Write-Up - LWSCLWE - Ectario

PwnMe: Crypto - Medium **Author: Ectario**

Chall Information

Script

From the `server.py`:

```
#!/usr/bin/env python3
from flag import FLAG
import json
import secrets

n = 512
q = 0x10001
g = secrets.randbits(64)
MAX_REQUESTS = 1024


def product(A, B):
    """
    Computes the product of:
    - Matrix x Matrix
    - Matrix x Vector
    - Vector x Vector (dot product)

    :param A: List of lists (matrix) or a simple list (vector)
    :param B: List of lists (matrix) or a simple list (vector)
    :return: The product result as a list or a scalar
    """

    # Case 1: Vector × Vector → Dot product
    if isinstance(A[0], (int, float)) and isinstance(B[0], (int, float)):
        if len(A) != len(B):
            raise ValueError("Both vectors must have the same size")
        return sum(a * b for a, b in zip(A, B))
```

```
    # Case 2: Matrix × Vector
    if isinstance(A[0], list) and isinstance(B[0], (int, float)):
        if len(A[0]) != len(B):
            raise ValueError("The number of columns in A must match the size of B")
        return [sum(A[i][j] * B[j] for j in range(len(B))) for i in range(len(A))]

    # Case 3: Matrix × Matrix
    if isinstance(A[0], list) and isinstance(B[0], list):
        if len(A[0]) != len(B):
            raise ValueError(
                "The number of columns in A must match the number of rows in B"
            )

        m, n = len(A), len(A[0])
        p = len(B[0])

        result = [[0] * p for _ in range(m)]
        for i in range(m):
            for j in range(p):
                for k in range(n):
                    result[i][j] += A[i][k] * B[k][j]
        return result

    raise ValueError("Invalid inputs, A and B must be lists or lists of lists")


class LearningWeirdStreamCipherLikeWithErrors____WhatTheFuckThisClassName
    def __init__(self, challenge, flag):
        self.S = [secrets.randbelow(q) for _ in range(n)]
        self.challenge = challenge
        self.FLAG = flag
        stream = []
        while len(stream) < n:
            k = secrets.randbits(2 * n)
            stream = [g << i for i in range(len(bin(k)[2:]))]
            k_binary = bin(k)[2:]
            deleted = 0
            for i in range(1, len(k_binary)):
                k_i = k_binary[i]
                if not int(k_i):
```

```
                del stream[i - deleted]
                deleted += 1
        self.stream = stream
        self.leaks = 0

    def get_leak(self, index, e=secrets.randbelow(q)):
        if index < 0 or index >= len(self.stream):
            return {"error": "Invalid index"}
        if not self.leaks+1 < MAX_REQUESTS:
            return {"error": "Too much requests"}

        A = [secrets.randbelow(q) for _ in range(n)]
        B = product(A, self.S) + (self.stream[index] + e) * q
        self.leaks += 1
        return {"A": A, "B": str(B)}

    def get_encrypted_challenge(self):
        binary_challenge = list(
            map(
                int,
                " ".join(
                    bin(int.from_bytes(self.challenge, byteorder="big"))[2:]
                ).split(" "),
            )
        )
        encrypted_challenge = product(
            binary_challenge, self.stream[: len(binary_challenge)]
        )
        return {"value": hex(encrypted_challenge)[2:]}

    def get_flag(self, challenge_guess):
        if challenge_guess == self.challenge:
            return {"success": self.FLAG}
        return {"fail": "hmmmmm"}


def main():
    challenge = secrets.token_bytes(64)
    challenge_instance = LearningWeirdStreamCipherLikeWithErrors____WhatTheFu
        challenge, FLAG
    )
```

```python
print("Welcome to the LWSCLWE Challenge!")

while True:
    try:
        command = json.loads(input("Enter your command in JSON format: "))
        if "action" not in command:
            print(json.dumps({"error": "Invalid command format."}))
            continue

        action = command["action"]

        if action == "get_leak":
            if "index" not in command:
                print(json.dumps({"error": "Missing index"}))
            else:
                print(
                    json.dumps(challenge_instance.get_leak(int(command["index"])))
                )

        elif action == "get_encrypted_challenge":
            print(json.dumps(challenge_instance.get_encrypted_challenge()))

        elif action == "get_flag":
            if "challenge_guess" not in command:
                print(json.dumps({"error": "Invalid command format."}))
            else:
                challenge_guess = bytes.fromhex(command["challenge_guess"])
                print(json.dumps(challenge_instance.get_flag(challenge_guess)))

        elif action == "exit":
            print(json.dumps({"status": "Goodbye!"}))
            break

        else:
            print(json.dumps({"error": "Unknown action."}))
    except Exception as e:
        print(json.dumps({"error": str(e)}))
```

```
if __name__ == "__main__":
    main()
```

Known Infos

Information from the `server.py`

```
n = 512
q = 0x10001
g = secrets.randbits(64)
MAX_REQUESTS = 1024
```

How to Interact

The challenge operates through a simple JSON-based command interface. You send JSON-formatted requests to interact with the system, and it responds with JSON output.

