MANDIANT
NOW PART OF Google Cloud

FLARE-ON CHALLENGE 9 SOLUTION

BY SAM KIM

Challenge 7: anode

Challenge Prompt

You've made it so far! I can't believe it! And so many people are ahead of you!

7-zip password: flare

Solution

This challenge consists of a node.js script compiled to a binary using nexe. (We can figure this out by the <nexe~~sentinel> near the end of the binary and the PDB path.) If we take a look at the nexe compiler (https://github.com/nexe/nexe/blob/805ca3514905adb4b8cdb24bffbcca2b46ce176a/src/compiler.ts#L301-L308), we see that it concatentates a prebuilt nexe binary, some support code, user code, and the lengths of the scripts.

```
    lengths.writeDoubleLE(codeSize, 0)
    lengths.writeDoubleLE(this.bundle.size, 8)
    return new (MultiStream as any)([
      binary,
      toStream(code),
      this.bundle.toStream(),
      toStream(Buffer.concat([Buffer.from('<nexe~~sentinel>'), lengths])),
    ])
```

All we need to do now is extract the script using the second length.

```
data = open("anode.exe","rb").read()
length = int(struct.unpack('d', data[-8:])[0])
offset = len(data) - 16 - len('<nexe~~sentinel>') - length
script = data[offset:offset+length]
open("script.js","wb").write(script)
```

We get the following script:

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout,
});

readline.question(`Enter flag: `, flag => {
  readline.close();
  if (flag.length !== 44) {
    console.log("Try again.");
    process.exit(0);
  }
  var b = [];
  for (var i = 0; i < flag.length; i++) {
    b.push(flag.charCodeAt(i));
  }

  // something strange is happening...
  if (1n) {
    console.log("uh-oh, math is too correct...");
    process.exit(0);
  }

  var state = 1337;
  while (true) {
    state ^= Math.floor(Math.random() * (2**30));
    switch (state) {
      case 306211:
        if (Math.random() < 0.5) {
          b[30] -= b[34] + b[23] + b[5] + b[37] + b[33] + b[12] + Math.floor(Math.random() *
256);
          b[30] &= 0xFF;
        } else {
          b[26] -= b[24] + b[41] + b[13] + b[43] + b[6] + b[30] + 225;
          b[26] &= 0xFF;
        }
        state = 868071080;
        continue;
...
      default:
        console.log("uh-oh, math.random() is too random...");
        process.exit(0);
    }
    break;
  }

  var target = ...;
  if (b.every((x,i) => x === target[i])) {
    console.log('Congrats!');
  } else {
    console.log('Try again.');
  }
});
```

This script takes in the flag, applies a few reversible transformations to the flag, then checks it against a target array to see if it was correct.

As stated in the comment, we immediately see that something strange is happening. There's a block of code wrapped in an if (1n) {...} which should execute and make the script exit, and indeed that's what happens when we try to run the script by itself, but that doesn't happen when we run the binary! There's also a bunch of places where Math.random() is used so we should almost always hit the default case in the switch statement and exit the script, but that doesn't also happen when running the binary! What's going on?

Since we're getting different behavior when running the binary and when running the script by itself, maybe the binary was tampered with? We can try finding a clean copy of the binary to see what changed. Luckily, the PDB path tells us that this is version 14.15.3 of nexe.

C:\Users\VssAdministrator\.nexe\14.15.3\out\Release\node.pdb

Since this is a 64-bit binary, we grab the windows-x64-14.15.3 version from the nexe releases page (https://github.com/nexe/nexe/releases/download/v3.3.3/windows-x64-14.15.3).

PRNG Tampering

RandomNumberGenerator::SetSeed and MathRandom::RefillCache have been patched to use the constant state (0x60c43c4809ad2d74, 0xce6a1a53db4c5403) in the xorshift128+ PRNG that Math.random() uses.

We can reimplement Math.random() using this fixed seed. Note that v8 generates 64 random values at a time and outputs them in reverse order.

