What is credential data and parity, examining card data, further Postman usage

August 11th, 2025

ALLEGION™

PIONEERING SAFETY™

What is a credential (card or mobile or NFC)

- A credential is any object that can transmit a series of bits to a reader (or read head) in order to gain access or other privileges.
  - Traditionally:
    - Barrium Ferrite (touch, a ceramic that can be encoded with magnetic charge)
    - Magstripe (touch, not encrypted, subject to replay, and fake manufacturing)
    - Proximity (touchless, not encrypted, subject to replay attacks and fake manufacturing)
    - **Smart Card (touch/touchless, encrypted, not subject to replay attacks or fake manufacturing, contains storage)**
    - Mobile Bluetooth (touchless, requires app to integrate into phone, encrypted, contains storage)
    - Near Field Communication (touchless, no app required to integrate into phone OS (soon), encrypted, contains storage)

ALLEGION

What is a credential (card or mobile or NFC)

- A credential has a series of bits one it, typically unique, (especially for encypted technologies)
  - The bits include things like the
    - Card ID - stamped number printed on the card (or plus an offset)
    - Issue Codes (optional, usually only on magstripe)
    - Site / Faciltity Codes (optional but very common)
    - Parity bits (error checking)
    - Other fields are possible, but rarely seen
  - Typically, the read head extracts all the data and then either the reader interface or controller parses out the card ID and site code, and then the controller / panel checks parity decides whether to open the door.
  - In Engage's case, the locks simply compares the entirety of the expected data to the read data, without parsing or checking parity

ALLEGION

Card Data Encoding

- We can extract the raw data read by the lock for failed access from audits
  - A good way to learn Engage's card data encoding method
    - Unpacking is a bit easier/quicker to "get right" than packing
      - Just make sure you get a matching stamped ID

  - Let's get into the encoding scheme by examining an access denied audit
    - I will generate one live, but also available in the "sample audits.json"
  - Note "Schlage ENGAGE Credential Sort and Encryption-1.34.pdf" is the formal documentation for this

ALLEGION

Quick and dirty access denied unpacking

- 5 events for an access denied audit.
  - First just is an access denied.
- Remaining 4 events - Each has four bytes of data
  - First byte identifies the type of audit
  - The second byte is a sequence number for a multi event audit
  - Final two bytes are data

Quick and dirty access denied unpacking

Remaining 4 events - Each has four bytes of data

- First byte identifies the type of audit (invalid card data)
- The second byte is a sequence number for a multi event audit
- Final two bytes are data

- "event": " 05400000", (ID - Access Denied – Unknown card)
- “time": " 20240208095014 ", ()
- "event": "13000025", (ID & seq & card data length)
- "time": "20240208095014"
- "event": "13010628", (ID & seq & data)
- "20240208095014"
- "event": "13025e31", (ID & seq & Data)
- "time": "20240208095014"
- "event": "13030081“ (ID & seq & Data),
- "time": "20240208095014"

ALLEGION

Quick and dirty access denied unpacking - 2

- For an access denied, the ID is 0x13 – “Card not in database” The data of the first event tells us the length of the card data
  - (0x25 or 37 in decimal)
- Remaining three events give unencrypted card data in reverse order (Least significant bytes first)
  - Can have up to 4 events in the case of a 64 bit (8 byte) card data
  - Packed to nearest two-byte boundary via padding left with 0

- "event": "13000025", (Card not in database) (Card data length is 37 bits)
- "event": "13010628", (Trailing card data)
- "event": "13025e31", (Middle card data)
- "event": "13030081", (Leading card data)

- So, the card data is 0x00815e310628

ALLEGION

Quick and dirty access denied unpacking - 3

- The number stamped on my card is "200802“
  - 00110001000001100010 in binary
- I happen to know the site code is "700".
  - 001010111100 in binary
- Let’s convert 0x00815e310628 to binary and have a look. If we can align the binary strings for the card number and site code and find matches within card data from the audit, it’s a promising sign

```
Audit Data    100000010101111000110001000001100010101000
Stamped card #                    11000100000110010
Site code #           1010111100
```

Refer to documentation for a more formal approach

When *sending* card data via json door files, note the data is encrypted with the site’s symmetric key.

Retrieved audits however are not encrypted, so easier to quickly analyze

ALLEGION

Picking up where we left off

- So far, we have captured 37 bits of card data from an “access denied – card not in database” audit, and compared it to the data we would expect to see on the card used to swipeRecall the raw data retrieved from the audits was 0x00815e310618
- Expanded to binary, we get data as shown:
- 1000000101011110001100010000011000011000
- However, we are expecting 37, not 40 bits, so we can disregard the trailing “000”. 40 bits are sent only for message alignment. This leaves:

```
Data from audits 1000000101011110001100010000011000101
Stamped card #                          1100010000011100010
Site code #                1010111100
```

This was a good start, but let’s get a formal understanding of all the bits.

