MITRE eCTF 2023:
PARED Architecture and Design Specifications

Purdue University Team

March 9, 2023

1 Introduction

1.1 Board Overview

We use Tiva C Series TM4C123GXL boards as an embedded device for this competition, representing both the fobs and the cars in the attack/defense scenarios.

From the microcontroller's datasheet, we came across several security features which help protect sensitive data and preserve robustness of our protocols. Features relevant to our design are discussed below, focusing on the memory types available to the board.

EEPROM. The available EEPROM memory size is 2KiB, which is accessible as 512 32-bit words. The EEPROM contains 32 blocks of 16 words (64 B) each. Access and lock protection is available on a per-block basis. Interrupt support allows for write completion instead of polling.

ROM. The ROM contains the bootloader, peripheral library, AES data tables, and CRC module. Of these, we utilize the CRC module. CRC can be used to validate the correct receipt of messages sent with UART (i.e., nothing lost or modified in transit), for example.

1.2 Implementation Overview

We implemented our boards' firmware in `C`, compiled using `gcc`. The host tools, which run on a general-purpose PC, were developed in `Python`.

To deter attackers from leaking data protected in the EEPROM, we have implemented Address Space Layout Randomization (ASLR)-like mechanisms to offset addresses of relevant data in the EEPROM. Our implementation does not utilize flash memory.

2 Threat Model

2.1 Users and Devices

Host. The host computer is a general-purpose PC which builds the car and fob binaries, and also communicates with the car and fobs over a UART serial interface for many of the PARED protocols. It contains tools which will be used to pair fobs, package and enable features, and unlock a car according to the aforementioned features.

Cars. In this scenario, a car is built with a list of available features, with the user having a subset of its features packaged. With these, a user who unlocks the car with their fob (ideally paired) should be able to send packaged features for the car to recognize and enable.

Fobs. A user can have fobs which are either *unpaired* or *paired* or with a particular car (for short, UPFOB or PFOB resp.). Every car comes with an initial PFOB which can subsequently be used to pair UPFOBs. Newly-paired fobs should operate identically to the initially-paired fob as well (i.e. it should be able to pair new fobs and unlock the car).

2.2 Security Requirements

For convenience, we re-iterate the security requirements (SRs) which our secured PARED protocol satisfies.

SR1. A car should only unlock and start when the user has an authentic fob that is paired with the car.

SR2. Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car.

SR3. Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future.

SR4. Having an unpaired fob should not allow an attacker to unlock a car without the corresponding paired fob and pairing PIN.

SR5. A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer.

SR6. Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car.

See Section 4–8 for a description of how our modified PARED protocol satisfies each SR. With the threat model defined, we now describe the cryptographic primitives which provide assurances of security in this scenario.

3 Cryptographic Primitives

3.1 Authenticated Encryption

Authenticated Encryption with Associated Data, or AEAD for short, is a cryptographic primitive which guarantees both confidentiality and also integrity/authenticity for the messages it encrypts. An AEAD scheme can be defined as follows:

$\mathsf{AEAD.Enc}(K, N, A, P) \to (C, T)$**:** Authenticated Encryption (AE) takes as input a symmetric key K, a *unique* nonce N, public associated data A, and plaintext P. It outputs its ciphertext C along with an authentication tag T used to verify its authenticity.

$\mathsf{AEAD.Dec}(K, N, A, C, T) \to P$**:** Verified Decryption takes as input the symmetric key K, the nonce N and associated data A used to encrypt, the ciphertext C, and the tag T. While decrypting, it uses tag T to verify that (K, N, A, C) was unmodified. It outputs the decrypted plaintext P, or error ($\perp$) if the verification check fails.

We choose to instantiate our AEAD with the authors' implementation of the NIST Lightweight Cryptography winner, Ascon-128 [DEMS21]. Ascon uses a duplex sponge-based mode of operation. We use the recommended parameters with uniformly random 128-bit key K, $r = 64$ bit rate (i.e., the block length for processing associated data, plaintext, and ciphertext), and a capacity of $c = 256$ bits. The output has $|C| = |P|$ with a 128-bit tag T. We choose 128-bit nonces N uniformly randomly.

