggest protecting the card from memory reading through physical attacks, for example, by using a hardware shield that erases the card's memory in case of tampering. Additionally, I propose not storing the ticket inside the token; instead, encrypt it with a key held by the token. Instead, store a version called $secureTicket$ constructed as follows:

$$secureTicket = Enc_{srk}(ticket|HMAC_{srk}(ticket))$$

where srk is a secret key shared among all readers implementing $UpdateTicket()$ and unknown to the $Token$. Passing $newTicket$ through a hash function aims to prevent it from being read and modified by an attacker who can read sk and smk.

The main assumption is that the secrecy of srk is guaranteed. In the event of a breach of the srk secret, it would be necessary to provide an $UpdateSRK()$ function to update the srk key on all readers aware of srk. I would also expect each card to contain a value $sec = HMAC_{srk}(srk)$ to perform a conformity check of the srk key by the readers. In case of srk violation, $UpdateSRK$ defined in section 2.3 could be invoked.

2.4 Implementation Note: Replay Attack

Problem Description: The Diffie-Hellman implementation intended for use lacks any protection mechanisms against replay attacks, allowing an attacker to reuse records of past Token communications to instantiate new communications that appear legitimate.

Demonstrating this as a vulnerability: Assuming that within a session, multiple function calls such as UpdateSMK(), UpdateTicket(), and SendTicket() may occur, the following sequence of events could occur:

1. Reader uses UpdateSMK (attacker copies the transaction)
2. Token correctly updates the card

3. Reader uses UpdateSMK
4. Token correctly updates the card
5. Attacker replays transaction 1
6. Token updates the card again

This scenario can be easily extended also to UpdateTicket() and SendTicket() functionalities. After this attack, an inconsistency arises between the reader and the token. This attack would violate the system's availability, preventing normal usage. If the system were deployed in a hospital, this could impede an operator, potentially causing harm during an emergency situation.

Proposed Improvement: Under the assumption that within a session, multiple function calls such as UpdateSMK(), UpdateTicket(), and SendTicket() can occur, it is crucial to implement a mechanism preventing an attacker from replaying previously occurred communications. One approach is to use a nonce added to each communication, such as a randomly generated value using the get_rand() function from the standard C library.

In case the basic assumption is not legitimate, meaning that within a single session, it is not possible to establish communication between the reader and token using the same functionality more than once, it would not be possible to execute a replay attack. This is because the session key is expected to be modified (or at least it is unlikely to reoccur in two consecutive sessions).

2.5 Implementation Note: Absence of Mutual Authentication in DH Communication

Problem Description: DH is known to be vulnerable to Man-In-The-Middle attacks, and the current implementation lacks countermeasures against such attacks. This would allow an attacker to impersonate one of the communication parties.

Demonstrating this as a vulnerability: The authenticity of the reader is not verified by the card. Consequently, a third party could emulate a reader and make modifications to the ticket associated with the card. This poses significant security problems, as an attacker could update the ticket's value.

To update the ticket, it is sufficient to know a common key between the reader and the card, after which the ticket encryption is performed using the secret key smk. The attack would unfold as follows:

1. $Token \xrightarrow{Hello} Reader$
2. $Reader \xrightarrow{Hello, Authenticate(g^R)} Attacker \xrightarrow{Hello, Authenticate(g^A)} Token$

3. $Token \xrightarrow{SecureId, g^T} Attacker \xrightarrow{SecureId, g^A} Reader$
4. *Reader* calculates sk =CBC-MAC($g^{A \cdot R}$)
5. *Token* calculates sk =CBC-MAC($g^{A \cdot T}$)
6. *Attacker* calculates $sk1$ =CBC-MAC($g^{A \cdot R}$) *and* $sk2$ =CBC-MAC($g^{A \cdot T}$)

Step 6 is possible because the specifications indicate that CBC-MAC uses a fixed key with a value of 0. If this were not the case, it would be more challenging for the attacker to obtain the value of sk since they would need to perform a brute force attack on the key used by CBC-MAC.

After completing step 6, the attacker would be able to intercept and discover the secret key smk during the execution of the update function of the System Master Key, as they would be able to decrypt the encrypted packet $Enc_{sk}(smk)$.

