COL759 Cryptography Semester 1, 2023-24

Problem Set 4(b)

Instructor: Venkata K *Due Date: 28 November 2023*

Instructions:

- Assignment must be done in groups of size at most 2. Each group must submit one pdf on Gradescope, and mention the partner's name (if any).
- This is the second part of Assignment 4 (worth 25 marks), consisting of coding questions. All questions are compulsory in this part. The deadline for this part is **28th November 2023**.
- You have to submit your description for these questions on Gradescope in `Assignment 4(b)`.

1. **RSA with a low-entropy prime generator** (5 marks)

   Ever since the start of the course, you have learned that you should never implement crypto on your own. In this problem, we will look at a case where the implementations seemed to have been correct, yet a devastating attack unfolded due to the manner of the implementation.

   Here we describe the setup: The system used RSA for encryption, and the keys were generated when the system was started up. However, the system had low entropy at boot and this caused the first prime sampled for the RSA key to originate from an extremely low-entropy distribution. This vulnerability allowed attackers to break RSA by factoring N in the public key.

   Your task is to figure out how sampling primes from a low entropy source could lead to breaking RSA. You have been given access to the following interface, and your task is to code an attack that successfully decrypts the message intended for some recipient in the system:

   - `restart_system()`: simulates rebooting the system and returns the freshly sampled public key and a ciphertext intended for the current system as $((N, e), ct)$. You can call this at most 200 times.
   - `decrypt(ct, N, d)`: Given a ciphertext ct and secret key (N, d), it returns the underlying message obtained on RSA decryption.

   Note: These functions are implemented in a file `mrsa.py`, which are imported in the `attack.py`. This file is not given to you.

   Files Given: You are given the following Python file on Teams (`COL759_A4_Coding1.zip`):

   - `attack.py`: You are required to implement the attack function in this file. You will be given access to a function `restart_system()` which returns a tuple $((N, e), ct)$ on being called. You can call this function at most 200 times and it returns (N, e) which is the new public key of the system on restart and also the ciphertext ct of an encrypted message addressed to this system.

     Your task is to generate the keys multiple times and use the fact that the prime is sampled from a low entropy distribution to break the key and return the message hidden by ct in the latest restart.

   Instructions:

   - You are only required to submit `attack.py` file on Gradescope in the `Assignment4 Coding1` assignment with your implementation of `attack`. You don't need to submit any other files.
   - All test cases are randomly generated, and all test cases have equal credits; you should be able to find the number of test cases that your code passes on Gradescope.
   - Provide a high-level description of your approach in the pdf submission.

2. **Another Attack on RSA Signatures** (8 marks) replace bytes with bits

You will see/have seen the Bleichenbacher attack on RSA signatures in one of the lectures. Here we present another similar attack that implements rsa-based signatures with a similar vulnerability.

Signing with RSA essentially means, using the secret d to first compute m^d and then verification proceeds with using e. Using RSA with $e = 3$ is used popularly since it is efficient to compute cubes rather than larger powers for the public side of operations.

Signing usually works as follows: pad the message M such that the formatted string is

$$\texttt{0x00 0x01 0xff 0xff} \ \ldots \ \texttt{0xff 0x00} \ [M].$$

This padded string is then used to produce the signature σ.

Verification proceeds by computing $\sigma^3 \bmod N$. The obtained string should then be checked for the correct padding.

However, in various implementations, some checks have been implemented incorrectly. For example, in this assignment problem, this check is implemented as follows:

(a) check that the string must start with 0x00 0x01

(b) the string must end with 0x00 followed by the message. Since the signature is usually processed in a streaming fashion, find the first 0x00 after the second byte, and use the remaining bytes as the message.

Note that the actual check does not involve checking if the bytes in the middle are 0xff. The way these checks are implemented in the library actually leads to a vulnerability.

Your task is to exploit the fact that $e = 3$ and the specific checks that have been implemented to forge a signature for a given small message.

The attack follows simple steps:

(a) first try to construct a string which when cubed produces $x < N$, where x has suffix 0x00 $[M]$.

An example to get an idea about doing this step is described in Figure 2a.

Try finishing this to find a valid value for s. Here, s is the string we are trying to create, c is the cube of s and tgt is the required target suffix of the forgery which we want to create.

(b) Next, you need to ensure that the length of this string is such that it produces the first two bytes as 0x000x01. This can be done by making your signature of appropriate length.

Hint: try taking numbers around the cube root of 0x000x010x00...0x00.

(c) Satisfying the first and second check will give you 80% credit. Additionally, you should also try to pass the 3rd check, where you avoid any 0x00 from being present in the middle part of the ciphertext. This can be done by generating random signatures(by changing some of the middle bits of your signatures) until this property is satisfied. If all of the above checks get passed then you will get full credit.

```
s:     0000000001  <- Initial s
c:     0000000001  <- Bit 0 an 1 match, skip
tgt:   1010101101  <- Target suffix of c

s:     0000000001
c:     0000000001  <- Bit 2 doesn't match, flip in s
tgt:   1010101101

s:     0000000101
c:     0001111101  <- Bit 3 matches, skip
tgt:   1010101101

s:     0000000101
c:     0001111101  <- Bit 4 doesn't, flip in s
tgt:   1010101101

s:     0000010101
c: 10010000101101  <- Bit 7 doesn't match, flip in s
tgt:   1010101101

s:     0010010101
c: ...00110101101  <- Bit 8 doesn't match, flip in s
tgt:   1010101101
```

Figure 1: Finding a string whose cube produces a number with suffix 0x00 [M].