Available Actions

1. `get_leak`

This action provides a leak of the internal system based on an indexed stream value. The response includes a matrix `A` and a corresponding computed value `B`.

Input Format:

```
{"action": "get_leak", "index": 10}
```

Response Format:

```
{"A": [random values], "B": "some computed value"}
```

2. `get_encrypted_challenge`

This action retrieves an encrypted form of a challenge value that you will need to decrypt in order to obtain the flag.

Input Format:

```
{"action": "get_encrypted_challenge"}
```

Response Format:

```
{"value": "encrypted hex value"}
```

3. `get_flag`

If you believe you have successfully decrypted the challenge, you can submit your guess to retrieve the flag.

Input Format:

```
{"action": "get_flag", "challenge_guess": "your hex-encoded guess"}
```

Response Format (if correct):

```
{"success": "flag{your_flag_here}"}
```

Response Format (if incorrect):

```
{"fail": "hmmmmm"}
```

4. `exit`

This action gracefully exits the challenge.

Input Format:

```
{"action": "exit"}
```

Response Format:

```
{"status": "Goodbye!"}
```

Objective

The **LWSCLWE** Challenge presents a cryptographic system that leaks information about an internal secret through matrix operations (that kinda looks like LWE).

The goal seems to be to analyze the leaks (`get_leak` results), derive the internal secret stream, and decrypt the challenge (`get_encrypted_challenge`). If you can correctly reconstruct the challenge, submit it using `get_flag` to retrieve the final flag.

Analyse

Recovering the secret value S

In `get_leak`:

```
def get_leak(self, index, e=secrets.randbelow(q)):
    if index < 0 or index >= len(self.stream):
        return {"error": "Invalid index"}
    if not self.leaks+1 < MAX_REQUESTS:
        return {"error": "Too much requests"}

    A = [secrets.randbelow(q) for _ in range(n)]
    B = product(A, self.S) + (self.stream[index] + e) * q
    self.leaks += 1
    return {"A": A, "B": str(B)}
```

This function resembles the Learning With Errors (LWE) problem, where an unknown secret `S` is hidden within noisy linear equations. Here, the function generates a random vector `A`, computes `B` using a dot product with the secret `S`, and adds noise scaled by `q` to obscure the stream value.

So:

$$B = \langle A, S \rangle + (\text{stream}[i] + e) \cdot q$$

This scheme allows us to forge 512 independent equations by requesting leaks with different values of `A`. If we switch to an appropriate modulus (working in $\mathbb{F}_q$), we can eliminate the unknown term $(\text{stream}[i] + e) \cdot q$, reducing the equation to:

$$B \mod q = \langle A, S \rangle \mod q$$

> This will results in a system of 512 linear equations with 512 unknowns (the components of `S`), which can be solved efficiently using standard linear algebra techniques such as Gaussian elimination.

$$\begin{cases} \langle A_1, S \rangle \equiv B_1 \mod q \\ \langle A_2, S \rangle \equiv B_2 \mod q \\ \vdots \\ \langle A_{512}, S \rangle \equiv B_{512} \mod q \end{cases}$$

And so:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,512} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,512} \\ \vdots & \vdots & \ddots & \vdots \\ A_{512,1} & A_{512,2} & \cdots & A_{512,512} \end{bmatrix} \cdot \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{512} \end{bmatrix} \equiv \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{512} \end{bmatrix} \mod q$$

$$\Rightarrow S \equiv A^{-1}B \mod q$$

Now, the next step is to successfully recover the `stream`. To do this, we need to understand how it was created in order to analyze its structure.

It is also important to note that the noise term `e` is only generated **once**, during the first execution of the function, and remains fixed for subsequent calls. Also, `e` is brute-forceable since it is `< q` and `q = 0x10001`.