Ascon-128 was chosen primarily for its impressive resistance against nonce-misuse even compared to AES-GCM-SIV, small code binaries, and resistance to various side-channel attacks (SCA). Its permutations use $a = 12$ rounds for initialization/finalization and $b = 6$ rounds for data processing (i.e., encryption/decryption). See [DEMS21] for more details about how Ascon works.

3.2 Cryptographic Hash Functions

Cryptographic hash functions are a one-way function which takes an arbitrary-length input and produces a (short) fixed-length output which minimizes the number of potential *collisions* (i.e., two distinct inputs producing the same output) and cannot be efficiently inverted:

$$H : \{0,1\}^* \to \{0,1\}^\ell$$

We choose to instantiate our hash function with Ascon hash [DEMS21], which uses a sponge-based mode of operation. We use the recommended parameters with a rate/block-length of $r = 64$ bits, a capacity of $c = 256$ bits, and a hash "digest" (output) size of $\ell = 256$ bits. Its permutations use 12 rounds for both initialization/finalization and data processing.

Ascon-Hash was chosen primarily for its collision resistance (particularly, against identical-prefix attacks) due to the sponge construction, resistance to SCA, and minimal binary size.

3.3 Pseudo-random Number Generators

A pseudo-random number generator (PRNG), also known as a deterministic random-bit generator (DRBG), is a function which takes a random and ideally high-entropy *seed* as input to produce a stream of pseudo-random bits which remain unpredictable unless the seed is known. More formally, a PRNG can be defined by the following algorithms:

$\mathsf{PRNG.init}(s) \to \sigma$**:** Given a high-entropy random seed s, initialize the internal state σ.

$\mathsf{PRNG.gen}(\sigma, b, \mathsf{aux}=\texttt{NULL}) \to (r, \sigma')$**:** Given the internal state σ, #bits requested b, and optional auxiliary data aux, update the state and return a b-bit pseudo-random r.

$\mathsf{PRNG.reseed}(\sigma, e, \mathsf{aux}=\texttt{NULL}) \to \sigma'$: Given the current internal state σ, a high-entropy random input e, and optional auxiliary data aux, update the state with fresh entropy.

We implement the HMAC-based DRBG as specified by NIST SP 800-90A Rev.1 [BK15] which has been proven secure as long as the underlying HMAC primitive satisfies the properties of a keyed PRF [Hir09]. Furthermore, HMACs are well-known to be less reliant on the collision resistance of their underlying hash functions [TC11]. Indeed, the authors' official implementation of Ascon provides a combined HMAC/keyed PRF construction to use.

See the NIST SP 800-90A specifications for more details on how the PRNG works.

Regardless of how secure the PRNG algorithm is, it is also important to consider the quality of the entropy sources used to instantiate the seed. And so, we briefly discuss how we obtained sources of entropy on the board as well.

Seed generation. One of the most robust sources we could find to create the initial seed on the board was SRAM since, immediately after a power cycle, most memory blocks contain random data prior to initialization, and most blocks remain untouched by the bootloader. We construct the seed by hashing together large blocks of SRAM along with other random data we store on EEPROM as entropy sources. Together, this ensures that the seed will change unpredictably across both power cycles and resets.

Reseeding policies. After the board has had time to "warm up", we have access to many other sources of entropy. In particular, we incorporate current `CPUUsageTick` as a source of randomness along with entropy provided by UART packet sends from previous protocol executions, if available. Then, as per NIST recommendations, we hash the entropy inputs and nonce together with the current state of the PRNG to incorporate additional entropy. We deviate slightly from the original specifications, fine-tuning its usage to this specific application by enforcing a reseed after the very first 128-bit random value that's generated. We also chose a significantly more conservative reseed interval of 8192 random values.