Ultimately, the attacker exploiting the Man-In-The-Middle attack could easily clone a card by performing the following steps:

1. Attacker retrieves the secureId in step 3 at the beginning of the interaction
2. Attacker uses the SendTicket function to retrieve the Token's ticket
3. Attacker uses the UpdateSMK function to retrieve the secret key smk

Proposed Improvement: To correct this vulnerability, I would recommend modifying the DH protocol, following the DH protocol proposed in RFC 5683 [7], to ensure the authenticity of the reader. This DH version uses a shared secret between the two parties and three hash functions used as random functions to allow mutual authentication. However, it requires the reader to also store its identity, similar to how the token stores its secureId. It is not necessary for each reader to store a different identity. The shared secret PW will be used as the smk key, and the identities A and B will be used as the secureId of the token and a readerId value stored by the reader, respectively. For the correct implementation, refer to pages 3 and 4 of the RFC [7].

A simpler solution, although less secure and lacking integrity, follows the model below:

1. *Reader* and $Token$ establish private keys R and T.
2. $Token \xrightarrow{Hello} Reader$
3. $Reader \xrightarrow{Hello, Authenticate(Enc_{smk}(g^R)}$
4. $Token \xrightarrow{Enc_{CBC-MAC_{smk}(g^{RT})}(secureId), Enc_{smk}(g^T)} Reader$
5. *Reader* computes $sk = CBC - MAC_{smk} g^{R \cdot T}$ and checks the $secureId$ of $Token$ by computing $Dec_{sk}(Enc_{sk}(g^R T))$. Then, it picks a random value x and computes $sk = CBC - MAC_x(g^{R \cdot T})$

6. $Reader \xrightarrow{Enc_{sk}(x)} Token$

7. $Reader$ computes $sk = CBC - MAC_x(g^{R \cdot T})$

A more secure solution is to use Diffie-Hellman with asymmetric encryption, but this would make the system less scalable and require more functions for updating public and private keys.

The second proposed method has the problem of allowing the key sk produced at the end of the operations to be mapped through CBC-MAC to reduce its size, posing collision risks. For a discussion on security and possible improvements resulting from reducing the key sk to 1536 bits from 128 bits, refer to section 2.8.

2.6 Implementation Note: AES-ECB to Forge New Ticket

Problem Description: Following the implementation notes, it is possible for an attacker to gain access permissions to a building by exploiting the System Master Key (SMK) update function and the encryption properties of AES-ECB.

Demonstrating this as a vulnerability: According to the implementation notes (page 3, line 8), "there may be several master keys." This implies that multiple users may possess the same SMK simultaneously. Knowing that the ECB mode will produce identical encrypted blocks for equivalent plaintext blocks, let's consider tokens A and B with the following characteristics:

- ticketA = building1, building2, building3
- ticketB = building1, building2
- smkA = updatedSMKB (smkB after an update)

It is possible for user A to exploit the fact that the System Master Key update function sends the secureTicket in clear text to observe how different secureTickets are presented. Assuming that building1 is the main building, and many users have access to it, if user A intercepts user B's communication during the SMK update phase, they could obtain secureTicketB.

Since ECB does not guarantee the indistinguishability property, if two ciphertexts encrypted with ECB with the same SMK key have plaintext in the form ABAAB and AABAB (with A and B being blocks of equal size), the ciphertexts after encryption will have the first, fourth, and fifth blocks the same. Therefore, if user A notices that the first part of secureTicketA and secureTicketB is the same, they could deduce that the key used for encryption is the same. At that point, they could concatenate their own secureTicketA with the secureTicket of B to gain access to buildings they do not have access to. Obtaining user B's secureId (a trivial operation as it is exchanged in clear text after establishing the session key) would allow user A to clone user B's card and access

all of their buildings or update their own secureTicket without knowing the session key used.

Proposed Improvement: To prevent this type of attack, it is necessary to ensure that the UpdateSMK function does not exchange the secureTicket in clear text after the update. One method to mask the secureTicket is as follows:

1. Reader: UpdateSMK($Enc_{sk}(smkU)$,$HMAC_{sk}(secureTicketU)$
2. Token: Check if $HMAC_{sk}(secureTicketU) = HMAC_{sk}(secureTicket)$, then update accepted, otherwise reject the update

where:

- smkU is the SMK after the update
- smk is the SMK before the update
- sk is the session key
- secureTicketU = $Enc_{smkU}(ticket)$
- secureTicket = $Enc_{smk}(ticket)$

This way, it is not possible to exploit the pattern matching offered by ECB. As the hash function is one-way, the attacker would not be able to observe the victim's ciphertext or concatenate it. The key used to perform the hash can actually be fixed to an arbitrary value without affecting the overall security of the SMK update function.