ALLEGION

Our 37 bit format

- Our card uses a common 37-bit wiegand card format called “37X” (allegion) or “H10304” (HID)
  - Many formats were originally created by HID, they managed card sales such that matching site/facility codes should never be issued card to different customers
    - In practice, other manufacturers can create proximity cards and ignore this, particularly when purchasing internationally. So do not rely on properly managed site codes
      - An exception are secure cards such as smart cards, which cannot be easily duplicated/forged/etc. These site codes are generally reliable
    - This format lacks an “issue code” field, but be aware many formats have this field as well. This field dates back decades to when people would lose their magstripe cards
- These rules of thumb apply to most card formats. The most common formats include 26-bit standard, HID 35-bit corporate 1000, 37X / HID 37-bit, and in our case, Allegion 48X / XceedID

ALLEGION

HID 37 bit format in detail

- Here is a complete description of the format.

Note bit positions are counted starting at zero, not one (zero indexed)

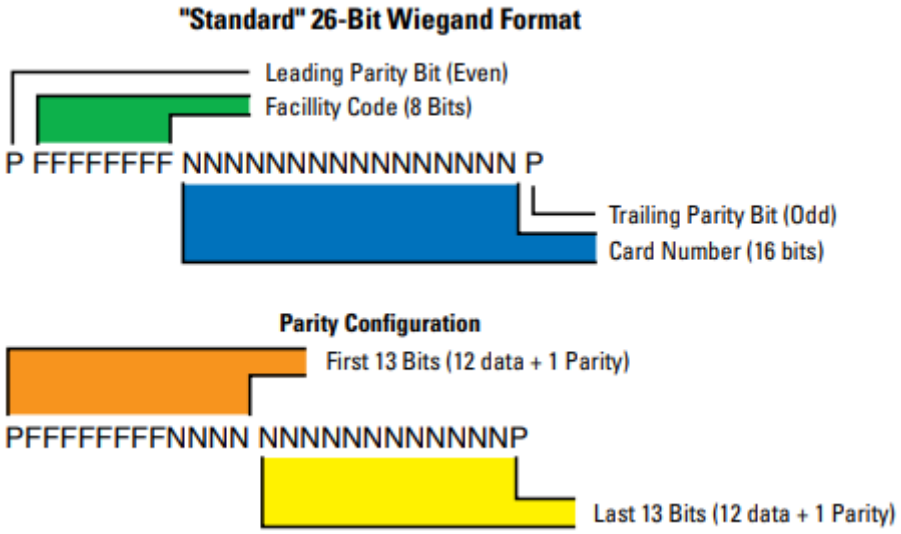
- 37 bits total indexed as 0 - 36
- Even parity mask (note leading MSB to LSB) - 19 bits position 0 to 18
- One even parity bit, located at bit zero
- Odd parity mask – trailing 19 bits starting at 18 and ending at 36
  - Note there is a one bit overlap with even mask
- One odd parity bit, located at bit 36
- 16 bits for facility / site code
- Site code start location at location 1 and ending at position 16
- 19 bits for Card ID / Cardholder ID / etc.
- Card ID start location at position 17 and ending at position 35

- Any questions on what this means?
  - (We will cover parity in a later slide)
  - See “Schlage Credential Formats-200624.pdf” for more common formats

ALLEGION

26 bit Standard format in detail

- Simpler visual tools / format builders are useful to technicians.
  - Here is a graphical representation of standard 26 bit, and software often contains “format builders” which look similar:



- However, developers, high level support, etc. should understand the fundamentals without “helper” tools

ALLEGION

Applying the 37 bit format to our data

- The full application of the format to the data (explanation to come):

