Is it wise to publish your public RSA keys?

Shay Gueron[1, 2* 2**] and Jean-Pierre Seifert [2*, 3]

shay@math.haifa.ac.il jeanpierreseifert@yahoo.com

FDTC 2006, October 10, 2006

[1] Intel Corporation, IDC, Israel

[2] University of Haifa, Israel

* Applied Security Research Group

Center for Computational Mathematics and Scientific Computations

** Department of Mathematics, Faculty of Science and Science Education

[3] University of Innsbruck, Austria

Department of Computer Science

Faculty of Computer Science, Mathematics and Physics

Agenda

- RSA authentication is not “secure by definition”
- Type of faults
  - Permanent faults
  - Transient glitches
  - Bypass attack
- How easy is it to fake the modulus?
- Mitigation techniques
- Conclusions

General perception

- RSA signature verification process is inherently secure because it only deal with public information.
- Wrong…
- Even if the signature process that uses secret keys is secure
- Attackers’ motivation:
  - BORE-defeat the authentication
  - Use only the authentication device

Type of faults

- Permanent faults
  - ➢Cheating the public key validation phase
- Transient glitches
  - ➢Cheating signature validation by forging the modulus
- Bypass attack
  - ➢Cheating by making the device skip critical decision points in the authentication flow or skip look in exponentiation to make it trivial

Threat model

- An adversary has a device which contains an RSA public key (hash), stored in protected memory (e.g. in ROM).
- The public key value is known to the adversary.
- On message-signature input, the device transfers the value of key hash from protected memory to the ALU and proceeds to check if input public key is valid and if the signature is valid.
- The adversary can induce data faults (transient fault), or
- The adversary can change (for e.g. through a FIB machine) dedicated bits of N (or hash), or
- The adversary can induce fault that would cause the device to skip some instructions
- The adversary uses <u>only on the authentication device</u>.
  - The signature process is assumed to be safe (different from threat model in Brier et al CHES 2006)

Standard RSA signature authentication flow

Input: M, S, N, E

ROM storage: hN (=HASH (N))

Authentication flow

1. **Phase 1: Validate N** (validate the public modulus)
    a. Compute T = HASH (N)
    b. Compare T to hN

Branch point 1: N is considered authentic if and only if T=hN

2. **Phase 2: Validate M** (validate the message)
    a. Compute hM = HASH (M)
    b. Compute A = S^E mod N by the following steps:
        i. A = S
        ii. Perform a sequence of modular squares (A $\rightarrow$ A^2 mod N) and modular multiplications (A $\rightarrow$ A*S mod N), according to the value of E
    c. Extract the expected hash value ExpectedhM from A
    d. Compare hm to ExpectedhM to hM.

Branch point 2: M is considered authentic if and only if hM= ExpectedhM
Branch point 3: Accept message M if and only if both N and M were validated.

Transient glitch to fake modulus

<u>Offline:</u>

- The adversary prepares a message and signs it with a fake modulus (e.g., prime) of his choice.

<u>Online:</u>

- After the device has checked input N against expected hash value.
- The adversary induces random transient fault to change original N to fake Ň
- Message would authenticate…
- To do this in practice Ň and N need to differ in a small number of bits

How many bits to play with?

Example

- **Random string (T) of 1012 bits, multiplied it by 2006 (current year)**
- **N=P*Q**
- **P and Q are the two next primes exceeding 2006T.**
- **Tampering with the 10 least significant bits of N,**
- **Screen (offline) for Ň = i + N- (N mod 512), for i=1, 2, …, 511**
- N=1B10FAB24763BD3C20A4DA2464B68ADB36A2A39FFEECF6A5453DA269CCE5870F3A309C1211131977AA9D523
  18BD437B3967FDEF5B4D76F5453263222BAAA19E1B243267162BFF9C9907A175271435D38EE7068D1CF020E2DC0
  D28087941F59B382D9EBAFACA46FD9433D9D6E2AC97BDC2C793FB744C1EB01D840B2F230E713431E93B4385354
  589DEA67C559FE6AF6550863446FA941B62EC6313ECC4B09A65A201FD61113DE425602DACCE8E32A2A75E2A6C
  D8A80A5F42FCA7699AEA53D64BB43898C5E12509A72AE6AF60A9A9CC77AC7C539EE8BEC9A4FD587CE7ED0148
  FFE25AA1F2A1ABF073CE84A0E11F2EEBDE48AFCEF1EAACED6F2ACE110DEE**DD5**