Another possible improvement is the method already mentioned in section 2.7, subsection 2.7.

2.7 Implementation Note: UpdateSMK and UpdateTicket do not guarantee integrity

Problem Description: The functions UpdateSMK and UpdateTicket, defined on pages 4 and 5 of the implementation notes, do not incorporate mechanisms to ensure the integrity of the exchanged messages. In the event that the key *sk* is discovered (e.g., through a Man-in-the-Middle attack), an attacker could compromise the confidentiality of the ticket and the key *smk*.

Demonstrating this as a vulnerability: If an attacker gains knowledge of the session key sk, for instance, through a Man-in-the-Middle attack during the communication initialization (assuming it is not corrected or not effectively secured), the attacker could:

1. Intercept the communication
   $Reader \xrightarrow{Enc_{sk}(newTicket)} Token$
   when the UpdateTicket function is called.
2. Calculate the value $newTicket = Dec_{sk}(Enc_{sk}(newTicket))$.
3. Modify the value $newTicket = fakeTicket$.
4. Proceed with the communication
   $Attacker \xrightarrow{Enc_{sk}(newTicket)} Token$.

At this point, Token would be unaware of the message manipulation and would subsequently update its ticket, believing it to be legitimate.

Similarly, an attacker could:

1. Intercept the communication
   $Token \xrightarrow{sendTicket(Enc_{sk}(Ticket))} Reader$
   when the SendTicket function is called.
2. Calculate the value $Ticket = Dec_{sk}(Enc_{sk}(Ticket))$.
3. Intercept the communication
   $Reader \xrightarrow{UpdateSMK(Enc_{sk}(smk),secureTicket)} Token$.
4. Calculate the value $Enc_{sk}(smkFake)$.
5. Modify the value $secureTicket = Enc_{smkFake}(Ticket)$.
6. Proceed with the communication
   $Attacker \xrightarrow{UpdateSMK(Enc_{sk}(smkFake),secureTicket)} Token$.

In this scenario as well, Token would not be aware of the secret key modification, and it would update its smk since $Dec_{smkFake}(secureTicket) = Ticket == Dec_{smkOld}(secureTicketOld)$. Updating the key with a non-legitimate one would lead to inconsistencies between what is held by Token and what Reader knows, preventing further iterations post-update.

Proposed Improvement: It is crucial that the UpdateTicket and UpdateSMK functions ensure the integrity of the exchanged message, even if the session key loses its secrecy property. I assume that the card has mechanisms in place to prevent memory reading, such as a shield that performs a memory wipe if tampering is detected.

Assuming that the physical tamper protection mechanisms are effective and that an attacker cannot read the token's memory, to ensure integrity, I would implement an HMAC function within the card. Subsequently, I would modify the UpdateTicket function as follows:

1. $Reader \xrightarrow{UpdateTicket(Enc_{sk}(newTicket|HMAC_{smk}(newTicket)} Token$
2. $Token$ computes:
   (a) $Ticket = Dec_{sk}(Enc_{sk}(newTicket))$
   (b) if $HMAC_{smk}(newTicket) = HMAC_{smk}(Ticket)$
   then $Token$ sets $secureTicket = Enc_{smk}(Ticket)$
   otherwise reject

Similarly, for the UpdateSMK() function, the reasoning is analogous. Assuming that the physical tamper protection mechanisms are effective and that an attacker cannot read the token's memory, to ensure integrity, I would implement an HMAC function within the card. Subsequently, I would modify the UpdateSMK() function as follows:

1. $Reader \xrightarrow{UpdateSMK(Enc_{sk}(new\_smk),HMAC_{new\_smk}(new\_smk|secureTicket)} Token$
2. $Token$ computes:
   (a) $smk = Dec_{sk}(Enc_{sk}(smk))$
   (b) if $HMAC_{new\_smk}(new\_smk|secureTicket)$
   $= HMAC_{new\_smk}(new\_smk|Enc_{new\_smk}(Dec_{smk}(secureTicketOld))$
   then $Token$ sets smk to new_smk,
   otherwise reject

Where:

- new_smk is the new smk key proposed by the $Reader$
- smk is the old key used by the $Token$
- $secureTicket = Enc_{new\_smk}(Ticket)$
- $secureTicketOld$ is the ticket previously held by the $Token$

If I couldn't guarantee device read protection to prevent potential ticket updates, as discussed in Section 2.3, I would modify UpdateTicket() as follows:

1. $Reader \xrightarrow{UpdateTicket(Enc_{srk}(newTicket|HMAC_{srk}(newTicket)} Token$

2. $Token$ computes:

   (a) $Ticket = Dec_{sk}(Enc_{sk}(newTicket))$

   (b) if $HMAC_{smk}(newTicket) = HMAC_{smk}(Ticket)$
   then $Token$ sets $secureTicket = Enc_{smk}(Ticket)$
   otherwise reject

Here, srk is a secret key shared among all readers implementing UpdateTicket() and unknown to $Token$. Passing $newTicket$ through a hash function aims to prevent its value from being read and modified by an attacker capable of reading sk and smk.

Similarly, if I couldn't guarantee device read protection to prevent potential ticket updates, as seen in Section 2.3, I would modify UpdateTicket() as follows:

1. $Reader \xrightarrow{UpdateSMK(Enc_{sk}(new\_smk),HMAC_{new\_smk}(new\_smk|HMAC_{)}} Token$

2. $Token$ computes:

   (a) $smk = Dec_{sk}(Enc_{sk}(smk))$

   (b) if $HMAC_{new\_smk}(new\_smk|secureTicket) =$
   $= HMAC_{new\_smk}(new\_smk|Enc_{new\_smk}(Dec_{smk}(secureTicketOld))$.
   then $Token$ sets smk to new_smk,
   otherwise reject

Where:

- new_smk is the new smk key proposed by the $Reader$
- smk is the old key used by $Token$
- $secureTicket = Enc_{new\_smk}(Ticket)$
- $secureTicketOld$ is the ticket previously held by $Token$
- srk is a secret key shared among all readers implementing $UpdateSMK()$ and unknown to $Token$.

Here, the assumption is that the secrecy of srk is guaranteed. If the secrecy of srk is compromised, a manual update of srk on all readers implementing the UpdateTicket() function would be necessary.

Additionally, I would expect each card to contain a value $sec = HMAC_{srk}(srk)$. In case of srk violation, calling $UpdateSRK$ would be possible, defined as:

1. $Reader \xrightarrow{UpdateSRK(request)} Token$

2. $Token \xrightarrow{sec} Reader$

3. $Reader$ if $HMAC_{srk}(srk) = sec$ then
   $Reader \xrightarrow{Enc_{srk}(srk)*Enc_{sk}(nonce),HMAC_{new\_smk}(Enc_{new\_srk}(new_srk)|smk)*nonce} Token$
   otherwise stop the operation

4. $Token$ calculate

$$K = HMAC_{new\_smk}(Enc_{new\_srk}(new\_srk)|smk)/Dec_{smk}(Enc_{smk}(nonce))$$

5. $Token \xrightarrow{HMAC_{sk}(nonce+1|K|smk)} Reader$

6. $Reader$ computes $M1 = HMAC_{sk}(nonce + 1|K|smk)$

7. $Reader$ checks if $M1 = M$ where M is the previous message received by token then

$$Reader \xrightarrow{HMAC_{sk}(nonce+2|Enc_{smk}(\text{"}success\text{"}))} Token$$

   $Token$ set $sec = HMAC_{new\_srk}(new\_srk)$
   otherwise reject the operation

The use of a nonce serves the purpose of ensuring that replay attacks cannot be executed, the HMAC function guarantees message integrity, and the use of both keys sk and smk aims to make completing the transaction more complex, as one would need to recover both keys. Clearly, this function assumes the impossibility of carrying out physical attacks to retrieve both sk and smk and requires a secure exchange of the session key.

2.8 Implementation Note: CBC-MAC for DH-Session Key Reduction Leads to Insecure Session Key, Page 4, Section 2.4.1, Row 5

Problem Description: Within the implementation notes, on Page 4, Row 5, Section 2.4.1, it is specified that the 1536-bit key generated through Diffie-Hellman is reduced to 128 bits using CBC-MAC, assuming that CBC-MAC ensures a form of probabilistic uniformity in the output, despite the large size of the generated key.