We have vetted the quality of our on-board PRNG implementation using an implementation of the test suites defined in NIST SP 800-22 Rev. 1A [BRS+10].

In the remaining sections, we will discuss at a high-level how each of these primitives are used to implement each PARED protocol, and how this achieves the security requirements.

4 Building & Deploying Environment

4.1 Deployment

Assumption. *All features which are available to a car must be decided during the car's build process.*

The host secrets file will contain the following values:

- K_{CID}: 16 B **encryption keys** for each Car CID, used by the car and its paired fobs.
- fpwd: 16 B random **feature passwords** for each (Car CID, Feature FID).
- psalt: 26 B random **secure salts** for each Car CID, shared by its paired fobs.
- fsalts: 48 B random **secure salts** for each (Car CID, Feature FID), used by the car.

For the following protocols, we define *secure salts* to have the following properties:

1. As a random *salt*, it is used to significantly lengthen the number of bits in a hash's input, making preprocessing attacks more difficult.
2. Unlike *public* salts, these salts should always remain hidden from the attacker.
3. Also unlike *secret* salts, these salts are not securely deleted after computing the hash; instead, they are protected in the EEPROM for later use.

4.2 Car and Fob Firmware

The Car IDs CID and Feature Numbers FID are 8-bit unsigned integers. The pairing PIN PIN is a 6 hex digit ASCII string ("`000000`"–"`FFFFFF`").

Car. For each Car CID, its encryption key K_{CID}, the list of hashes for all available features' passwords $F_{\mathsf{CID}}[\mathsf{FID}] \coloneqq H\left(\mathsf{fsalt}_{\mathsf{CID}}^{(\mathsf{FID})} \| \mathsf{fpwd}\right)$, and its secure salt(s) $\mathsf{fsalt}_{\mathsf{CID}}^{(\mathsf{FID})}$ are protected in the EEPROM.

PFOB. For each fob which is initially paired to some Car CID, its corresponding encryption key K_{CID}, 256-bit pairing PIN hash $\mathsf{H}(\mathsf{psalt}\|\mathsf{PIN})$, and its secure salt psalt are written to and protected in the EEPROM. CID (public) is also contained on the EEPROM, which helps tie it to Car CID (although only the key is used for proving that it's paired; see Section 8).

UPFOB. Unpaired fobs have no special values initially written into the EEPROM beyond those mentioned in the technical specifications. See Section 6 for details on how it retrieves the above values to become a PFOB.

5 Packaging features

5.1 Secure Protocol

A packaged feature contains a feature password $\mathsf{fpwd}_{\mathsf{CID}}^{(\mathsf{FID})}$ (from host secrets) which uniquely corresponds to a Feature Number FID and Car ID CID, where:

$$\mathsf{pkg}_{\mathsf{CID}}^{(\mathsf{FID})} := \left(\mathsf{FID}, \mathsf{fpwd}_{\mathsf{CID}}^{(\mathsf{FID})}\right)$$

5.2 Analysis

Regarding functional requirements, it must take at most 1 second to package a feature. As packaging features only involves collecting input values, this should not be an issue.

Note that the scenario only requires us to implement unique message outputs for Feature Numbers 1-3 (for detecting when to release the flag). Feature Numbers 0, 4-255 are outside the scope of this challenge. Our code nominally supports these values as desired, but does not need to perform checks to recognize a feature (password) outside this required range 1-3. Thus, there are a very limited number of unique feature-car pairs to keep track of in this competition (3 features per car) and so it is feasible for a car/fob to store any of $[0, 3]$ unique feature passwords in the available space of the EEPROM. With only 6 cars in the scenario, the host secrets file will only need to store $6 \cdot 3 = 18$ feature passwords (see Section 4).

See Section 7-8 for more details on installing, recognizing, and enabling packaged features.