Files Given: You are given the following Python files on Teams (`COL759_A4_Coding2.zip`):

- `rsasig.py`: It has two functions: `keygen` and `verify`. The `keygen` function is used to generate all the parameters required for this signature scheme. The `verify` function takes as input a signature σ, message `msg`, and the verification key (N, e), and checks whether σ is a valid signature on `msg` or not with all the above-mentioned checks.
- `attack.py`: You are required to implement the attack function in this file. You

will be given $N, e = 3$ (fixed) and msg as parameters to the function attack. The modulus $N(= pq)$ is 2048 bytes long, where p and q both are 1024 byte primes, $e = 3$ is fixed, msg is 1-10 bytes long with the first and last bits guaranteed to be both 1. It is supposed to return a valid signature σ for the given message msg, as per the signature scheme. This means: calling $\mathsf{Verify}(\sigma, \mathsf{msg}, N, e)$ must return 1 or 2. A return value of 1 means 80% credit and a return value of 2 means 100% credit. Any other return value means that the attack has failed and 0 credit is awarded.

Instructions:

- You are only required to submit `attack.py` file on Gradescope in the `Assignment4 Coding2` assignment with your implementation of `attack`. You don't need to submit any other files.
- All test cases are randomly generated, and all test cases have equal credits; you should be able to find the number of test cases that your code passes on Gradescope. As explained above, there are multiple checks in the verification; if you get check 1 and check 2 passed, then it will give you 80% of the marks for that test case, and if you get all checks passed, then you will get 100%.
- Provide a high-level description of your approach in the pdf submission.

3. **Attack on RSA PKCS Padding: Bleichenbacher's attack** (12 marks)

This is a Chosen Ciphertext Attack against protocols based on the RSA Encyption Standard PKCS#1 v1.5

When encrypting something with RSA, using PKCS#1 v1.5, the data that is to be encrypted is first padded, then the padded value is converted into an integer, which is then followed by RSA modular exponentiation (with the public exponent e). On decryption, the modular exponentiation of the ciphertext (with the private exponent d) is applied and the padding is removed to recover the message.

The core of this attack relies on a padding oracle: given a sequence of bytes of the same length as the ciphertext, tells whether the decrypted message would have a proper padding or not.

Padding Scheme: Let N be the public modulus for the RSA and e be the public exponent. Let M be the sequence of bytes which is to be encrypted. The PKCS#1 v1.5 padding scheme adds some bytes to the left, so that the total length of the padded message is equal to that of N. Let k be the byte length of N.

A properly padded message M has the following format:

$$\texttt{0x00 0x02 [PS] 0x00 } [M]$$

(a) The sequence of bytes begins with a zero byte, which is followed by a byte of value 2

(b) `PS`: The padding string, which is a sequence of random bytes, which cannot have a zero byte

(c) Then a byte of value 0, followed by the message M itself

(d) Restrictions:

i. The padding string `PS`, has $k - 3 - |M|$ bytes

ii. $|M|$ cannot exceed $k - 11$ bytes, which means that the byte length of `PS` is at least 8

iii. The encryption block 00 || 02 || `PS` || 00 || M is formed, converted to an integer x and the ciphertext is $c = x^e \bmod N$.

Attack: The attacker has access to the encrypted message or the ciphertext c. To decrypt the message, the attacker makes use of the padding oracle as follows:

- The adversary knows that $c = m^e \bmod N$, where e is the public exponent, and m is the padded message.
- The attacker will initiate many requests to the oracle.
- For each request, the attacker generates a value s and sends, a value $c' = c \cdot s^e \bmod N$.
- Most of the times, the padding oracle will return a 'bad padding' error. However, with a low but non-negligible probability, it will not return a padding error. Using this information from the oracle and the value of s, attacker can narrow down the search for m in a specific range.

- Refer to the paper by Bleichenbacher to learn about the attack. There are also several expository articles on Bleichenbacher's attack. If you refer to any of them, please cite them appropriately in your submission.

Files given: Since the messages and the cipher-texts are arbitrary sequences of bytes, we represent each of them by a list of integers within the range [0, 255]. You are provided with the following files on MS Teams (`COL759_A4_Coding3.zip`):

- `rsa.py`:
  - It contains a function called `check_padding`(c) : Takes the ciphertext and decrypts it to get the padded message. It checks whether it has a valid padding i.e, it should have the first byte as 1, the second byte as 2 which is followed by a sequence of $(k - 3 - |M|)$ non-zero bytes and finally 0. If any of these is violated, outputs `False`, and outputs `True` otherwise.
  - It has a function called `encryption(msg)`, which takes the unpadded messaged (list of integers) as input, internally pads it and returns the ciphertext. You can use it to check the correctness of your code.
- `attack.py`:
  - You are required to implement your attack in this file. The function to be implemented is `attack(cipher_text, N, e)`
  - It is supposed to take a ciphertext, the public modulus and the public exponent as input, and the return the original message (as a list of integers)
  - You are allowed to make calls to `check_padding()` from `rsa.py`

Instructions:

- You are only required to submit `attack.py`, with your implementation of `attack()`. You don't need to submit `rsa.py`
- Submit the `attack.py` file on Gradescope in the `Assignment 4 Coding3` assignment
- During grading, we would change the keyPair so you cannot simply decrypt the ciphertext.
- Provide a high-level description of your approach in the pdf submission.