justCTF teaser 2024: Casino

> by FluxFingers

The challenge was a provably fair gambling site and the goal was to make 1 billion dollars to buy the flag. A user starts off with 1000 dollars and can bet on dice rolls with a minimum bet of 1$ and a maximum of their current balance. If the user predicts the dice roll correctly, they get triple their wager back. If they mispredict, they lose their wager.

The provable fairness is implemented with a server seed, a client seed, and a nonce. The server seed is a securely generated, 64-character, hexadecimal string and is not known by the user while being used. The user can regenerate the server seed which will reveal the old seed. The client seed is submitted by the client with every bet and must be a string consisting of 64 Unicode code units (or, in more simple terms, must have a `.length` of 64 in JavaScript). The nonce is a number visible to the user that starts at 0 and is incremented for after each bet. Regenerating the server seed also resets the nonce to 0. Both server seed and nonce are stored per user.

A dice roll is not entirely random. To provide provable fairness, the dice roll outcome is derived from the three values discussed above (server seed, client seed, nonce). This is done by seeding a random number generator (RNG) with those three values. The challenge uses the seedrandom library for this by building a JSON string from the three values, using it to index the seedrandom RNG, generating a 32-bit integer, and mapping the result to the desired target range of $[1, 6]$:

```
let roll = (seedrandom(JSON.stringify({
    serverSeed: req.user.serverSeed,
    clientSeed,
    nonce: req.user.nonce++
})).int32() >>> 0) % 6 + 1
```

According to its documentation, seedrandom uses RC4 to compute its outputs, and the seed string seems to be used as RC4's key. Before that, they reduce the arbitrarily long string to a length of 256 with the `mixkey` function:

```
function mixkey(seed, key) {
  var stringseed = seed + '', smear, j = 0;
  while (j < stringseed.length) {
    key[mask & j] =
      mask & ((smear ^= key[mask & j] * 19) + stringseed.charCodeAt(j++));
  }
  return tostring(key);
}
```

To beat the casino and reliably win, we will try to cause collisions in this function. This will result in the RNG generating the same number repeatedly, allowing us to predict the dice roll and win big!

The `mixkey` function will fill the 256-element `key` array based on the characters in the `seed` string. For the first 256 characters, the function will simply copy the character to the array, but limit each character value to the range $[0, 255]$ by applying `mask` which has the value 0xFF.

For all characters after the 256th, the array index (`j`) wraps around and sets the array element based on its previous value, the current `seed` character, and a continuously changing `smear` value:

$$smear = smear \oplus (key[j \ mod \ 256] * 19)$$

$$key[jmod256] = smear + seed[j] \ mod \ 256$$

We can already make a few observations here:

- Seeds shorter than 256 characters will not cause a trivial collision
- Only the least significant byte of `smear` is actually used for the computation of `key`

- The nonce always changes between two dice rolls but we can also change the client seed if we want

Let's take a look at a normal JSON seed value that the website will produce with a 64-character hexadecimal client seed (whitespaces added for readability):

```
{
  "serverSeed": "84ec7b197400d10d83e8d7a41422116ea7715e1979b1d65ecd2ac8f6264d4e38",
  "clientSeed": "e3bc1fc37ce0b81f869412befba853e20dbebb9ad8c16568f31492a04118bc22",
  "nonce": 0
}
```

This JSON string has a length of 171 characters. Since the client seed is limited to 64 characters, does the `mixkey` function ever wrap over? Yes! The wrong assumption here is that a 64-character string will also take up 64 characters in the JSON string. This is true for many characters, but some characters will be converted to sequences of more than 1 by `JSON.stringify()`.

One example of this is the null byte (0x00). It is a perfectly valid character for a JavaScript string, but it will be represented as `\u0000` inside a JSON string. Similarly, characters such as double quotes (`"`), backslashes (`\`), or newlines (0x0A) are represented as `\"`, `\\`, and `\n` respectively. This allows us to use a 64-character client seed that will take up almost any amount between 64 and $6 * 64 = 384$ characters in the JSON string. This means that we can cause a wrap-around in `mixkey` and start trying to build a collision!

As our next step, we will elongate the JSON string so that the characters of the nonce will be mixed with characters from the client seed after the wrap-around:

```
{"serverSeed":"84ec7b197400d10d83e8d7a41422116ea7715e1979b1d65ecd2ac8f6264d4e38","clientS
eed":"AAAAAAAAAAAAAAAAAAAAAAAA\n\n\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u000
0\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u000

0\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000","n
once":0}

^
```

For this seed, the key array item at the marked position will be based on:

- `A` (value 65, from the client seed)
- `0` (value 48, from the nonce)
- `smear` (depends on the previous characters, including the server seed)

If `smear` was `0` here, the output value would be:
$(smear \oplus ('A' * 19)) + '0' \ mod \ 256 = (0 \oplus (65 * 19)) + 48 \ mod \ 256 = 3$

For the next bet, the nonce will change from `0` to `1`. Assuming `smear` to be `0` again, the output value would be: $(smear \oplus ('A' * 19)) + '1' \ mod \ 256 = (0 \oplus (65 * 19)) + 49 \ mod \ 256 = 4$

To cause a collision, we just have to find a character instead of `A` that will result in 3 like the calculation for the previous bet. We could do math here, or be lazy and write a for loop 🤷 In any case, we will find that 38 fits here, which is the value of the ampersand character (`&`). We will have to do the same calulation for the next position (`A` and `}`) because the `smear` value will now be different for that position.

And with that, we are done and can create arbitrary collisions, right? Right??

Unfortunately, we left out a crucial part: we don't know the value of `smear`. Since that value is based on the first characters of the seed, which includes the entire server seed, we can't know the value. We were stuck here for some time, trying to find a way of predicting a `smear` value by finding some pattern.

Luckily we remembered one of our earlier observations: only the least significant byte of the `smear` is ever used for the output of the `mixkey` function! While this does not help us to predict the value of `smear`, it does tell us that we can just guess it. We can test our guess by assuming the value of `smear` and trying out if we can predict the next few dice rolls with that. Since we have a 1000$ balance, we can easily make 4 rolls per guess and try out all 256 possible values.

Easier said than done, we wasted an embarrassing amount of time debugging and fixing our solve script here. One hurdle we had to overcome was that sometimes the character we had to insert to make the calculations right would be one of those that are represented with more than one character in JSON. To overcome this, we simply added 256 to the character value because the final output was truncated to its least significant byte using `mask` (0xFF). This was possible because the seed characters were not limited to the ASCII range. After fixing more implementation bugs in our script, the calculations seemed to work and we were able to find a valid value pretty reliably.

All that was left now was to point our script at the challenge's remote instance and add some code that would continue betting the maximum amount until we're rich and then buy the flag. And with that, we got the flag!

> You are soo lucky, here is your flag: justCTF{n0_w4y__h0w_1ucky_4re_y0u??}