6 Pairing new fobs

6.1 Secure Protocol

Assumption. *When pairing a new fob, it does not inherit any features currently enabled on the paired fob. As such, the user should engage in the "Enable Feature" protocol with the newly-paired fob before attempting to unlock the car with a given feature (see Section 7).*

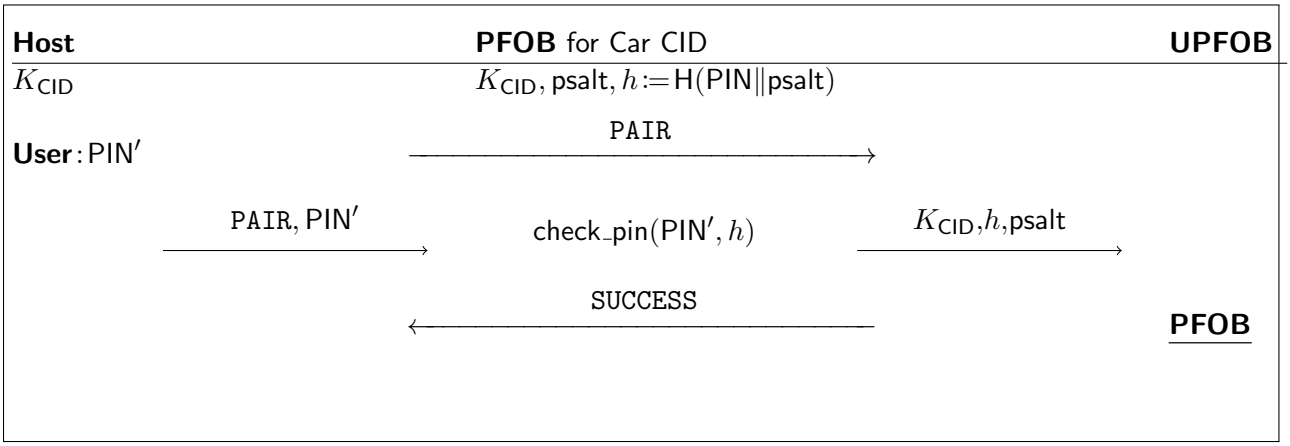

Figure 1: The pairing protocol

check_pin(PIN', h):

Access psalt from EEPROM;
$h' \coloneqq \mathsf{H}(\mathsf{PIN}' \| \mathsf{psalt})$;
if $h' \neq h$ **then**
  defense();
  return $\perp$
end

defense():

// Attack detected
Block until 5 sec. passed
// Stop protocol
return $\perp$

6.2 Analysis

An appropriate choice of algorithms and parameters for hashing (see Section 3) should run quickly enough on the board that an honest pairing with the correct PIN can be completed within 1 second. Additionally, we can delay failed protocol attempts to 5 seconds if our system detects a potential attack: here, the user submitting an incorrect pairing PIN.

This approach should be sufficient to satisfy the relevant security requirements as well:

SR1/Car1. N/A

SR2/Car2. N/A. Even though an attacker has temporary access to both a UPFOB and the PFOB, they are unable to perform the pairing operation as they do not have knowledge of the pairing PIN.

SR3/Car3. N/A

SR4/Car4. N/A

SR4/Car5. Since Car 5 explicitly does not release flags from an unlock (see above as well), revealing K to allow for a passive unlock is irrelevant. Second, rate-limiting is a crucial consideration here: if the user enters a wrong PIN, we enter defense() mode which enforces timeout even across power cycles and resets. Third, it is difficult to perform a brute-force or preprocessing attack in order to infer a valid PIN because the hash is salted with multiple bits that are unknown to the attacker (i.e., psalt). Fourth, using a hash function based on sponges instead of Merkle-Damgärd constructions (e.g., MD5 or SHA-1) avoids potential identical-prefix attacks on the hash. Lastly, SCA resistance and constant-time comparisons of Ascon-Hash also avoid leakage of psalt.