- **a prime Ň was found for i=35**
- Ň=1B10FAB24763BD3C20A4DA2464B68ADB36A2A39FFEECF6A5453DA269CCE5870F3A309C1211131977AA9D523
  18BD437B3967FDEF5B4D76F5453263222BAAA19E1B243267162BFF9C9907A175271435D38EE7068D1CF020E2DC0
  D28087941F59B382D9EBAFACA46FD9433D9D6E2AC97BDC2C793FB744C1EB01D840B2F230E713431E93B4385354
  589DEA67C559FE6AF6550863446FA941B62EC6313ECC4B09A65A201FD61113DE425602DACCE8E32A2A75E2A6C
  D8A80A5F42FCA7699AEA53D64BB43898C5E12509A72AE6AF60A9A9CC77AC7C539EE8BEC9A4FD587CE7ED0148
  FFE25AA1F2A1ABF073CE84A0E11F2EEBDE48AFCEF1EAACED6F2ACE110DEE**C23**

Almost perfect match...

- N=1B10FAB24763BD3C20A4DA2464B68ADB36A2A39FFEECF6A5453DA269CCE5870F
- Ň=1B10FAB24763BD3C20A4DA2464B68ADB36A2A39FFEECF6A5453DA269CCE5870F

```
3A309C1211131977AA9D52318BD437B3967FDEF5B4D76F5453263222BAAA19E1B243
3A309C1211131977AA9D52318BD437B3967FDEF5B4D76F5453263222BAAA19E1B243

267162BFF9C9907A175271435D38EE7068D1CF020E2DC0D28087941F59B382D9EBAF
267162BFF9C9907A175271435D38EE7068D1CF020E2DC0D28087941F59B382D9EBAF

ACA46FD9433D9D6E2AC97BDC2C793FB744C1EB01D840B2F230E713431E93B438535
ACA46FD9433D9D6E2AC97BDC2C793FB744C1EB01D840B2F230E713431E93B438535

4589DEA67C559FE6AF6550863446FA941B62EC6313ECC4B09A65A201FD61113DE425
4589DEA67C559FE6AF6550863446FA941B62EC6313ECC4B09A65A201FD61113DE425

602DACCE8E32A2A75E2A6CD8A80A5F42FCA7699AEA53D64BB43898C5E12509A72A
602DACCE8E32A2A75E2A6CD8A80A5F42FCA7699AEA53D64BB43898C5E12509A72A

E6AF60A9A9CC77AC7C539EE8BEC9A4FD587CE7ED0148FFE25AA1F2A1ABF073CE8
E6AF60A9A9CC77AC7C539EE8BEC9A4FD587CE7ED0148FFE25AA1F2A1ABF073CE8


4A0E11F2EEBDE48AFCEF1EAACED6F2ACE110DEEDD5
4A0E11F2EEBDE48AFCEF1EAACED6F2ACE110DEEC23
```

Mitigation strategy

- **Against permanent faults**
  - **Make sure the attacker does not know what the device compares to when it validates N**
- **Against transient glitches**
  - **Interleave values that depend on authentic modulus into the modular exponentiation (mask the base)**
- **Against bypass attacks**
  - **Embed a running counter starting from a random (or secret) point**
- **Steps assume that**
  - ***it is physically possible to safely embed a secret into the device, which cannot be read out (even by reverse-engineering methods) ``too" easily***

RSA signature authentication flow, with countermeasures against fault attacks

Input: M, S, N, E

Protected ROM storage: hkN (=HASH (N || k))   (k is short a secret key)

Counter:

Start counter from unknown initial value
Propagate per instruction

Authentication flow

1. **Phase 1: Validate N** (validate the public modulus)
    a. Compute T = HASH (N || k)
    b. Compare T to hkN

Branch point 1: N is considered authentic if and only if T=hkN

2. **Phase 2: Validate M** (validate the message)
    a. Compute hM = HASH (M)
    b. Compute A = $(S+U)^E$ mod N   (U = unpredictable multiple of authentic N)
    c. Extract the expected hash value ExpectedhM from A
    d. Compare hm to ExpectedhM to hM.

Branch point 2: M is considered authentic if and only if hM= ExpectedhM
Branch point 3: Accept message M if and only if both N and M were validated, and final counter value is the expected one.

Take-home lesson

- Signature verification is not inherently secure under a fault attacks threat model
- Countermeasures are needed
- Whenever possible - it is of great advantage to keep public RSA keys secret

Thank you for your attention!

Questions