Demonstrating this as a vulnerability: The number generated by Diffie-Hellman is over 1536 bits, and an attempt is made to reduce it by a factor of 12, bringing it down to 128 bits. This results in a significant number of collisions, and the session key would no longer depend on the discrete logarithm problem (considered secure), but rather on the security of CBC-MAC, which is not intended to work as a Key Derivation Function (KDF). If key reduction is not performed while ensuring fair output probability and a

reduced number of collisions, it can decrease the attacker's efforts needed to discover the session key, rendering any encryption relying on the session key sk ineffective and compromising the overall security of the system.

Proposed Improvement: The suggested approach is not to generate a 128-bit session key and then reduce it; rather, it is better for the secret to be generated by an algorithm that produces a 128-bit value directly. This avoids compromising security.

The more rigorous method would involve updating the entire shared secret generation process, using, for instance, the protocol proposed by Alamr, Kausar, and Kim [8]. This protocol utilizes an enhanced version of Elliptic Curve Diffie-Hellman (ECDH) to generate a session key, providing various security properties such as mutual authentication, anonymity, confidentiality, forward security, location privacy, resistance to man-in-the-middle attacks, resistance to replay attacks, and resistance to impersonation attacks, as stated in the paper's abstract.

A more straightforward solution is to use a Key Derivation Function (KDF). A KDF is a cryptographic algorithm designed to generate a robust secret key from a single key value. Its primary purpose is to protect sensitive data by discouraging attackers from predicting or deciphering the keys used.

To reduce the 1536-bit key sk to 128 bits, instead of using CBC-MAC (not designed as a KDF), consider using HKDF (HMAC-based Key Derivation Function). Clearly, this involves an increase in the number of functions to be implemented on the card, so the context in which the system will be applied, the number of users, and the technical specifications of the RFID tag must be taken into consideration.

2.9 Implementation Note: Fixed Key Equal to Zero Facilitates MitM Attacks, Page 3, Section 2.3

Problem Description: Using a hash function with a fixed key makes it easier for an attacker to impersonate a reader or token by conducting a MitM attack.

Demonstrating this as a vulnerability: At the bottom of page 3 in the implementation notes, a modification to the standard Diffie-Hellman protocol is described, using the session key value of $CBC - MAC_k(g^{r \cdot t})$ to reduce the size of sk. The CBC-MAC function makes it more challenging for an attacker to carry out a MitM attack. Assuming that the attacker has obtained the Reader's value R and forwarded a fake value E to the Reader (thus impersonating the token), they would be able to calculate the value $g^{R \cdot E}$ but not $CBC - MAC_k(g^{R \cdot E})$ without knowing the key k used for hashing. The issue is that at the bottom of page 3, the notes state that k is fixed at 0. Consequently, Eve would be able to calculate sk and easily impersonate one of the communication parties.

Proposed Improvement: Using a secret key for hashing would be a naive but better choice than the applied one, preventing an attacker from easily calculating the key sk.

A more appropriate choice would be to use the value of *smk* as the key for CBC-MAC, representing a secret shared between the Reader and Token. However, this secret key should also be stored inside the card, with no specified measures preventing its reading through physical attacks such as cold boot or register reading. The issue of using CBC-MAC for the described purpose has also been addressed in the previous paragraph 2.8.

2.10 Implementation: Number of Rounds, masked_combined.c, Row 21

Problem Description: The AES implementation uses 9 rounds, deviating from the AES standard, which requires 10 rounds for a 128-bit key. This non-conformity could compromise the security and integrity of the system, as the number of rounds directly affects the algorithm's resistance to cryptographic attacks.

Demonstrating this as a vulnerability: The paper "Structural Evaluation of AES and Chosen-Key Distinguisher of 9-round AES-128" [9] by Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin from the University of Rennes 1, France, École Normale Supérieure, France, and Nanyang Technological University, Singapore, describes a distinguisher for AES-128. A distinguisher is a cryptographic attack technique that seeks to distinguish between encrypted data and random data. In other words, a distinguisher is an algorithm that takes a set of data as input and determines whether this data has been encrypted with a particular key or generated randomly. However, the existence of a distinguisher for AES-128 with 9 rounds does not necessarily imply that using only 9 rounds is a vulnerability. In general, the security of a symmetric cipher like AES-128 depends on the number of rounds used, the choice of the key, and the correct implementation of the cipher. Choosing to use 9 rounds provides attackers with an opportunity to find a way to attack encrypted messages.

