CTF Blackalps - Blum Blum Shub Crypto Challenge

Nathan Rayburn

November 8, 2024

Abstract

This challenge was presented during the Blackalps CTF event. It involves reversing a Blum Blum Shub (BBS) implementation, which is used for pseudorandom number generation. All the related files are available on GitHub.

1 Introduction

Here is how the challenge was presented:

```python
# Parameters are in the github.

# PRNG that generates number_bytes pseudo-random bytes
# seed has to be a random element in Z_n
# n = pq with p, q prime
def bbs(seed, number_bytes, n):
    ret = 0
    for _ in range(number_bytes*8):
        seed = pow(seed, 2, n)
        ret <<= 1
        ret |= (seed % 2)
    return ret.to_bytes(number_bytes), seed

# Encrypts the flag with a stream cipher based on the BBS PRNG.
# Returns the ciphertext and a masked final state of the PRNG
# new_seed + p is given as rp in the parameters
# new_seed + q is given as rq in the parameters
def hide_flag(seed, n, flag, p, q):
    (random, new_seed) = bbs(seed, len(flag), n)
    return strxor(flag, random), new_seed + p, new_seed + q
```

Listing 1: Challenge

2 Breakdown

Our goal is to obtain the original seed so that we can calculate the random number by reversing the BBS algorithm.

2.1 Finding p and q

What we know:

We have n, which is equal to $p \cdot q$, and we are given rp and rq in the parameters, which correspond to $rp = \text{new\_state} + p$ and $rq = \text{new\_state} + q$. We do not know p or q, but we have the rest (n, rp, rq).

We can isolate the difference between p and q:

$$\text{new\_state} + p - (\text{new\_state} + q) = p - q = d$$

Thus, we know that:

$$p = d + q$$

Therefore:

$$n = (d + q) \cdot q$$

$$n = d \cdot q + q^2$$

$$0 = q^2 + d \cdot q - n$$

We can solve for q using the quadratic formula:

$$q = \frac{-d \pm \sqrt{d^2 + 4n}}{2}$$

where d and n are constants.

Then, we can test which solution satisfies $p \cdot q = n$.

```
q1 = (-d + sqrt_discriminant) // 2
q2 = (-d - sqrt_discriminant) // 2
p1 = q1 + d
p2 = q2 + d

if p1 * q1 == n:
    return p1, q1
elif p2 * q2 == n:
    return p2, q2
else:
    raise ValueError("Failed to find factors p and q")
```

Listing 2: Code Snippet

2.2 Finding the Original Seed

In the given parameters, it is stated that rp is for the first state mask and rq is the second seed mask.

We can obtain the final state outputted by the `bbs` function. Since we know p and q, we can compute the final state by subtracting:

```
final_state = rp - p if rp - p == rq - q else None
```

Listing 3: Code Snippet

We observe that when the seed is processed in the `bbs` function, it undergoes *length* $\times$ 8 squaring operations. To reverse this process, we need to compute square roots iteratively, operating within the ring modulo n, which we can achieve using the Chinese Remainder Theorem (CRT).

To find the length of the input, we can decode the base64-encoded ciphertext and compute its length. This yields 31 bytes.

Now, we can implement our 248 iterations (since $31 \times 8 = 248$) of square roots. This should return the original seed that was input.

```
# Function to reverse the BBS and retrieve the original seed
def reverse_bbs(final_seed, p, q):
    current_state = final_seed
    for _ in range(31*8):
        root_seed_p = pow(current_state, (p+1)//4, p)
        root_seed_q = pow(current_state, (q+1)//4, q)
        current_state = crt([root_seed_p, root_seed_q], [p, q])
    return current_state
```

Listing 4: Code Snippet

We can approach this in multiple ways: either reverse the random bytes or simply pass the reversed seed back into the `bbs` function. I chose simplicity.

```
random, _ = bbs(reversed_seed, len(ct), n)
flag = strxor(random, ct)
```

Listing 5: Code Snippet

And voilà! We have our flag.

```
Recovered flag as string: BA24{B1umB1umShubh4s4tr4pd00r!}
```

Listing 6: Code Snippet