BabyAuth writeup

This writeup was generated by Google Translate from the original Chinese version, sorry for inconvenience.

This task mainly examines some details of the actual use of elliptic curve signature and block cipher CBC working mode. In the question, a national secret algorithm library (https://github.com/duanhongyi/gmssl) on Github is used to implement the national secret algorithm SM2 , SM3, SM4.

Description

The server will generate a string of `session id` every time it connects, and the corresponding administrator user name is $\text{'admin'}||\text{session id}$. The two functions of the topic are respectively for registration and login:

- When registering, the user provides a username, which cannot contain the administrator username. Then the server uses to generate a json string, where `rwx` is set to False and `id` is set to the username. Then use SM2 to sign it, SM3 to calculate its digest, connect the three in series and encrypt it with SM4 of the CBC model to obtain a token, and return the token to the user.

- When logging in, the user provides a token. After decryption, if both the digest and the signature match, the login is successful. If `rwx` is True and `id` is the username of the administrator, the flag is returned.

Problem Solving Ideas

First briefly describe the two core points used in this question:

- https://github.com/duanhongyi/gmssl/blob/master/tests/test_sm2.py#L22, SM2 calls the `sign()` function by default when signing, and the message passed in by this function should be a digest value. The parameter in this question is the message plaintext. If two messages m_0, m_1 satisfy $m_0 \equiv m_1 \pmod{n}$, where n is the order of the curve, then the signature for m_0 obviously applies to m_1 .
- https://github.com/duanhongyi/gmssl/blob/master/gmssl/func.py#L16, the PKCS7 padding used by SM4 does not verify the validity of the padding when removing the padding (assuming that the last plaintext byte is x, it should be satisfied that x is

between 1-16, and the value of the last x bytes are all x) .

In the end, we must construct an SM4 ciphertext. The decrypted result contains the complete administrator username, but we cannot directly use the name when registering. In order to bypass this limitation, consider adding the original json header `{"rwx ": False, "` is compressed into `{"rwx":1,"id":"a`, so that the subsequent part of the administrator user name except the first character can be registered as the user name to get the corresponding SM4 The ciphertext, and then modify the IV so that the "id" field parsed from the first two ciphertexts is exactly the username of the administrator. Next, we need to construct a legal SM2 signature. Using the first vulnerability, we can find out that the json corresponding to another legal username just collides with our target json, that is, the two are equal under the modulo n. We can pre-calculate the final SM3 digest, and note that it is in hex form, so it can be directly registered as a user name to obtain the corresponding SM4 ciphertext. Finally, we need to solve the unpadding problem. The second vulnerability above allows us to remove plaintext blocks of arbitrary length.

Signature Collision

This step can be done by lattice reduction, here is a simplified example. Suppose, we need to construct a string like `0123456789abcdef????` (the last 4 bytes are controllable and lowercase letters, and the values are respectively recorded as c_1, c_2, c_3, c_4; the first few bytes have been Knowing that the corresponding value is recorded as k), the value of the string $x := k + c_4 + 2^8 c_3 + 2^{16} c_2 + 2^{24} c_1$ satisfies $x \equiv t \ pmod n$.

Consider constructing lattice $\mathcal{L}(\mathbf{B})$ as follows

$$\mathbf{B} := \begin{bmatrix} Cn & & & & & \\ C & 1 & & & & \\ C2^8 & & 1 & & & \\ C2^{16} & & & 1 & & \\ C2^{24} & & & & 1 & \\ C(t-k) & D & D & D & D & 1 \end{bmatrix},$$

where C is a large enough number, and D is set to a number close to the average value of lowercase letters of 110, then there is likely to be a short vector
$\mathbf{v} := (0, c_4 - D, c_3 - D, c_2 - D, c_1 - D, -1) \in \mathcal{L}(\mathbf{B})$, so you can use LLL or BKZ algorithm to find $\mathbf{v}$ and then construct a satisfying string.

Arbitrary Unpadding

Assuming we have a legal token, we can add two message blocks behind it to construct a new token, namely $token' := token||c_1||c_2$, where c_2 is completely randomly generated, and c_1 Except the last byte, the rest are \x00. Every time we change the last byte of c_1 and

traverse 255 possibilities, there must be a value that makes the new token legal, which is equivalent to knowing the last byte of $Dec(c_2)$. Then by setting the last byte of c_1, we can make the last byte of $Dec(c_2) \oplus c_1$ a specified value, and realize unpadding of any length. Here we record a set of c_1, c_2 as p.

Final Construction

Then the final payload is shaped like $iv'||ct_{name}||p||ct_{sig}||p||ct_{dig}||p||p$, that is, there is a The padding block makes the result of unpadding the expected value.