Proposed Improvement: It is crucial that the adopted implementation uses 10 rounds to avoid the possibility of exploiting the distinguisher mentioned in the previous paragraph. Therefore, the value at row 21 of the *masked_combined.c* file should be modified.

2.11 Implementation Note: Lack of Physical Tampering Protection Can Cause Secrets Leakage

Problem Description: Within the implementation notes, there is no indication of the use of effective protections against physical tampering. Without adequate defense measures, sensitive information inside the cards, particularly *sk* and *smk*, becomes vulnerable to physical manipulations, allowing unauthorized access, cloning, or unauthorized reading.

Demonstrating this as a vulnerability: Through physical reading of the values of the keys sk and smk, it becomes evident that the entire cryptographic security system is compromised, as every encryption function relies on these two secrets. An attacker could easily clone or replicate the card and carry out numerous attacks as seen in the previous sections.

Proposed Improvement: To protect an RFID card from physical tampering, various strategies and preventive measures can be adopted. These may include using robust and tamper-resistant materials for the RFID card body, implementing security layers or seals to make unauthorized opening of the card difficult, adopting tamper-evident sealing technologies, placing the RFID chip in a position that makes physical access difficult without damaging the card, and implementing additional protections for the chip itself, such as durable coatings or sealed enclosures.

The use of anti-skimming materials can also make it more challenging to apply skimmers and unauthorized reading devices to the card surface. Without knowledge of the specific card specifications and available budget, it is challenging to determine which of these strategies is the most effective. Nevertheless, it is crucial to implement some form of physical protection.

2.12 Implementation Note: PRNG is not Proven to be Compliant with Random Standards, Page 6, Section 3.4

Problem Description: In the implementation notes on page 6, Section 3.4, it is stated that the PRNG function used is considered secure as it has been tested through serial tests. However, this test alone is not sufficient to ensure the actual capability of generating random numbers. If the PRNG were deterministic, it could pose security issues, especially in the implementation of AES-128 and the Diffie-Hellman key generation if the same PRNG is used.

Demonstrating this as a vulnerability: A PRNG function that does not meet security standards may allow an attacker to predict, for example, the random values used within the masking function of AES. This could potentially undermine the masking capability and provide the opportunity to execute attacks such as DPA. The "randomness_testsuite" analysis tool available on the GitHub page https://github.com/stevenang/randomness_te as described in the repository description, "is a Python implementation of NIST's A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications." A preliminary analysis of the used PRNG highlights some NIST-recommended analysis tests that potentially reveal an insecurity in the PRNG [Fig 1].

Proposed Improvement: It is crucial to perform tests other than the serial test to verify that the PRNG provides effective randomness. For this purpose, it is important to follow the tests proposed within the NIST SP 800–22 [10].

