CTF Blackalps - Sponge Crypto Challenge

Nathan Rayburn

November 13, 2024

Abstract

This document outlines the solution to the Blackalps CTF Sponge Crypto Challenge. The task involves reversing a cryptographic sponge function to recover the flag, utilizing known parameters and brute-forcing the unknown rate and internal state.

1 Introduction

The challenge provided the following parameters:

- A hash value.
- The flag's format and length: `BA{...}`, consisting of 19 ASCII characters.

The flag's format `BA{...}` differs from the usual `BA24{...}` format observed in similar challenges, which raised some suspicion. Below is the Python implementation of the sponge function used in the challenge:

```python
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.strxor import strxor

cipher = AES.new(b"\x07"*16, AES.MODE_ECB)

def permutation(b):
    return cipher.encrypt(b)

def pad(message, rate):
    missing = rate - len(message) % rate
    if missing == 0:
        missing = rate
    message += b"\x80" + b"\x00" * (missing - 1)
    return message

def sponge(rate, message, output_size):
    # Padding
    message = pad(message, rate)
    blocks = [message[rate*i : rate*(i+1)] for i in range(len(message)
   // rate)]
    state = b"\x00" * 16

    # Absorbing phase
    for b in blocks:
        state = strxor(state[:rate], b) + state[rate:]
        state = permutation(state)

    # Squeezing phase
    hash = b""
    while len(hash) < output_size:
        hash += state[:rate]
        state = permutation(state)
    return hash[:output_size]
```

Listing 1: Sponge Function Definition

2 Challenge Breakdown

2.1 Stage 1 - External State Recovery

The goal was to brute-force the `rate`, leveraging the small state size:

- The hash length is 64 bytes.
- The sum of `rate` and `capacity` equals the state size (16 bytes).

To reverse the sponge function, we tested possible `rate` values by brute-forcing unknown bytes in the `capacity` to reconstruct the external state. Starting with a small capacity (e.g., 1 byte) could yield faster results.

Given the initial hash:

$$z_0 = \text{hash}[: r]$$

where r is the unknown rate, the next hash output block is:

$$z_1 = \text{hash}[r : r \cdot 2]$$

The concept is as follows:

$$z_1 = \text{permutation}(z_0 || \text{guess\_bytes})$$

2.1.1 Implementation

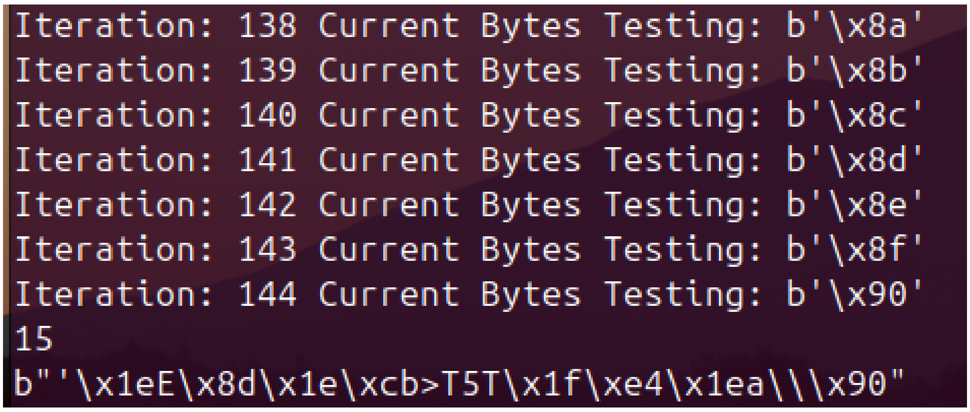
```python
def bruteforce_ext_state(hash):
    state_size = 16
    for rate_size in range(state_size - 1, 1, -1):
        capacity_size = state_size - rate_size
        current_rate = hash[:rate_size]
        current_capacity = b'\x00' * capacity_size

        for i in range(2 ** (capacity_size * 8)):
            concat = current_rate + current_capacity
            before_perm = zeroPad(concat, state_size)

            if hash[rate_size:rate_size*2] == permutation(before_perm)[:rate_size]:
                return rate_size, current_rate + current_capacity

            current_capacity = increment_bytes(current_capacity)
    return None
```

Listing 2: Bruteforce External State

2.1.2 Results

The brute-forcing process revealed:

- The `rate` value was determined to be 15.
- The external state was successfully reconstructed.



```
Iteration: 138 Current Bytes Testing: b'\x8a'
Iteration: 139 Current Bytes Testing: b'\x8b'
Iteration: 140 Current Bytes Testing: b'\x8c'
Iteration: 141 Current Bytes Testing: b'\x8d'
Iteration: 142 Current Bytes Testing: b'\x8e'
Iteration: 143 Current Bytes Testing: b'\x8f'
Iteration: 144 Current Bytes Testing: b'\x90'
15
b"'\x1eE\x8d\x1e\xcb>T5T\x1f\xe4\x1ea\\\x90"
```

Figure 1: Output

2.2 Stage 2 - Internal State Recovery

Due to the architecture of the sponge construction, we know that the message block is xored with the rate, therefore now we know our message is split into 2 blocks of 15 bytes.

