Flare-On 11 Challenge 7: fullspeed

By Sam Kim

Overview

We can determine that the binary is a .NET native AOT binary since it has the `.managed` and `.hydrated` sections. For ease of reversing, we can take a memory dump after `.hydrated` is filled out with rehydrated data so we have access to strings.
By searching some of the strings used, we can figure out that this binary uses BouncyCastle for cryptography.

If we look at some of the more suspicious strings that are hex encoded, we see that they are passed to a function at `0x140107D80`, which decrypts them with a simple XOR variant. The list of decrypted strings is below in Figure 1:

```
Unset
c90102faa48f18b5eac1f76bb40a1b9fb0d841712bbe3e5576a7a56976c2baeca47809765283aa078583e1e65172a
3fd
a079db08ea2470350c182487b50f7707dd46a58a1d160ff79297dcc9bfad6cfc96a81c4a97564118a40331fe0fc13
27f
9f939c02a7bd7fc263a4cce416f4c575f28d0c1315c4f0c282fca6709a5f9f7f9c251c9eede9eb1baa31602167fa5
380
087b5fe3ae6dcfb0e074b40f6208c8f6de4f4f0679d6933796d3b9bd659704fb85452f041fff14cf0e9aa7e45544f
9d8
127425c1d330ed537663e87459eaa1b1b53edfe305f6a79b184b3180033aab190eb9aa003e02e9dbf6d593c5e3b08
182
1337133713371337133713371337133713371337133713371337133713371337133713371337133713371337133713371
337
null
inf
inf
verify
verify failed
verify
null
too long
null
|
cd
ok
ls
=== dirs ===
.
```

```
=== files ===
.
cat
exit
bad cmd
err
192.168.56.103;31337
;
err
```

Figure 1: decrypted strings

Looking at the xrefs to this function, we see a lot of interesting code. With extensive cross-referencing of strings and the [source of BouncyCastle] to figure out library functions, we can figure out that the binary connects to `192.168.56.103:31337` and performs an ECDH key exchange with a randomly generated 128-bit private key using the curve y2=x3+ax+b over ℤ/pℤ and the generator point Gx,Gy, where the following values are set from Figure 2:

```python
p =
0xc90102faa48f18b5eac1f76bb40a1b9fb0d841712bbe3e5576a7a56976c2baeca47809765283aa078583e1e6517
2a3fd
a =
0xa079db08ea2470350c182487b50f7707dd46a58a1d160ff79297dcc9bfad6cfc96a81c4a97564118a40331fe0fc
1327f
b =
0x9f939c02a7bd7fc263a4cce416f4c575f28d0c1315c4f0c282fca6709a5f9f7f9c251c9eede9eb1baa31602167f
a5380
gx =
0x087b5fe3ae6dcfb0e074b40f6208c8f6de4f4f0679d6933796d3b9bd659704fb85452f041fff14cf0e9aa7e4554
4f9d8
gy =
0x127425c1d330ed537663e87459eaa1b1b53edfe305f6a79b184b3180033aab190eb9aa003e02e9dbf6d593c5e3b
08182
```

Figure 2: Initial Generator values

All values passed during the key exchange are xored with `0x133713371337133713371337133713371337133713371337133713371337133713371337133713371337133713371337`. We then convert the shared secret to bytes, take the SHA256, then use bytes 0~32 as a ChaCha20 key and bytes 32~40 as the nonce for further communication. The client and server then both send `verify` encrypted to check that the derived key is correct before entering the command loop. The binary supports 3 commands: `ls`, which lists files, `cat`, which dumps a file in Base64, and `exit`, which exits.

Note that it's very unusual that the randomly generated private key is only 128 bits instead of the full 384 bits available! We'll see how this helps later.

Let's check what the order of the generator is using SageMath:

```python
sage: p =
0xc90102faa48f18b5eac1f76bb40a1b9fb0d841712bbe3e5576a7a56976c2baeca47809765283aa078583e1e6517
2a3fd
....: a =
0xa079db08ea2470350c182487b50f7707dd46a58a1d160ff79297dcc9bfad6cfc96a81c4a97564118a40331fe0fc
1327f
....: b =
0x9f939c02a7bd7fc263a4cce416f4c575f28d0c1315c4f0c282fca6709a5f9f7f9c251c9eede9eb1baa31602167f
a5380
....: gx =
0x087b5fe3ae6dcfb0e074b40f6208c8f6de4f4f0679d6933796d3b9bd659704fb85452f041fff14cf0e9aa7e4554
4f9d8
....: gy =
0x127425c1d330ed537663e87459eaa1b1b53edfe305f6a79b184b3180033aab190eb9aa003e02e9dbf6d593c5e3b
08182
....: E = EllipticCurve(Zmod(p), [a,b])
....: gen = E(gx,gy)
....: factor(gen.order())
35809 * 46027 * 56369 * 57301 * 65063 * 111659 * 113111 *
7072010737074051173701300310820071551428959987622994965153676442076542799542912293
```

The order is almost smooth! If it were smooth, then we could use Pohlig-Hellman to recover the private key, but unfortunately we still have a large prime factor in the order. However, we can still apply Pohlig-Hellman partially to gain information about the private key:

Recall that Pohlig-Hellman does the following (using multiplicative group notation):

- Given: generator g in a group of order `n`, element `h = secret*g` where `secret < n` is secret.
- For each prime (power) p in the factorization of the order:
    - Compute `g2 = (n/p)*g, h2 = (n/p)*g`.
    - Solve the discrete logarithm problem using `g2` as the generator and `h2`. This subgroup has order p.
    - The solution to the discrete logarithm problem is `secret` modulo p.
- Combine all the values for `secret` modulo `p` using the Chinese remainder theorem to recover `secret`.