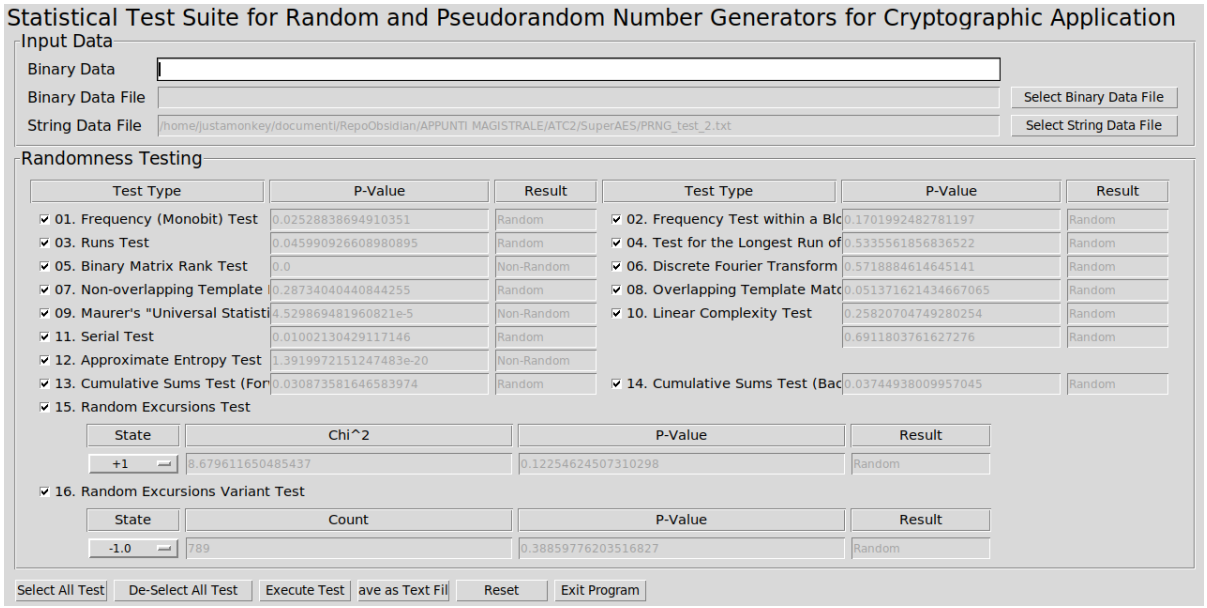

Figure 1: randomness_testsuite results

2.13 Implementation: Wrong Seeds Can Cause DPA Attack, main.c, Rows 14/15

Problem Description: Setting the PRNG seeds to the value 0 turns the getRand() function into a generator of the byte 0x00, allowing an attacker to conduct a first-order DPA attack and consequently recover the encryption key.

Demonstrating this as a vulnerability: The main.c script contains two variables, *unsigned int rngSeed1* and *unsigned int rngSeed2*, initialized to 0. Leaving these two variables unchanged will cause getRand() to always return 0. This behavior is motivated by the fact that, observing the PRNG implementation, during the initialization phase, the values of the *lfsr32* and *lfsr31* variables correspond to those of the used seeds. Once the getRand() function is called, with both *lfsr32* and *lfsr31* set to 0, the value of the *feedback* variable will always be 0. Consequently, the values of *lfsr32* and *lfsr31* will never be updated, and in the last instruction of the getRand() method, $lfsr32 \oplus lfsr31$ will always be calculated as $0 \oplus 0 = 0$.

During the encryption of plaintext, the masked_combined.c script will not correctly apply the mask, which would normally prevent DPA-type attacks.

Considering obtaining physical access to a token and being able to perform a DPA attack through physical removal of the chip and copying to an external medium, to demonstrate its feasibility, I conducted the attack by measuring power traces only during the first round of the S-Box, using Hamming Weight as a model.

After collecting the traces, a Python script was sufficient to analyze the correlations between power traces and hypothetical consumption values. The obtained result is shown

in Figure 2.

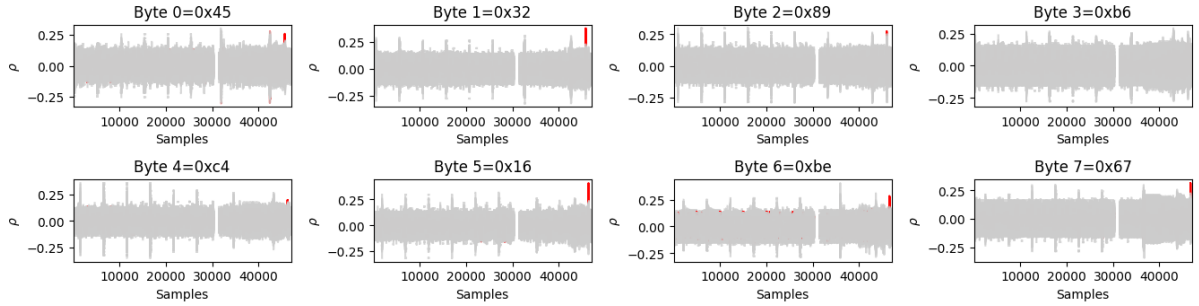

Figure 2: DPA attack on 8-byte key

Figure 2 represents the attack on the first 8 bytes of the key used for encryption, obtained by studying the peaks in correlations with bytes of the hypothetical key. Only the first 8 bytes are marked for demonstrative purposes, but the attack can be conducted for all 16 bytes of the key used in the proposed implementation.

Proposed Improvement: Change the Seeds The two seeds should be initialized with a random value using a suitable PRNG to protect the AES implementation from first-order DPA attacks, as the initial values are shielded by a mask that utilizes the PRNG. To obtain the required random values, I would suggest using the get_rand() function from the stdlib.h library in the C language.