Here is a quick overview for the sponge construction.

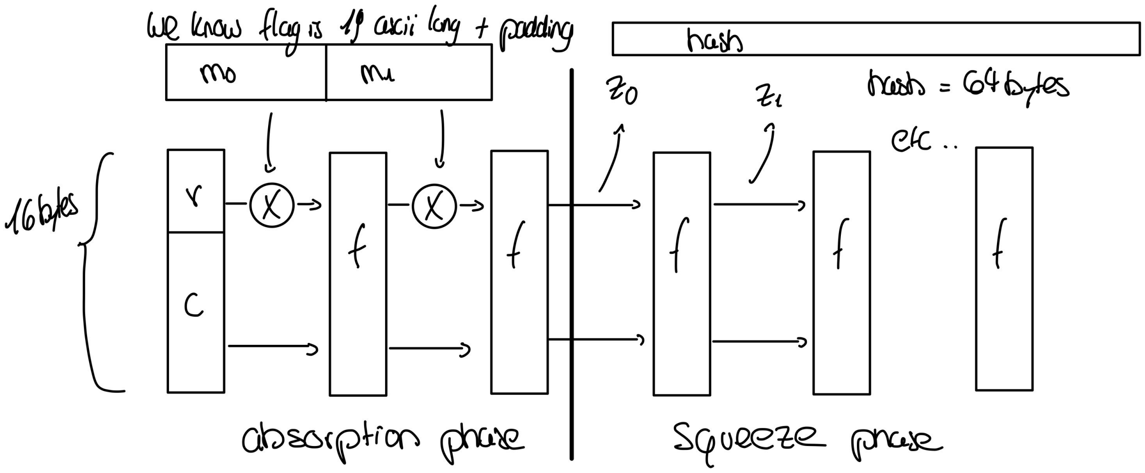


Figure 2: Sponge construction

Using the sponge construction's properties, the flag's second block was identified. With a 19-character ASCII flag, the second block comprises 4 characters plus padding. The padding format is:

"xxx}\x80\x00..."

We aimed to reverse the external state to determine the internal state:

$$\text{internal\_state} = \text{inverse\_permutation}(\text{external\_state})$$

2.2.1 Implementation

The Python implementation for reversing the internal state and reconstructing the flag is shown below:

```python
def bruteforce_inner_state(state,r):
    know_string_start_flag = b'BA24{'  # Known start of the flag
    known_string_end_flag = b"}\x80" + b"\x00" * 10 # end flag + padding
    unknown_bytes = 3

    # All possible 3-byte combinations of ASCII characters
    ascii_range = range(32, 127)
    for guess in itertools.product(ascii_range, repeat=unknown_bytes):

        bytes_to_brute_force = bytes(guess)

        test_string = bytes_to_brute_force + known_string_end_flag

        guess_state = inverse_function_perm(strxor(state[:r], test_string[:r]) + state[r:])

        # Check if we have found the flag
        if guess_state[:r].startswith(know_string_start_flag):
            print("Match found with current bytes:", bytes_to_brute_force)
            return guess_state[:r] + bytes_to_brute_force + known_string_end_flag
```

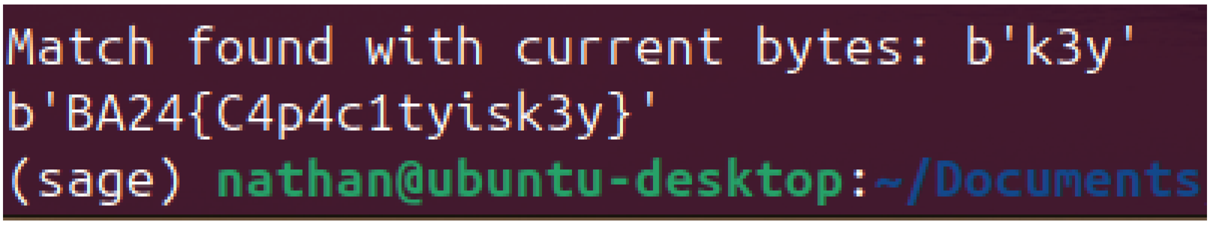
Listing 3: Bruteforce Internal State

2.2.2 Result

All we need to do is unpad and voilà.

```
b'BA24{C4p4c1tyisk3y}'
```

Listing 4: Flag



```
Match found with current bytes: b'k3y'
b'BA24{C4p4c1tyisk3y}'
(sage) nathan@ubuntu-desktop:~/Documents
```

Figure 3: Output