Discovering PIN Prints
In Mobile Applications

Tomáš Rosa
Raiffeisenbank, a.s.
Definition (ATA). Let the After-Theft Attack (ATA) be any attacking scenario that assumes the attacker has unlimited physical access to the user’s smart phone.

- Imagine somebody steals your mobile phone...
- Despite being really obvious threat, it is often neglected in contemporary applications.
- By a robbery, the attacker can even get access to unlocked screen or a synced computer, hence receiving another considerable favor!
Hackers conferences are not the only place to look for an inspiration.

There are also forensic experts who publish very interesting results.

- Actually, they often take hacking techniques and refine them to another level of maturity.
- The main purpose is to prosecute criminals, of course.
- But it is just a question of who is holding the gun...
- Anyway, security experts shall definitely consider looking into forensic publications, at least time to time.
We shall assume that:

- once having unlimited physical access to the mobile device,
- the attacker can read any binary data stored in its FLASH memory.
- This also applies to certain encryption keys!

Despite not being trivial, we shall further assume this also applies to the content of the volatile RAM.
PIN Prints

- This can be any direct or indirect function value that:
  - once gained by the attacker,
  - leads to a successful brute force attack on the PIN,
  - under the particular attack scenario.

- Principally, the same applies to general passwords, too.
  - However, we can mitigate the risk by enforcing strong password policy here.
Postulate (NP3). *In the time the application process is closed (from the client perspective)*...

- *there is not enough information stored in the whole mobile device that would allow an attacker to disclose the client’s PIN successfully.*
Once Upon a Time

- There was a PKI based approach...
- ...and there was RSA private key encrypted by a derivative of a decimal PIN.
- First factor: mobile device with the encrypted RSA key
- Second factor: the PIN
- Idea: gorgeous PKI and RSA take care about the rest...
So, this was the plaintext obtained from the ciphertext under the correct PIN value:

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- N = p*q*other_factors_if_any
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d, d*e ≡ 1 (mod \(\lambda(N)\))
    prime1            INTEGER,  -- p, p | N
    prime2            INTEGER,  -- q, q | N
    exponent1         INTEGER,  -- d_p, d_p = d mod (p - 1)
    exponent2         INTEGER,  -- d_q, d_q = d mod (q - 1)
    coefficient       INTEGER,  -- q_{inv}, q_{inv}*q ≡ 1 (mod p)
    -- ...
}
```
Incorrect PIN

- The plaintext obtained for a wrong PIN can be considered as a pseudorandom sequence.
  - The **ASN.1 format rules** as well as the **algebraic relations** are probably corrupted.
- PIN searching hint – do you remember TV tuning? *Just turn the tuning knob until you get any plausible picture and sound...*
We have seen that...
- according to PKCS#1, there is a huge redundancy based on the **ASN.1 structure syntax**.
- furthermore, there is a terrible amount of algebraic-based redundancy.

So, the decimal PIN was in fact packed together with the encrypted key store.
- as a bonus gift to the diligent attacker!
Another Example

- This time, there was a PIN-encrypted symmetric authentication key.
  - Great, there is a chance to eliminate the algebraic redundancy!
  - First factor: device with the encrypted auth. key
  - Second factor: the PIN
  - Idea: HOTP and OCRA-based verification of the symmetric key (with implicit PIN check)

20. února 2013
PIN key derivation

\[ K = \text{SHA-1}(Salt_A \ || \ PIN \ || \ Salt_B)[0..15], \]
where \( Salt_{A,B} \) are device-dependent static strings.
- We shall assume \( Salt_{A,B} \) is accessible under ATA.
- Anyway, this is OK.

HOTP/OCRA key generation and encryption
- (P)RNG used for key generation.
- No usable algebraic redundancy inside. OK.
- Encrypted using AES-ECB\(_K\).
- OK. But... wait a minute – what is the padding?
Randomized Padding Structure

- **$L$-byte message:** $M = M_1 \ || \ M_2 \ || \ ... \ || \ M_L$
- **Pad to $N$ bytes:** $OT = M \ || \ PS_1 \ || \ ... \ || \ PS_{N-L}$
- **Padding string construction:** For each $PS_i$, $1 \leq i \leq N-L$, choose $j \in R \{1, 2, ..., L\}$ randomly, and set $PS_i = M_j$.

In other words, the padding string consists of randomly indexed bytes from the original message.
Again, the obtained plaintext OT’ can be regarded as a pseudorandom sequence.

- The better the encryption algorithm is, the closer to ideal random noise OT’ is... (sad, but true).

The probability of accidentally correct padding structure can be estimated as

\[ p_{\text{padding}} < \left( \frac{L}{256} \right)^{N-L} \].

**Proof.** \( PS_i = M_j \) for particular \( i \) and some \( j \) holds with \( p < L/256 \). To be a valid padding, all \( N-L \) independent equations must hold.
In one setup, we had $N = 32$, $L = 20$.

- So, there were in total 12 bytes of padding string.
  \[
  p_{\text{padding}} < (L/256)^{N-L} = (20/256)^{12} < 2^{-44}
  \]
- In other words, if we try $Q$ incorrect PIN guesses, we can expect, in mean value,
  \[
  E = Q * p_{\text{padding}} < Q * 2^{-44}
  \]
  accidentally correct padding structures.
- This directly corresponds with the number of false positives in a brute force searching for PIN.
Let the PIN be any value with a variable length of $r$ to $s$ digits.

There are

$$W = \sum_{i=r}^{s} 10^i < \frac{10^{s+1}}{9} < 10^{s+0.05}$$

possible PIN values.

For instance, $r = 4$, $s = 8$ gives $W = 111110000$.

Note that “1234” is not the same as “01234”.

20. února 2013
When brute forcing $r$-to-$s$-digit PIN, we need to verify no more than $W$ incorrect PIN values. So, we can expect to encounter, in mean value, at most

$$E = W \cdot p_{\text{padding}} < W \cdot 2^{-44} < W \cdot 10^{-13.2}$$

false positives.

In particular, **4-to-13-digit PIN** gives

$$W < 10^{13.05},$$

still leading to

$$E < 1.$$
We have seen that...

- ...given one particular encrypted authentication key, we could successfully brute force any PIN in the range of 4 to 13 decimal digits.

So, the PIN was again gracefully packed right with the encrypted authentication key.

- ...and the diligent attacker was happy again!
Be Aware of OTPs

- If the PIN is involved in OTP generation, then any OTP itself is a valuable PIN print.
  - This is true even if the OTP is also based on some symmetric key stored in the mobile device.
  - Or, we have to prove the key cannot be retrieved by respective forensic techniques.

Therefore, we shall:
- not store OTPs in permanent memory,
- wipe OTPs out of the volatile memory as soon as possible,
- regardless whether they were already used or not.
Consider the HOTP according to RFC 4226.

- This is a popular OTP generator based on HMAC-SHA-1.
- Its reference Java implementation (cf. RFC 4226), however, contains a security flaw.
- OK, it is a reference design in the sense of test vectors, which are correct.
  - On the other hand, the RFC does not warn clearly that this code shall not be used for real implementations.
  - Especially on Android, it is probably tempting to simply copy-paste the code. Do not do that!
result = Integer.toString(otp);
while (result.length() < digits) {
    result = "0" + result;
}
return result;
With each iteration, there are two new instances created:

- ("+") `java.lang.StringBuffer` or `StringBuilder` to perform the concatenation,
- ("=") `java.lang.String` to hold the result.

However, the references to the previous iteration result and to the concatenation instance are lost.

- So, we cannot wipe them even if we want to...
We have compiled the original HOTP padding procedure for Gingerbread.

To exhibit the faulty behavior, we have deliberately shortened the input integer, so we were able to see the zero-padding in action.

In particular, we set:

- $\text{otp} = 755224$,
- $\text{digits} = 9$. 
invoke-static  (p0), <ref Integer.toString(int) imp. @ _def_Integer_toString@LI>
move-result-object v0

loc_4A0:
 invoke-virtual
move-result
if-til
locret:
   return-object v0

# CODE XREF: PaddingLeak_doPad@LII+3C↓j
   {v0}, <int String.length() imp. @ _def_String_length@I>
   v1
   v1, p1, loc_4AE

# CODE XREF: PaddingLeak_doPad@LII+10↑j
new-instance v1, <t: StringBuilder>
const/16 v2, 0x30
invoke-static {v2}, <void StringBuilder.<init>(ref) imp. @ _def_StringBuilder_init@V
   v1
   v1, v0, <ref StringBuilder.append(ref) imp. @ _def_StringBuilder_append@LL>
   v1
   {v1}, <ref StringBuilder.toString() imp. @ _def_StringBuilder_toString@L>
   v0
   loc_4A0
1. Avoid encrypting keys with intrinsic algebraic redundancy.
   - If you want RSA, think twice. In principle, RSA key can be wrapped by other protocol (e.g. secret sharing), but is it really worth it? Be careful about the public key – it can also break NP3!

2. Avoid adding any “technical” redundancy.
   - ASN.1, XML, padding, ...

3. Avoid storing any PIN-based OTP.
   - Regardless whether it was already used!
Two-factor authentication resistant against After-Theft Attack is a doable adventure.

- It is a pity that ATA is still often ignored in practice.

The key idea is a distributed implicit PIN verification.

- Seems to be well-known approach.

We shall, however, carefully verify the No PIN Prints Postulate holds.

- Seems to be somehow lesser known in practice.
Děkujeme za pozornost.

Tomáš Rosa, Ph.D.
http://crypto.hyperlink.cz