```
def xorshift128(state0, state1):
    MASK = 0xFFFFFFFFFFFFFFFF
    while True:
        s1, s0 = state0, state1
        state0 = s0
        s1 ^= (s1 << 23) & MASK
        s1 ^= s1 >> 17
        s1 ^= s0
        s1 ^= s0 >> 26
        state1 = s1
        yield struct.unpack('<d', struct.pack('<Q', (state0 >> 12) | 0x3FF0000000000000))[0] - 1

def math_random():
    xs128p = xorshift128(0x60c43c4809ad2d74, 0xce6a1a53db4c5403)
    while True:
        bucket = []
        for i in range(64):
            bucket.append(next(xs128p))
        for i in range(64):
            yield bucket.pop()
```

Control Flow Tampering

Literal::ToBooleanIsTrue was patched with the following changes:

- For Smis (small ints: ints that fit in the signed 31-bit range), the result is inverted.
- For Bigints, 2n and all bigints with more than 1 digit that don't contain a 0 are treated as false.

This, combined with the Math.random() reimplementation above, lets us figure out which blocks are executed in each if statement. Once we know which transformations are applied to the flag, we can apply the reverse transformations to the target array to get the final flag:

```
n0t_ju5t_A_j4vaSCriP7_ch4l1eng3@flare-on.com
```

Alternate approaches

Instead of reversing the nexe binary to see what changed, we can try modifying the script to add our own code while using the same modified binary. After modifying the script, we also need to update the script length at the end of the binary (this.bundle.size in the first code snippet).

PRNG Recovery

We can add the following block of code before the script to get all the outputs of Math.random() that are used in the script.

```
for (var i = 0; i < 1024; i++) {
   console.log(Math.random());
}
```

Control Flow Recovery

We can add a console.log to every basic block, then see which log statements were executed from the output.

SOLVE SCRIPT

```
import struct
import math

def xorshift128(state0, state1):
    MASK = 0xFFFFFFFFFFFFFFFF
    while True:
        s1, s0 = state0, state1
        state0 = s0
        s1 ^= (s1 << 23) & MASK
        s1 ^= s1 >> 17
        s1 ^= s0
        s1 ^= s0 >> 26
        state1 = s1
        yield struct.unpack('<d', struct.pack('<Q', (state0 >> 12) | 0x3FF0000000000000))[0] - 1

def math_random():
    xs128p = xorshift128(0x60c43c4809ad2d74, 0xce6a1a53db4c5403)
    while True:
        bucket = []
        for i in range(64):
            bucket.append(next(xs128p))
        for i in range(64):
            yield bucket.pop()

r = math_random()

data = open("anode.exe","rb").read()
length = int(struct.unpack('d', data[-8:])[0])
offset = len(data) - 16 - len('<nexe~~sentinel>') - length
script = data[offset:offset+length].decode('utf8')

cases = {}

for case in script.split('case')[1:]:
    case = case.split('continue')[0]
    key = int(case.split(':')[0].strip())
    if 'break' in case:
        cases[key] = 'break'
    else:
        cond = case.split('if (')[1].split(') {')[0]
        if_true = case.split(') {')[1].split('} else {')[0].strip()
        if_false = case.split('} else {')[1].split('}')[0].strip()
        next_state = int(case.split('state = ')[1].split(';')[0])
        cases[key] = (cond, if_true, if_false, next_state)

transforms = []

state = 1337
while True:
    state ^= math.floor(next(r) * 2**30)
    if cases[state] == 'break':
        break
    cond, if_true, if_false, next_state = cases[state]
    if 'random' in cond:
        cond_value = next(r) < 0.5
    elif cond[-1] == 'n':
        # bigint
        cond_value = '0' in cond
    else:
        # smi
        cond_value = int(cond) > 2147483647
    stmt = if_true if cond_value else if_false
    operation = stmt.split('=')[0][-1]
    idxs = [int(i.split(']')[0]) for i in stmt.split(';')[0].split('[')[1:]]
    constant = stmt.split(';')[0].split(' + ')[-1]
    if 'random' in constant:
        constant = math.floor(next(r) * 256)
    elif operation == '^':
        constant = int(constant.split(')')[0])
    else:
        constant = int(constant)
    print(operation, idxs, constant)
    transforms.append((operation, idxs, constant))
    state = next_state

target = eval(script.split('target = ')[1].split(';')[0])
for operation, idxs, constant in transforms[::-1]:
    value = (sum(target[idx] for idx in idxs[1:]) + constant) & 0xFF
    if operation == '^':
        target[idxs[0]] ^= value
    elif operation == '+':
        target[idxs[0]] -= value
        target[idxs[0]] &= 0xFF
    elif operation == '-':
        target[idxs[0]] += value
        target[idxs[0]] &= 0xFF

print(bytes(target).decode('utf8'))
```