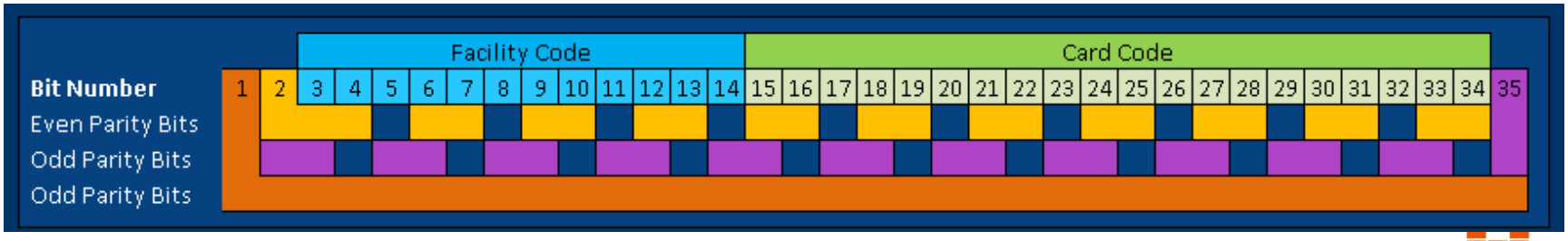
```
INDEX             0123456789012345678901234567890123456
RAW DATA          1000000101011110001100010000011000011
EVEN PARITY MASK  1111111111111111110000000000000000000
ODD PARITY MASK   0000000000000000001111111111111111111
SITE CODE DATA     0000001010111100
CARD ID DATA                      01100010000011000010
EVEN PARITY BIT   1
ODD PARITY BIT                                        1
```

- That's a lot of dense, hard to read data. Let's go one step at a time, with a brief explanation of parity

ALLEGION

Parity

- Parity is a method to perform error detection (not correction)
- “Mask” in this case refers to all bits included in the calulation
  - If we sum all bits in a parity mask the result should be even or odd, depending whether an even or odd check. The Parity bit is actively set to ensure correct even/odd-ness.
    - If not, there was an error transmitting data.
- There are a few ways to express parity masks. It’s common to assume a leading even parity bit, trailing odd parity bit, with start and end positions for their masks splitting the data in half. However, this only covers “most” formats.
  - An exception is HID corporate 1000. Three parity bits with complex masks shown below.



ALLEGION

Parity

- Parity is a method to perform error detection (not correction)
- “Mask” in this case refers to all bits included in the calulation
  - If we sum all bits in a parity mask the result should be even or odd, depending whether an even or odd check. The Parity bit is actively set to ensure correct even/odd-ness.
    - If not, there was an error transmitting data.
- There are a few ways to express parity masks. It’s common to assume a leading even parity bit, trailing odd parity bit, with start and end positions for their masks splitting the data in half. However, this only covers “most” formats.
- Another approach is to express as a bit string, where a “1” means the bit should be included in the mask, and a “0” it should not.
  - This covers non-contiguous masks, can be easily “translated” into a hexadecimal number for compactness.

Putting it all together

- Now we're ready to approach the card data from the other direction
  - We know a stamped ID and a site code, as well as our format's length and parity behavior
  - How can we turn this into a full and correct 37-bit string?
    - `?` information we don't have yet and need to figure out
    - `1` is an active parity bit in a mask.
    - `0` is zero padding inserted to fit the format
-

```
INDEX              0123456789012345678901234567890123456
RAW DATA           ???????????????????????????????????
EVEN PARITY MASK   1111111111111111100000000000000000
ODD PARITY MASK    0000000000000000011111111111111111
SITE CODE DATA      000000101011110 0
CARD ID DATA                       0110001000001100010
Even PARITY BIT    ?
Odd PARITY BIT                                       ?
```


ALLEGION

Putting it all together – even parity

- Many technicians have difficult with parity. Here how I approach it.
  - Fill in as much raw data as we know, giving us

```
INDEX              01234567890123456789012345678901234567
RAW DATA           ?0000001010111100011000100000110001 0?
EVEN PARITY MASK   11111111111111111110000000000000000000
```

- Now determine even parity bit
  - Add up all non ? bits lined up with the 1s in the parity mask. This sum is 7 (decimal) or 1 (binary).
    - Since this is an odd number, we must set the even ? to 1 so that the resulting sum will be even. Now we have more complete data:

```
New RAW DATA       10000001010111100011000100000110001 0?
```


ALLEGION

Putting it all together – odd parity

- Odd parity follows the same approach, except we manipulate the parity bit so the sum is odd.
  - Fill in as much raw data as we know, giving us

```
INDEX            01234567890123456789012345678901234567
RAW DATA         1000000101011110011000100000110001?
ODD PARITY MASK  00000000000000000111111111111111111111
```

- The sum of relevant bits is 6 (decimal) or 0 (binary).
  - Since this is even, we must set the odd parity bit, forcing the sum to 7, and thus an odd number.
    - If the original sum had already been odd, we would leave the ? bit unset as 0.
  - Our final raw data is

```
RAW DATA         10000001010111100110001000001100001 1
```


ALLEGION