Network Security - Assignment 1

Kevork C. Dounato & Kristoffer P. B. S rensen

04-10-2022

Contents

Task 2 - RSA attack . . . 1
The assigned task: . . . 1
Solving the task . . . 1
The Attack . . . 2
Proving the attack works . . . 2
How to run the attack . . . 3
Task 3 - Implement RSA PSS . . . 3
The assigned task . . . 3
The solution . . . 3
The implementation . . . 3
Testing methodology . . . 4
Repository . . . 4

Task 2 - RSA attack

The assigned task:

> *Your task is to obtain a signed message stating you got a 12. To this end, you can use the given API and the malleability properties of plain, textbook RSA.*

Solving the task

Plain textbook RSA has the property that it is *malleable* which means that the product of two messages m_1 and m_2 encrypted under the same key (e, N) is equal to the product of the two messages encrypted. Written out more formally the equality is[1]:

$$E_{e,N}(M_1) \cdot E_{e,N}(m_2) = (m_1 \cdot m_2)^e \bmod N = E_{e,N}(m_1 \cdot m_2)$$

This property makes it possible to compose multiple messages as one, and later decompose it into its constituent messages. It is this property that make the requested attack possible. The desired message to send to the server is `"m = You got a 12 because you are an excellent student! :)"` The webserver has 4 HTTP GET routes:

1. `/pk`

Responds with the public keypair (e, N)

2. `/quote`

[1] Taken from lecture slides 2 in week 3.

Responds with a random quote if client sents a cookie with a message `m` and a valid signature s that signs the message given. Otherwise return an error.

3. `/grade`

If the client sends a cookie with a message `m` and a valid signature s that signs the message given. The server will reply back with the message you gave it, indicating that you managed to change your own grade as a student to 12.

4. `/sign_random_document_for_students/<data>`

Returns the signature of `<data>`, where the signature is calculated with the server's private key d. If `<data>` contains one of the following sub strings `['grade', '12', 'twelve', 'tolv']` an error message is returned instead.

The Attack

With this attack surface it is not possible to complete the task by just requesting

`/sign_random_document_for_students/m` as m contains the sub string `12`. Instead we have to do the following:

1. Get the public keypair (e, N) from the server.
2. Construct a modified message m' that can be signed correctly by the server and return a signature s'. m' is generated by multiplying m with a random encrypted message $c = r^e \pmod N$, $m' = mc \pmod N$. Since the RSA algorithm is malleable, the random encrypted message c can be removed from s' by multiplying with R, where $rR \equiv 1 \pmod N$. This will effectively give a new signature s that signs the remaining message component m of m'. Even though we as the attacker do not know the private key d.
3. Send the cookie `{'cookie: {'msg': m, 'signature': s}}'` to the server on route `/grade` to give ourselves a grade of 12.

Proving the attack works

With RSA we have the three integers N, e and d which is the modulus, public key exponent and private key exponent respectively. We have a message m, which is a nonnegative integer where $0 \leq m < N$, that we want generate a signature of, but we cannot do it directly. If we generate a random message r, where $0 \leq r < N$, and encrypt it

$$c_r = r^e \pmod N$$

We can generate a composite message as

$$m_c = mc_r \pmod N$$

m_c is allowed to be signed

$$s_c = m_c^d \pmod N = (mc_r)^d \pmod N = (mr^e)^d \pmod N$$

$$= m^d r^{ed} \pmod N = m^d r \pmod N$$

Exponentiating by e and d under $\pmod N$ are each other's inverses so they cancel each other out; $a^{ed} \pmod N = a \pmod N$.

By multiplying s_c with the *modular multiplicative inverse* of r we can cancel it out. The modular multiplicative inverse R is defined as the congruence

$$rR \equiv 1 \pmod{N}$$

R can be found with the *extended Euclidean algorithm.*

If we do this we get a new signature s

$$s = s_c R \pmod{N} = m^d rR \pmod{N} = m^d 1 \pmod{N} = m^d \pmod{N}$$

If we verify the signature s we get

$$s^e (\text{mod } N) = (m^d)^e (\text{mod } N) = m^{ed} (\text{mod } N) = m (\text{mod } N)$$

This shows that signature s found without knowing the private key d correctly signs the message m.

How to run the attack

```
./exploiting_textbook_RSA_signatures.py <webserver-url>:<port>
# example
# ./exploiting_textbook_RSA_signatures.py https://cbc-rsa.netsec22.dk:8001/
```

Task 3 - Implement RSA PSS

The assigned task

> *Implement a fix for this security problem by means of the standardized RSA-PSS signature algorithm.*

The solution

The Probabilistic Signature Scheme (PSS) will be used to solve the malleability problem of textbook RSA. RSA-PSS provides a probabilitic signature when using at least one salt byte. Furthermore PSS hashes the data in the process and provides a signature that is only as long as the digest size of the hashing algorithm. This limited signature size is a big usability enhancement, as it avoids sending a signature that would otherwise be the the size of the data, which could be really large in some use cases.

The implementation

Our implementation is based directly upon `RFC 3447`, where we implemented the signature generation operation as described in `8.1.1` and its verification operation as described in `8.1.2`. However, the singing and verification functions further depend on the following operations to be implemented.

Both depend on the following data conversion primitives:

- `4.2 OS2IP` Converts octet string to integer
- `4.1 I2OSP` Converts integer to octet string

The signature generation operation needs dependencies:

- `9.1.1 EMSA-PSS-ENCODE` Encoding operation
- `5.2.1 RSASP1` Signature primitive

The verification generation operation needs dependencies:

- `9.1.2 EMSA-PSS-VERIFY` Verification operation
- `5.2.2 RSAVP1` Verification primitive

Data conversion primitives

`OS2IP`: This primitive simply converts a non-negative integer into an octet string.

`I2OSP`: This primitive converts the octet back into a non-negative integer.

Signature primities

`RSASP1`: This primitive signs the message by applying $m^d \pmod N$, where m is the message, d is the private key and N is the modulus component.

`RSAVP1`: This is the verification primitive to the public key. It applies $s^e \pmod N$, where s is the signature, e is the public exponent of the private key, and N is the modulo

Encoding and Verification operations

`EMSA-PSS-ENCODE`: This operation hashes and applies the probabilistic encoding of the message. It is parameterised with a hash function, salt length and MGF[2].

`EMSA-PSS-VERIFY`: This operation verifies the signature. It is parameterised with a hash function, salt length and MGF.

Default hash function

`RFC 3447` provides optional arguments that have defaults. For the hash functions we chose our default to be SHA256, because it generates a 256-bit hash values compared to SHA1's 160-bit hash value.

Testing methodology

In addition to testing the `sign()` and `verify()` functions with each other, we also cross-test them with `PyCryptodome`'s own PSS sign and verify functions. This verifies each function to be in compliance with the `RFC 3447` specification. So a message signed with our `sign()` function should be verifiable with `PyCryptodome`'s verify function and vice versa. The only difference is that `PyCryptodome`'s sign function deviates slightly from the specification as it does not hash the data or message. Therefore we pre-hash the data before sending it for signing with `PyCryptodome`, while we provide the data as is and directly to our own `sign()` function. We test the functions in two ways, the first is by signing and verifying the string `"It's me Mario!"`, and in the other, we import the image `spiderman.jpg` to be signed and verified.

As can be observed in the output, both our `sign()` and `verify()` functions pass all tests. This can be observed by running the `rsa-pss.py` file with `python3 rsa-pss.py` after installing dependencies `pip3 install pycryptodome ansicolors`.

Repository

`https://github.com/KevorkC/netsec_assignment1`

[2]Mask Generating Function