FLARE-On 13 Challenge 8: crux

By Sam Kim (@sdkim)

Overview

Based on the presence of `manifest.json` and `wasm_exec.js`, `crux` is a Chrome extension containing a WebAssembly binary (`main.wasm`) compiled from Go. The frontend script `popup.js` invokes `window.check`, which is registered by the WASM module via `js.Global().Set("check", js.FuncOf(check_func))`. Reversing `check_func` reveals that it validates a serialized Pratt primality certificate for a hardcoded 256-bit prime and uses the certificate's SHA-256 digest to decrypt the hidden flag: **`Pr4tt_Pr1M4lItY_C3rt5_4nd_Wa4aA4aAA4aAA4a4a5M@flare-on.com`**.

Initial Triage & Go WASM Obfuscation

Inspecting the strings inside `main.wasm` reveals that the Go binary was obfuscated with `garble`, as the package and function identifiers are short random alphanumeric strings:

```
$ strings main.wasm | grep main
main
hasmain
*main.c0_AKgX
*main.FPJNbG8y
*main.B1698pyli
*[]main.FPJNbG8y
*main.MQljbiUyAfc
*main.VSZ2lqvlggVb
*main.b_QciGbjzfSw
*[]*main.b_QciGbjzfSw
...
main.R3nNAl
main.LWTSNHU
main.WtD0ajazVx55
main.AMlEQDH4
main.(*c0_AKgX).Replace
main.Kd5oA2q
```

We can load `main.wasm` into Ghidra using the ghidra-wasm-plugin. All custom application functions are located at the end of the binary (which can be verified by compiling a small unobfuscated Go WASM binary and comparing how the loader resolves function symbols).

Reversing Tips

- **String & Literal Obfuscation**: Literals are obfuscated by `garble` (garble literal obfuscators).

- **Differential Function Identification**: Compiling a fresh Go WASM program and comparing functions is effective for identifying standard library calls (`crypto/aes`, `crypto/sha256`, `math/big`), since `garble` in this build obfuscates literals without altering function control-flow structure (beyond the state-machine dispatcher naturally emitted by the Go WASM compiler).

Reversing the Pratt Primality Certificate Verifier

Reversing the custom functions at the end of `main.wasm` yields the following pseudocode (with error handling elided and `AES` / `math/big` operations simplified):

```
# Defines 2 node types:
#   Leaf
#   Tree(factors: (Leaf|Tree) list, val: Bignum, a: Bignum)

# (inlined function)
value(Leaf) = 2
value(Tree(factors, val, a)) = val

# 0x8040d664
def deserialize(b):
    if b[0] == 0:
        return Leaf
    if b[0] == 1:
        numFactors = readU32(b)
        factors = [None] * numFactors
        for i in range(numFactors):
            factorLen = readU32(b)
            factors[i] = deserialize(readBytes(b, factorLen))
        valLen = readU32(b)
        val = bignumFromBytes(readBytes(b, valLen))
        aLen = readU32(b)
        a = bignumFromBytes(readBytes(b, aLen))
        return Tree(factors, val, a)

# 0x8040ed35
def testA(a, n, factors):
    if pow(a, n - 1, n) != 1:
        return False
    for f in factors:
        if pow(a, (n - 1) // value(f), n) == 1:
            return False
    return True

# 0x8040f237
# verify(Leaf) = True
def verify(Tree(factors, val, a)):
    prev = bigint(1)
    product = bigint(1)
    for f in factors:
        # verify each factor recursively
        if not verify(f):
            return False
        # check factors are non-decreasing
        factorVal = value(f)
        if factorVal < prev:
```

```
            return False
        prev = factorVal
        product *= factorVal
        # check that a is a Lucas-Lehmer / Pratt primitive root witness and a is minimal
        if not testA(a, val, factors):
            return False
        for b in range(2, a):
            if testA(b, val, factors):
                return False
    if product != val - 1:
        return False
    return True

# 0x8040fadf
def check(s):
    decoded = base64_decode(s)
    if len(decoded) % 16 != 0:
        return "Bad input"
    key = [0xca, 0xbe, 0xba, 0x7a, 0xd1, 0x56, 0x80, 0x1d, 0x1c, 0x82, 0x38, 0x1c, 0x7c, 0xb8,
0xa3, 0xc4, 0x06, 0xae, 0x2c, 0x3c, 0xd1, 0x68, 0xa9, 0x6c, 0xd8, 0x37, 0x36, 0x84, 0xcd, 0xbf,
0x87, 0x23]
    iv = [0x5f, 0xf2, 0x60, 0x40, 0xc4, 0xc2, 0x0b, 0x9f, 0x2f, 0xfd, 0xef, 0xa7, 0xec, 0xf6, 0x5d,
0x91]

    decrypted = aes_cbc_decrypt(decoded, key, iv)
    hash_digest = sha256(decrypted)

    cert = deserialize(decrypted)
    if not verify(cert):
        return "Bad input"

    targetValue = bignum([0x61, 0xe2, 0x30, 0xc9, 0x79, 0xe1, 0xbe, 0x78, 0x55, 0x1b, 0x7c, 0x44,
0x96, 0x3c, 0x5a, 0x65, 0xe3, 0xe8, 0x27, 0xb3, 0xb0, 0x4a, 0x7a, 0xda, 0xcc, 0x02, 0x9a, 0x37,
0xbb, 0xdf, 0x89, 0xd3])
    if value(cert) != targetValue:
        return "Bad input"

    encFlag = [0x78, 0xb9, 0x5d, 0x28, 0x4f, 0xb5, 0xa2, 0x32, 0xdf, 0x1c, 0xcc, 0x90, 0x8e, 0x10,
0xfe, 0x1a, 0x2d, 0x7f, 0xaa, 0x97, 0xcc, 0x43, 0xc2, 0x79, 0x70, 0x87, 0xf0, 0x3a, 0xd3, 0x11,
0xc5, 0x12, 0x99, 0x7c, 0xd0, 0xe4, 0xfb, 0x72, 0xca, 0x30, 0x72, 0xc3, 0xe2, 0xad, 0x0d, 0xe6,
0x23, 0x2f, 0xf6, 0x19, 0x63, 0x27, 0x97, 0xe3, 0x70, 0x8a, 0xd2, 0x9b, 0x2c, 0xdd, 0x44, 0x91,
0xa3, 0xc2]
    return aes_cbc_decrypt(encFlag, hash_digest, iv)
```