Stream Initialization

In `__init__`:

```
def __init__(self, challenge, flag):
    self.S = [secrets.randbelow(q) for _ in range(n)]
    self.challenge = challenge
    self.FLAG = flag
    stream = []
    while len(stream) < n:
        k = secrets.randbits(2 * n)
        stream = [g << i for i in range(len(bin(k)[2:]))]
        k_binary = bin(k)[2:]
        deleted = 0
        for i in range(1, len(k_binary)):
            k_i = k_binary[i]
            if not int(k_i):
                del stream[i - deleted]
                deleted += 1
    self.stream = stream
    self.leaks = 0
```

The code generates a sequence called `stream`, which serves as a **random basis** while ensuring it is **super-increasing *(that is really important to note)***. The process involves:

1. Initializing an empty list `stream`.
2. Generating a random integer `k` with `2 * n` bits.

3. Constructing an initial sequence by left-shifting a base value `g` according to the positions of the bits in `k`.
4. Filtering out elements corresponding to zero bits in the binary representation of `k`.
5. Assigning the final sequence to `self.stream`.

Since the sequence construction is directly based on bit positions, it retains an inherent **order**. The filtering step ensures that the resulting sequence remains sparse while preserving the increasing nature of its elements.

Lil' answer for this one: Why the sequence is super-increasing?

A sequence $\{s_1, s_2, ..., s_n\}$ is **super-increasing** if:

$$s_i > \sum_{j=1}^{i-1} s_j, \quad \forall i \geq 2$$

Step 1: Constructing the Sequence

- Let k be a randomly generated integer with up to 2^n bits.
- The binary representation of k is parsed, and an initial sequence is formed using **left shifts** of a base value g:

$$s_i = g \cdot 2^i, \quad \text{where } i \text{ corresponds to the bit positions of } k$$

- Any bit set to zero results in the removal of the corresponding term from `stream`.

Step 2: The nature of the Super-Increasing Property

- Consider any two consecutive elements in the sequence, say:

$$s_1, s_2, \ldots, s_m$$

where $s_i = g \cdot 2^{b_i}$ for strictly increasing indices $b_1 < b_2 < \cdots < b_m$.

- By definition of left shifts we have:

$$s_i = g \cdot 2^{b_i}$$

Also:

$$2^{b_{i+1}} = 2 \cdot 2^{b_i}$$

$$g \cdot 2^{b_{i+1}} = g \cdot 2 \cdot 2^{b_i}$$

And so, since $b_{i+1} > b_i$, implying $2^{b_{i+1}}$ is at least **twice** 2^{b_i} and same for $g \cdot 2^{b_{i+1}} > g \cdot 2^{b_i}$, the following is trivially true:

$$\Rightarrow 2^{b_0} < 2^{b_1} < 2^{b_0} + 2^{b_1} < 2^{b_2} < 2^{b_0} + 2^{b_1} + 2^{b_2} < 2^{b_3} < \cdots < \sum_{j=1}^{i} 2^{b_j} < 2^{b_{i+1}}$$

$$\Rightarrow 2^{b_{i+1}} > 2^{b_1} + 2^{b_2} + \cdots + 2^{b_i}$$

$$\Rightarrow g \cdot 2^{b_{i+1}} > g \cdot (2^{b_1} + 2^{b_2} + \cdots + 2^{b_i})$$

$$\Rightarrow s_{i+1} > \sum_{j=1}^{i} s_j$$

(note: could be proved rigorously by induction, this is a fun exercise left to the readers)

Step 3: Conclusion

Since each element in `stream` is strictly greater than the sum of all previous elements, we can effectively conclude that:

$$\forall i \geq 2, \quad s_i > \sum_{j=1}^{i-1} s_j$$

Thus, **`stream` is guaranteed to be a super-increasing sequence.**

This process creates a **randomized basis** because `k` is randomly chosen, affecting which indices contribute to the sequence. However, the structural nature of left shifts and binary filtering **guarantees** that the resulting sequence remains super-increasing.

