DES-On-Fire: How Physical Access Control Breaks

ROMHACK2022

Markus Vervier, Yasar Klawohn

2022

X41 D-Sec GmbH

WHOAMI & WHOAMI ?

Markus Vervier

- HackDirecting stuff at X41
- Professional experience in offensive security working as a security researcher and penetration tester

Yasar Klawohn

- Working student at X41
- Studying Computer Science at RWTH Aachen

Talk Overview

- How we discovered a critical vulnerability affecting real world alarm systems (CVE-2021-34600)
- How does DESFire AES authentication work under the hood?
- How to attack it
- Tag emulation

Prior Research Examples

- *"Studying the Pseudo Random Number Generator of a low-cost RFID tag"*: Identified no weaknesses in the PRNG of MIFARE Ultralight C cards
- *"Certifiably Biased: An In-Depth Analysis of a Common Criteria EAL4+ Certified TRNG"*: Found that the majority of DESFire EV1 tags they tested were biased

Target

Telenot “complex” and “compact easy” Systems

- Alarm systems for:
  - home and small business use
  - professional environments (industrial use, retail, banks) where reliable alarm notifications and access control are crucial
- Physical access control supports NFC tags (DESFire EV1 / EV2)
- Remote access via TCP/IP using a GUI app called CompasX possible, AES encrypted connections

DESFire Applications

- Access management/smart locks
- Transport ticketing
- Closed loop payment systems
- And more..

Original Motivation

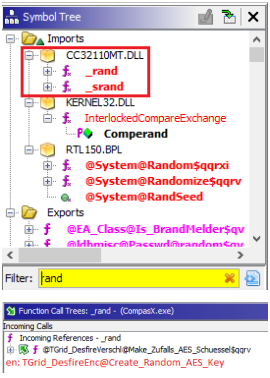
- Goal: automatically pull access logs
- Remote access via TCP/IP possible
- But: only a GUI app (CompasX) available
- Traffic is encrypted

⇒ time to fire up IDA & x64dbg to capture unencrypted traffic and implement our own compatible client application

Tools Used

- Disassembler
- Proxmark
  - Nexus 5 and other readers capable of spoofing ATQA/SAK/ATS can also be used with more development work

Wait... They Import rand()?




```
undefined * @TGrid_DesfireVerschl@Make_Zufalls_AES_Schuessel$qqrv(undefined4 param_1)

{
  time_t *current_unix_timestamp;
  int random_byte;
  time_t *in_stack_fffffff0;
  CHAR random_byte_hex [3];
  undefined4 local_8;
  int i;

  local_8 = param_1;
                    /* 0x141e0c  5242  @TGrid_DesfireVerschl@Make_Zufalls_AES_Schuess
  _memset(&ascii_key,0,0x32);
  current_unix_timestamp = _time((time_t *)0x0,in_stack_fffffff0);
  _srand((uint)current_unix_timestamp);
  i = 0;
  do {
    random_byte = _rand();
                    /* convert byte to uppercase hex */
    wsprintfA(random_byte_hex,s_%02X_00a05714,random_byte & 0xff);
    _strncat(&ascii_key,random_byte_hex,2);
    if (i < 0xf) {
                    /* append a space */
      _strcat(&ascii_key,s__00a05719);
    }
    i = i + 1;
  } while (i < 0x10);
  __lstrupr();
  return &ascii_key;
}
```

RNG Seeded With UNIX Timestamp

```
1 static uint8_t key[16];
2 memset(key, 0, 16);
3 srand(time());
4 for (int i = 0; i < 16; i++) {
5     key[i] = rand() % 0xFF;
6 }
7 return key;
```

Listing 1: AES Key Generation in ***Make_Zufalls_AES_Schluessel()*** (Simplified)

Issues

1. RNG is seeded with the current Unix Epoch time stamp
2. RNG is not cryptographically secure, each rand() call outputs a max value of $2^{32} - 1$, so keyspace bounded by $16 * 2^{32}$

Why Is rand() Imported and What Are The Implications?