This algorithm verifies **Pratt primality certificates**, with canonicalization constraints ensuring that the valid certificate for `targetValue` is strictly unique (prime factors of n - 1 are sorted in non-decreasing order, each primitive root witness a is minimal, and little-endian bignums have no trailing zero bytes).

Generating the Primality Certificate & Flag Decryption

The following Python script constructs the canonical Pratt primality certificate for `p`, encrypts it under the hardcoded AES-256-CBC key, and outputs the Base64 payload expected by `window.check` (requires `pycryptodome` and the `factor` coreutils binary):

```python
import base64
import subprocess
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad, pad

def get_prime_factors(n):
    # shell out to `factor`
    return [int(i) for i in subprocess.check_output(['factor', str(n)]).decode().split(':
')[1].split()]

def decompose(n):
    if n == 2:
        return 2
    factors = sorted(get_prime_factors(n - 1))
    for a in range(2, n):
        if pow(a, n - 1, n) == 1 and all(pow(a, (n - 1) // f, n) != 1 for f in factors):
            break
    return {
        'n': n,
        'a': a,
        'factors': [decompose(i) for i in get_prime_factors(n - 1)]
    }

def p32(x):
    return x.to_bytes(4, 'little')

def serialize(x):
    if x == 2:
        return b'\x00'
    factors = b''
    for f in x['factors']:
        f = serialize(f)
        f = p32(len(f)) + f
        factors += f
    factors = p32(len(x['factors'])) + factors
    val = x['n'].to_bytes(9999, 'little').rstrip(b'\x00')
    val = p32(len(val)) + val
    a = x['a'].to_bytes(9999, 'little').rstrip(b'\x00')
    a = p32(len(a)) + a
    return b'\x01' + factors + val + a

p = 9568161323574474011980796101886568805134018369657076115341935198710665844387 3
serialized = serialize(decompose(p))

iv = bytes([0x5f, 0xf2, 0x60, 0x40, 0xc4, 0xc2, 0x0b, 0x9f, 0x2f, 0xfd, 0xef, 0xa7, 0xec, 0xf6,
0x5d, 0x91])
key = bytes([0xca, 0xbe, 0xba, 0x7a, 0xd1, 0x56, 0x80, 0x1d, 0x1c, 0x82, 0x38, 0x1c, 0x7c, 0xb8,
0xa3, 0xc4, 0x06, 0xae, 0x2c, 0x3c, 0xd1, 0x68, 0xa9, 0x6c, 0xd8, 0x37, 0x36, 0x84, 0xcd, 0xbf,
0x87, 0x23])
cipher = AES.new(key, AES.MODE_CBC, iv)
enc = cipher.encrypt(pad(serialized, AES.block_size))
print(base64.b64encode(enc).decode())
```

Plugging the generated Base64 string into the extension (or directly decrypting `encFlag` with `sha256(serialized)`) yields the final flag:
`Pr4tt_Pr1M4lItY_C3rt5_4nd_Wa4aA4aAA4aAA4a4a5M@flare-on.com`.