SR5/Car5. N/A

SR6/Car5. N/A

7 Enabling features

7.1 Secure Protocol

Figure 2: The enable feature protocol

See Section 5 for a description of the contents of a given packaged feature $\mathsf{pkg} \in \mathsf{pkgs}_{\mathsf{CID}}$.

See Section 8 for how an unlocked car recognizes and enables the packaged features.

7.2 Analysis

Regarding functional requirements, it must take at most 1 second to enable a feature. As this only involves receiving and storing packaged features, this should not be an issue.

This approach should be sufficient to satisfy the relevant security requirements as well:

SR5. For an attacker to enable a feature (after unlocking the car) without the manufacturer having installed it on a paired fob, the attacker must either send the correct feature password in the packaged feature during the unlock protocol or else produce a hash collision (see $\mathsf{start\_car}(\cdot)$ in Section 8). Assuming the car protects the installed feature in its EEPROM and that we use a suitable cryptographic hash function, having a 128-bit value across each feature should deter an attacker from doing so.

SR6. To use a given feature either from another car (e.g., after interception) or from one derived by a different packaged feature on the same car, an attacker would either need to have identical feature passwords for distinct car-feature pairs or else produce a hash collision. However, since the host tools choose these 128-bit feature passwords uniformly randomly and we use a suitable cryptographic hash function, the chance of this occurring is negligible. In other words, if the password for authenticating features is unique across any car and (challenge-relevant) feature, interception during the enable feature process of one car-feature pair never helps a different (car's) feature be enabled after unlock.

See $\mathsf{start\_car}()$ in Section 8 for more details about how features are recognized and enabled on the car after being installed on the fob.

8 Unlocking car

8.1 Functionality

The unlock and feature messages are as follows, where these strings will reside within the last 256 B of EEPROM as specified:

$$\begin{aligned} \forall \mathsf{FID} \in \{1,2,3\} : \mathsf{FEAT\_MSG(FID)} &:= \text{“Feature [FID] Enabled: [NAME]\0”} \\ \mathsf{UNLOCK\_MSG} &:= \text{“Car Unlocked\0”} \end{aligned}$$

Using Feature FID should be replaced with the corresponding value as appropriate. All messages fit within the 64 B allotted to each message in the last 256 B of EEPROM.

8.2 Secure Protocol

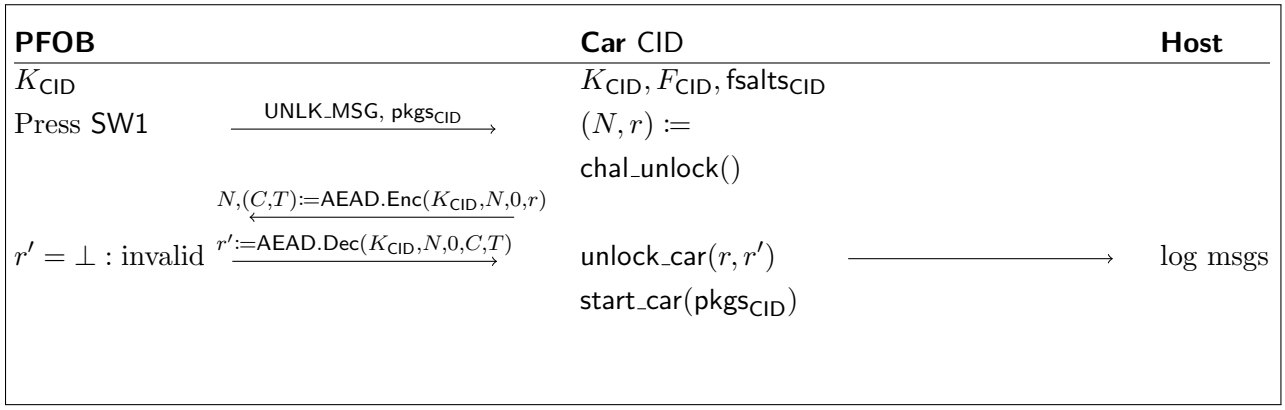

Figure 3: The unlock protocol

chal_unlock(CID, CID′):