We can apply these steps to the small prime factors of the order to get the private key modulo the product of those factors! If the product was larger than the private key, then this would just be the private key and we would be done. Unfortunately, we still come up short: the product is only 112 bits, not the full 128 bits we need.

However, note that 112 is not much smaller than 128. We can do a ~2^16 brute force to recover the private key as follows: we know the secret modulo n, where n is approximately 112 bits. We also know the secret is up to 128 bits. Then we know `secret = secret_modulo_n + k*n`, where k is at most 128-112 = 16 bits. We can then bruteforce k to get candidates for secret, and test those candidates by deriving the ChaCha20 keys/nonces and decrypting the encrypted traffic with them.

Full solution script:

Python
```python
from sage.all import EllipticCurve, Zmod, factor, Factorization
from scapy.all import rdpcap, Raw
from Crypto.Cipher import ChaCha20
import hashlib
import base64
pcap = rdpcap("capture.pcapng")
raw = b''

for pkt in pcap:
    if Raw in pkt:
        raw += pkt[Raw].load

p =
0xc90102faa48f18b5eac1f76bb40a1b9fb0d841712bbe3e5576a7a56976c2baeca47809765283aa078583e1e6517
2a3fd
a =
0xa079db08ea2470350c182487b50f7707dd46a58a1d160ff79297dcc9bfad6cfc96a81c4a97564118a40331fe0fc
1327f
b =
0x9f939c02a7bd7fc263a4cce416f4c575f28d0c1315c4f0c282fca6709a5f9f7f9c251c9eede9eb1baa31602167f
a5380
genx =
0x087b5fe3ae6dcfb0e074b40f6208c8f6de4f4f0679d6933796d3b9bd659704fb85452f041fff14cf0e9aa7e4554
4f9d8
geny =
0x127425c1d330ed537663e87459eaa1b1b53edfe305f6a79b184b3180033aab190eb9aa003e02e9dbf6d593c5e3b
08182

xorkey =
0x1337133713371337133713371337133713371337133713371337133713371337133713371337133713371337133
71337

E = EllipticCurve(Zmod(p), [a,b])
gen = E(genx,geny)

def unpack(x):
    return int.from_bytes(x, byteorder='big', signed=False)
```

```
ax = unpack(raw[48*0:48*1]) ^ xorkey
ay = unpack(raw[48*1:48*2]) ^ xorkey
bx = unpack(raw[48*2:48*3]) ^ xorkey
by = unpack(raw[48*3:48*4]) ^ xorkey
raw = raw[48*4:]
A = E(ax, ay)
B = E(bx, by)

print('calculating order...')
order = gen.order()
print(f'{order=}')
factors = factor(order, proof=False)
smooth_part = Factorization(factors[:-1]).value()
big_part = Factorization(factors[-1:]).value()

# apply pohlig-hellman partially by dividing out the big factor
secret_mod_smooth = (gen*big_part).discrete_log(A*big_part)
print(f'{secret_mod_smooth=}')

possible_secret = secret_mod_smooth
while True:
    AB = possible_secret * B
    h = hashlib.sha512(int(AB[0]).to_bytes(48, 'big')).digest()
    key = h[:32]
    iv = h[32:40]
    cipher = ChaCha20.new(key=key, nonce=iv)
    test = cipher.decrypt(raw[:6])
    if test == b'verify':
        print('secret found:', possible_secret)
        cipher.seek(0)
        lines = cipher.decrypt(raw).decode().strip('\x00').split('\x00')
        print(lines)
        print(base64.b64decode(lines[-2]))
        break
    possible_secret += smooth_part
```

Solution script output:

```
Unset
calculating order...
order=30937339651019945892244794266256713890440922455872051984762505561763526780311616863989511376879697740787911484829297
```

```
secret_mod_smooth=3914004671535485983675163411331184
secret found: 168606034648973740214207039875253762473
['verify', 'verify', 'ls', '=== dirs ===\r\nsecrets\r\n=== files ===\r\nfullspeed.exe\r\n',
'cd|secrets', 'ok', 'ls', '=== dirs ===\r\nsuper secrets\r\n=== files ===\r\n', 'cd|super
secrets', 'ok', 'ls', '=== dirs ===\r\n.hidden\r\n=== files ===\r\n', 'cd|.hidden', 'ok',
'ls', "=== dirs ===\r\nwait, dot folders aren't hidden on windows\r\n=== files ===\r\n",
"cd|wait, dot folders aren't hidden on windows", 'ok', 'ls', '=== dirs ===\r\n=== files
===\r\nflag.txt\r\n', 'cat|flag.txt', 'RDBudF9VNWVfeTB1cl9Pd25fQ3VSdjNzQGZsYXJlLW9uLmNvbQ==',
'exit']
b'D0nt_U5e_y0ur_Own_CuRv3s@flare-on.com'
```

Final Flag

```
Unset
D0nt_U5e_y0ur_Own_CuRv3s@flare-on.com
```