Recovering the secret value `stream` & `challenge` using `S`

Since we know that the stream is **super-increasing** and that `e` is brute-forceable, we can simply apply the following algorithm which simply tries to find the correct `stream` by recomputing a `challenge` for each attempt and sending it to our oracle **`get_flag(challenge_guess)`**, which tells us whether it is correct or not:

```
stream_plus_e = []

# Here, we're just recovering the unknown: stream[index] + e
for i in tqdm(range(512)):
    A, B = collected_A[i], collected_B[i]
```

```
    stream_plus_e.append(int(B - int(np.array(A) @ S)) // q)

# We brute-force 'e' while trying to reverse the super-increasing sequence appli
ed to the challenge
_sum_init = int(get_encrypted_challenge()["value"], 16)
for e in tqdm(range(q)):  # Brute-force search on 'e'
    binary_challenge = []
    _sum = _sum_init
    for i in range(511, -1, -1):  # Process the stream in reverse order
        stream_i = stream_plus_e[i] - e
        b = 0
        if _sum >= stream_i:
            b = 1
            _sum -= stream_i
        binary_challenge.append(str(b))

    # oracle stage to whether validate or not our challenge_guess
    challenge = int("".join(binary_challenge[::-1]), 2)
    challenge = hex(challenge)[2:]
    if len(challenge) % 2 == 1:
        challenge = "0" + challenge  # Pad with a leading zero if necessary
    flag_response = get_flag(challenge)
    if "success" in flag_response:
        print(flag_response)
        FOUND = True
        break
```

This algorithm exploits the **super-increasing** property of the stream, meaning that each element is significantly larger than the sum of all previous elements. This makes it possible to reconstruct the binary sequence using **greedy subtraction**. Since `e` is small, brute-forcing it is computationally feasible. The main logic can be simply described as follows:

Starting from the last element and moving backward, there are two cases:

1. **If** $\text{sum} < \text{arr}[i]$, the element is **excluded** since adding it would exceed the target sum.
2. **If** $\text{sum} \geq \text{arr}[i]$, the element **must be included**, as the remaining elements alone cannot reach the sum in a super-increasing sequence. The sum is then updated as:

$$\text{sum} = \text{sum} - \text{arr}[i]$$

From this point, once the challenge is correct, we immediately Capture The Flag.

Final Exploit

Sometimes, the script doesn't work on the first try and requires a few attempts. I haven't really looked into why, but it might be due to approximations failing because of the large numbers being manipulated.

```
from sage.all import *
from pwn import process, remote
import numpy as np
from json import loads, dumps
from tqdm import tqdm

HOST = "localhost"
PORT = 32770


def get_leak(index):
    proc.sendline(dumps({"action": "get_leak", "index": str(index)}))
    res = proc.recvline()
    clean()
    return loads(res)


def get_flag(challenge_guess):
    proc.sendline(dumps({"action": "get_flag", "challenge_guess": challenge_guess}
    res = proc.recvline()
    clean()
    return loads(res)


def get_encrypted_challenge():
    proc.sendline(dumps({"action": "get_encrypted_challenge"}))
    res = proc.recvline()
    clean()
    return loads(res)


def clean():
    return proc.recvuntil(b"format: ")
```

```
n = 512
q = 0x10001
MAX_REQUESTS = 512

Fq = FiniteField(q)
FOUND = False

while not FOUND:
    # proc = process("./server.py")
    proc = remote(HOST, PORT)
    print(clean())

    collected_A = []
    collected_B = []

    for i in tqdm(range(512)):
        R = get_leak(i)
        collected_A.append(list(map(int, R["A"])))
        collected_B.append(
            int(R["B"])
        )

    Amat = Matrix(Fq, collected_A) # being in Fq, so it cancels (self.stream[index] + e
    Bvec = vector(Fq, collected_B) # being in Fq, so it cancels (self.stream[index] + e
    S = list(map(int, Amat.solve_right(Bvec)))

    stream_plus_e = []

    for i in tqdm(range(512)):
        A, B = collected_A[i], collected_B[i]
        stream_plus_e.append(int(B - int(np.array(A) @ S)) // q)

    # we bruteforce 'e' while trying to reverse the super increasing sequence applied
    _sum_init = int(get_encrypted_challenge()["value"], 16)
    for e in tqdm(range(q)):
        binary_challenge = []
        _sum = _sum_init
        for i in range(511, -1, -1):
            stream_i = stream_plus_e[i] - e
            b = 0
```

```
        if _sum >= stream_i:
            b = 1
            _sum -= stream_i
        binary_challenge.append(str(b))
    challenge = int("".join(binary_challenge[::-1]), 2)
    challenge = hex(challenge)[2:]
    if len(challenge) % 2 == 1:
        challenge = "0" + challenge  # Pad with a leading zero if necessary
    flag_response = get_flag(challenge)
    if "success" in flag_response:
        print(flag_response)
        FOUND = True
        break
proc.close()
```