$N :=$ PRNG.gen(128);
$r :=$ PRNG.gen(128);
return (N, r)

unlock_car(r, r'):

if $r = r'$ **then**
  print(UNLOCK_MSG)
else
  defense();
  return $\bot$
end

start_car(pkgs$_{\mathsf{CID}}$):

// Enable features
forall pkg $\in$ pkgs$_{\mathsf{CID}}$ **do**
  (FID, fpwd) =: pkg;
  Load from EEPROM: $F_{\mathsf{CID}}[\mathsf{FID}], \mathsf{fsalt}_{\mathsf{CID}}^{(\mathsf{FID})}$;
  $\mathsf{fh} := \mathsf{H}\left(\mathsf{fsalt}_{\mathsf{CID}}^{(\mathsf{FID})} \| \mathsf{fpwd}\right)$;
  if $\mathsf{fh} \neq F_{\mathsf{CID}}[\mathsf{FID}]$ **then**
    return $\bot$
  end
  print(FEAT_MSG(FID))
end

The encrypted plaintext r above is a 128-bit uniformly random *challenge* which, after the fob decrypts and presents the *response* r', proves to Car CID whether it knows the car's 128-bit key K_{CID}. The fob essentially uses its K_{CID} to authenticate as a paired fob.

8.3 Analysis

An appropriate choice of algorithms for encryption and hashing (see Section 3) should run quickly enough on the board that an honest unlock protocol will be completed within 1 second. Additionally, we can delay failed protocol attempts to 5 seconds if we detect a potential attack: here, the user failing the challenge-response that they know K_{CID}.

Furthermore, it should also satisfy the relevant security requirements:

SR1. An UPFOB does not have key K_{CID} by definition, and so a UPFOB could only decrypt to r in the challenge-response phase with negligible probability (i.e. it's forced to guess random 128-bit key K_{CID} to encrypt with). The car detects this with $\mathsf{unlock\_car}$ and aborts.

SR2. PFOB's key K_{CID} is protected in the EEPROM. Futhermore, valid transactions with the PFOB before time t should not allow subsequent transactions with the UPFOB to unlock after time t, since both the car and fob must reset after disabling the pairing fob and the challenge r is unpredictable by PRNG construction: see also SR3.

SR3. Assuming secret seed and a cryptographically-secure PRNG, an attacker observing valid transactions between the PFOB and car should be unable to predict future values of the challenge r between a UPFOB and car to forge a response with, even if previous outputs of PRNG are known, especially by frequent reseeding with new entropy.

Furthermore, using Authenticated Encryption and verifying ciphertext integrity helps a fob detect if an attacker tampered with the ciphertext, even witout associated data.

SR4. See SR1-SR2 and Section 6 for dependency on knowing PIN to obtain K_{CID}. This, in turn, is needed to pass the challenge-response and successfully unlock the car as above.

SR5. A UPFOB which obtains a packaged feature should be unable to unlock the car before the car even attempts to check whether the attacker has the correct packaged feature. Furthermore, $\mathsf{start\_car}(\cdot)$ checks that the package uses the correct $(\mathsf{FID}, \mathsf{fpwd})$.

SR6. Same as SR5, but relying on the fact that all keys K_{CID} differ for each car.

References

[BK15] Elaine Barker and John Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, June 2015.

[BRS+10] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. NIST SP 800-22 Rev. 1a.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report, Gaithersburg, MD, USA, 2010.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, July 2021.

[Hir09] Shoichi Hirose. Security analysis of DRBG using HMAC in NIST SP 800–90. In Kyo-Il Chung, Kiwook Sohn, and Moti Yung, editors, *WISA 08*, volume 5379 of *LNCS*, pages 278–291. Springer, Heidelberg, September 2009.

[TC11] S. Turner and L. Chen. Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms, Mar 2011. RFC 6151. `https://www.rfc-editor.org/info/rfc6151`.