The generated secrets are deployed to the central alarm unit as:

- DESFire application keys
- Encryption keys for TCP/IP remote access

...basically everything security critical!

- UNIX timestamp used for seeding

⇒ Since the keyspace is small enough, *these keys can be brute forced in certain scenarios*!

Attack Vectors

- Usual strategies to exploit weak authentication secrets:
  1. *Online*-Attack - try to authenticate live to a real target alarm system (usually slow and rate limited)
  2. *Offline*-Attack - try to break data that is encrypted/signed with the weak secret using your own cracking harness
- *QUESTION: But is an offline attack possible against DESFire EV*?*

DESFire High Level Overview

- Multiple versions: EV1 was introduced 2006, EV3 is the latest
- NFC Tags have unique, 7-byte identifier (UID) set at factory
- Can have multiple applications
  - Each capable of storing multiple keys and files
  - Authentication on application level
- Keys can be DES, 2K3DES, 3K3DES or AES128 keys
- Communication with APDUs, using ISO/IEC 14443 Type A

Preliminary Conclusions

- UNIX timestamp used for seeding, so possible keys equal the 5.1×10^8 seconds since 2006-01-01

$\Rightarrow$ If these keys are used in our lock, *can the key be brute forced*?

So Can We Actually Recover Our DESFire Tag's Key?

- Plan: Build a piece of code that explores the keyspace and find a way to recover a real, in-use, DESFire AES key such as generated with CompasX
- ... but what now? How to use this to attack the system? Even if a key is weak, we need a signal to learn when we found it

Tracing a DESFire AES Authentication

1. Reader asks for tag information and UID
2. Reader selects wrong application [seems odd, we don't know why]
3. Reader starts authentication, this fails [seems odd, we don't know why]
4. Reader selects correct application
5. **Reader starts authentication, this time it is successful**
6. Reader reads a file from Tag, contains a UID, secured with CMAC
7. Reader opens the door, if UID read from file is known

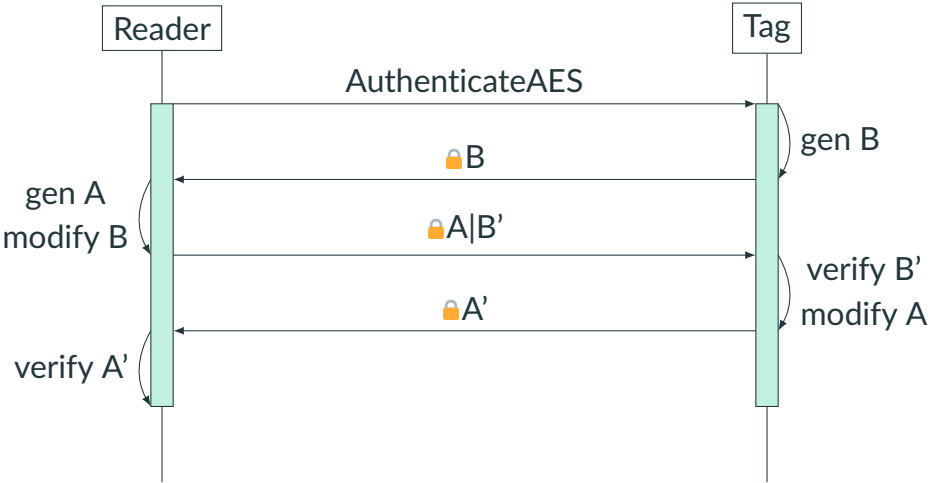
Peculiarities of the Telenot Implementation

- Reader first selects wrong application ID and tries to auth, which needs to fail
- The UID sent in the beginning is ignored, instead UID from file 0x00 is read in the end
  - ⇒ Even third party tags where the tag's own UID can't be modified, can be used for cloning
  - ⇒ Simplifies the search for vulnerable systems

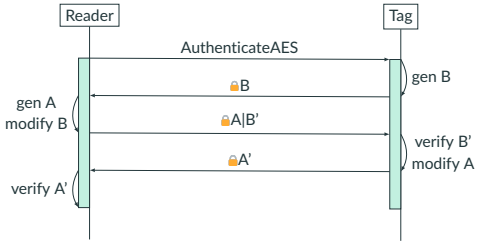
What Do We Need To Emulate a Tag?