The use of this function (get_rand()) throughout the entire implementation may not necessarily be an appropriate choice, as one would need to assess the performance difference on the system resulting from the use of the getRand() function compared to the get_rand() function.

Another improvement could be reducing the operational range of the system, in order to limiting the electromagnetic emissions of the system, making more difficult to record the power traces.

2.14 Implementation Note: Fault attacks on AES-ECB, Section 2.3, on Page 3

Problem Description: Within the implementation notes, it is stated, quoting, "We will use AES with 128-bit keys as a cryptographic primitive. Because all core data is likely to be smaller than 128 bits, we will use AES in ECB mode." The issue is that the ECB mode will produce identical encrypted blocks for equivalent plaintext blocks. This choice can potentially introduce vulnerabilities to the confidentiality of encrypted data in the event of fault attacks. The implementation notes do not exclude this type of attack.

Demonstrating this as a vulnerability: Since the token is a device provided to users, it must be assumed that they are capable of physically accessing it. Consequently,

in the absence of fault prevention measures, an attacker could launch an adaptive chosen plaintext attack by using a known plaintext concatenated with a chosen plaintext for the attack, encrypting the combination with the key already present in the device. The attacker could leverage specific tools designed to flip a bit in the device's memory, for example, through voltage glitching or laser beam, to perform a fault injection on the AES implementation. Since AES is implemented in ECB mode, and the same key and plaintext always generate the same ciphertext, an attacker could reveal a relationship between modifying the value in one of the secrets and generating an output different from the one previously recorded without the modification. Through the progressive left-shifting of the known plaintext one byte at a time, the attacker can force the decryption of the original plaintext by observing the obtained ciphertexts.

Proposed Improvement: The AES implementation should be protected against fault attacks by using a fault-tolerant mechanism. This would prevent especially the token from being vulnerable to tampering by an attacker. The easiest way to achieve this security goal whould be to use a tamper-resistant RFID tag, which can be found on the market. In order to prevent fault-attacks, a naive strategy could be to make more difficult for the attacker to understand the internal system and behaviour, increasing the complexity of the internal circuit of the RFID chip. However, this strategy might be difficult to apply and expensive because of the small size of the common RFID tags.

2.15 Implementation: Predictability of getRand()'s Seeds, masked_combined.c

Problem Description: The implementation of getRand() allows the derivation of initialization seeds since only the bottom 16 bits are masked. This enables an attacker to predict the random values that will be generated in the future.

Proposed Improvement: To address the issue, the result of $lfsr32 \oplus lfsr31$ should be masked. One method is to use a Key Derivation Function (KDF) to reduce the output size from 32 bits to 16 bits in order to shuffle the 32 bits before output them. This way, it would not be possible for the attacker to brute-force the missing 8 bits and reverse the steps that modify the values of $lfsr32$ and $lfsr31$.

A naive solution could be to use a third secret value q to perform $(lfsr32 \oplus lfsr31) \mathrm{mod}(q)$ before extracting the bottom 16 bits.

References

[1] Mika Kojo and Tero Kivinen. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526, May 2003.

[2] Imperfect forward secrecy: How diffie-hellman fails in practice, https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf.

[3] Next generation cryptography https://sec.cloudapps.cisco.com/security/center/resources/next_generation_cryptography.

[4] diffie-hellman-groups/ta-p/3147010, Jun 2020.

[5] Yoav Nir, Tero Kivinen, Paul Wouters, and Daniel Migault. Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2). RFC 8247, September 2017.

[6] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin Vandersloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, Denver, Colorado, United States, October 2015.

[7] Zachary Zeltsan, Sarvar Patel, Igor Faynberg, and Alec Brusilovsky. Password-Authenticated Key (PAK) Diffie-Hellman Exchange. RFC 5683, February 2010.

[8] Amjad Ali Alamr, Firdous Kausar, and Jong Sung Kim. Secure mutual authentication protocol for rfid based on elliptic curve cryptography. In *2016 International Conference on Platform Technology and Service (PlatCon)*, pages 1–7, 2016.

[9] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of aes and chosen-key distinguisher of 9-round aes-128. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 183–203. Springer, 2013.

[10] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, and Lawrence Bassham. Nist special publication 800-22: A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. *NIST Special Publication 800-22*, 04 2010.