- UID of a tag known to the system
  - Easily collected with a smartphone
- Correct AES key
  - *This is our current challenge*

DESFire AES Authentication: Overview

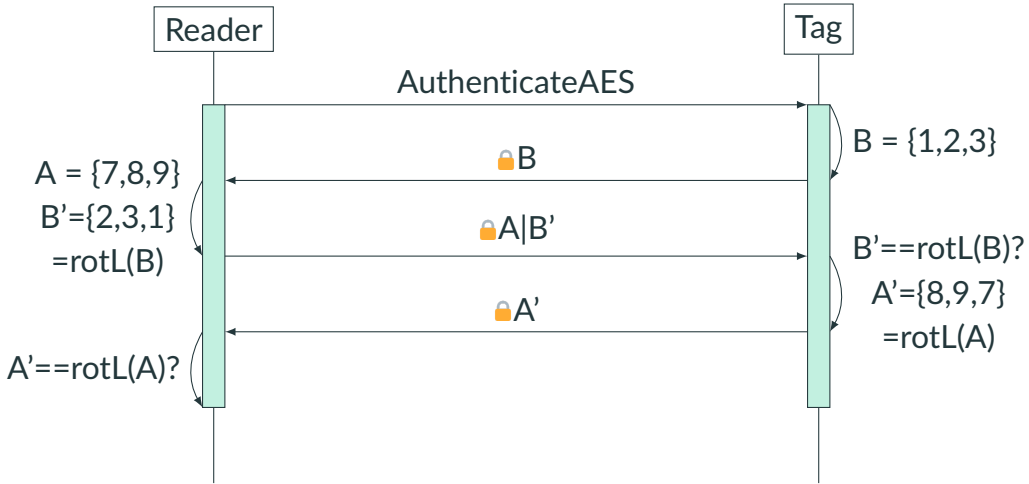


DESFire AES Authentication: Details



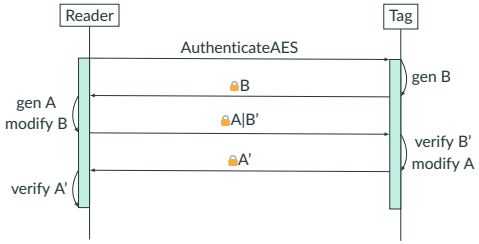
- **gen:** generate 16 random bytes
- **modify:** rotate bytes to the left
- **verify:** check if bytes were rotated correctly
- 🔒**:** AES CBC encrypted

DESFire AES Authentication: Example

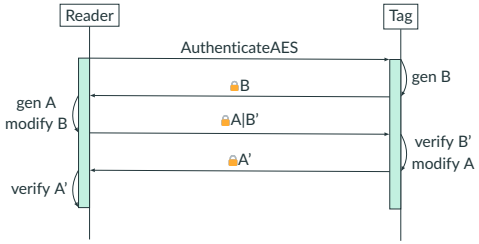


DESFire AES Authentication: Analysis

- During this authentication each side proves to the other that they know the key without ever sharing the key itself
- Can't encrypt our B with the real key without knowing it



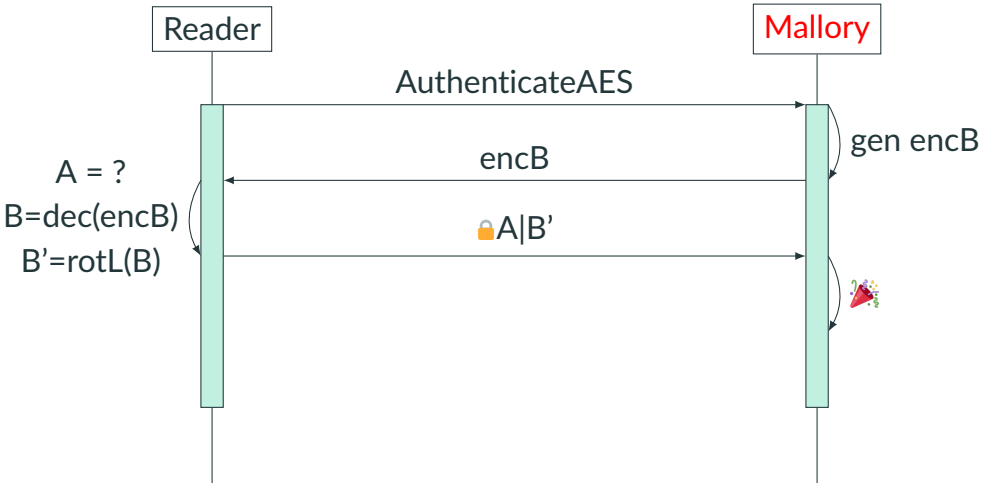
DESFire AES Authentication: Analysis



- But reader will
  1. decrypt whatever value (V) we send with the real key: `B=decrypt(V);`
  2. rotate it to the left
  3. encrypt A and B' (to enc_A_B') with the real key, and
  4. return it to us

⇒ : **try all keys until rotate_left(decrypt(V)) matches the last 16 bytes of decrypt(enc_A_B')!**

DESFire AES Authentication: Challenge Collection



Simple Brute-Forcing: Concept

```
1  for (; timestamp < start_time; timestamp++) {
2          make_key(timestamp, key);
3          decrypt(tag_challenge, 16, key, iv, dec_tag);
4          decrypt(lock_challenge, 32, key, tag_challenge, dec_lock);
5          if (dec_tag[0] != dec_lock[16+15])
6              continue;
7          for (int i = 0; i < 15; i++)
8              if (dec_tag[i+1] != dec_lock[i+16])
9                  continue;
10         // print key
11 }
```

Simple Brute-Forcing: Results

- Single threaded and starting in 2006, so pretty naive
- Still able to try all keys in a little over 3 min on a Ryzen 7 4750U

Multi-Process Brute-Forcing

- Replace OpenSSL with AES-NI
- Use `fork()`

Multi-Process Brute-Forcing

DEMO:

(`https://github.com/x41sec/poc/tree/master/CVE-2021-34600-brute-force/`)

Tag Emulation PoC

https://x41-dsec.de/static/videos/cve-2021-34600-poc.webm

Tag Emulation POC Code

- https://github.com/x41sec/CVE-2021-34600
- Iceman Fork (once merged)

Hardening your DESFire Setup

- Use a cryptographically secure PRNG *only* to generate secrets
- DESFire EV2 and EV3 support key rollover and have additional features that can make recovery from compromised keys less costly:
  `https://www.nxp.com/docs/en/application-note/AN12752.pdf`
- *We cannot break DESFire AES if a securely generated random key is used*, so make sure your systems have securely generated keys
- Rotate keys on a regular basis

Future Mitigations Against Such Attacks

- Improve the challenge-response using more modern protocols (PAKE)
- Investigate improvements of DESFire using improved algorithms such as Leak Resistant Primitive (LRP) (otherwise we might do so ;-)
- Try to use rolling keys (this might have implementation challenges)

Aside: Relay Attacks

- Command delays of up to 75ms are allowed by the our tested doorknob

⇒ Relay attacks within the same WiFi network should be possible

Conclusion

- Having secure random number generators is crucial
- DESFire can be attacked using offline attacks on captured challenges from a reader, endangering systems that use weak secrets
- Card emulation is fun..

Resources

- https://github.com/revk/DESFireAES/blob/master/DESFire.pdf
  - Awesome DESFire protocol documentation