Flare-On 11 Challenge 3: aray

By Jakub Jozwiak

Overview

This challenge consists of a single file "`aray.yara`" containing one Yara rule named "`aray`". The challenge requires participants to understand and reverse Yara rule conditions to recover a file the rule is designed to match on - file containing a flag.
The challenge can be solved and validated using freely available software including CyberChef, Python and of course Yara.

Solution

At first look, the Yara rule "`aray`" seems to be quite complex - its condition section contains over 500 expressions concatenated using logical AND operators.
The condition section starts with expressions defining size (85 bytes) and MD5 hash (b7dc94ca98aa58dabb5404541c812db2) of a file it matches on and that needs to be recovered.

The complexity of the rule can be reduced by placing each expression in a separate line, sorting them and observing that:

- There are only 36 expressions directly matching file content using the equality operator ("==")
- The expressions using bitwise AND operator (“`uint8(X) & 128 == 0`”) are always true for bytes with values below 128
- The expressions using the remainder operator (”`uint8(X) % Y < Y`”) are always true
- The expressions using the less than (<) and greater than (>) operators are redundant and only provide additional information about expected byte value ranges which on average fall into printable characters range
- The expressions using the not equal (“!=”) operators are also redundant to the set of 36 expressions using the equality operator

```
Unset
$ tail -n 2 aray.yara | head -n 1 | sed 's/and /\n/g' | sort
...
filesize ^ uint8(84) != 231
filesize ^ uint8(84) != 3
```

```
filesize ^ uint8(9) != 164
filesize ^ uint8(9) != 5
hash.crc32(34, 2) == 0x5888fc1b
hash.crc32(63, 2) == 0x66715919
hash.crc32(78, 2) == 0x7cab8d64
hash.crc32(8, 2) == 0x61089c5c
hash.md5(0, 2) == "89484b14b36a8d5329426a3d944d2983"
hash.md5(0, filesize) == "b7dc94ca98aa58dabb5404541c812db2"
hash.md5(32, 2) == "738a656e8e8ec272ca17cd51e12f558b"
hash.md5(50, 2) == "657dae0913ee12be6fb2a6f687aae1c7"
hash.md5(76, 2) == "f98ed07a4d5f50f7de1410d905f1477f"
hash.sha256(14, 2) == "403d5f23d149670348b147a15eeb7010914701a7e99aad2e43f90cfa0325c76f"
hash.sha256(56, 2) == "593f2d04aab251f60c9e4b8bbc1e05a34e920980ec08351a18459b2bc7dbf2f6"
uint32(10) + 383041523 == 2448764514
uint32(17) - 323157430 == 1412131772
uint32(22) ^ 372102464 == 1879700858
uint32(28) - 419186860 == 959764852
uint32(3) ^ 298697263 == 2108416586
uint32(37) + 367943707 == 1228527996
uint32(41) + 404880684 == 1699114335
uint32(46) - 412326611 == 1503714457
uint32(52) ^ 425706662 == 1495724241
uint32(59) ^ 512952669 == 1908304943
uint32(66) ^ 310886682 == 849718389
uint32(70) + 349203301 == 2034162376
uint32(80) - 473886976 == 69677856
uint8(0) % 25 < 25
uint8(0) & 128 == 0
uint8(0) < 129
uint8(0) > 30
uint8(1) % 17 < 17
uint8(1) & 128 == 0
uint8(1) < 158
uint8(1) > 19
uint8(10) % 10 < 10
uint8(10) & 128 == 0
uint8(10) < 146
uint8(10) > 9
uint8(11) % 27 < 27
uint8(11) & 128 == 0
uint8(11) < 154
uint8(11) > 18
uint8(12) % 23 < 23
uint8(12) & 128 == 0
uint8(12) < 147
uint8(12) > 19
uint8(13) % 27 < 27
uint8(13) & 128 == 0
```

```
uint8(13) < 147
uint8(13) > 21
uint8(14) % 19 < 19
uint8(14) & 128 == 0
uint8(14) < 153
uint8(14) > 20
uint8(15) % 16 < 16
uint8(15) & 128 == 0
uint8(15) < 156
uint8(15) > 26
uint8(16) % 31 < 31
uint8(16) & 128 == 0
uint8(16) < 134
uint8(16) > 25
uint8(16) ^ 7 == 115
...
```

Figure 1: Expressions within the Yara Rule

Based on the above observations only the 36 expressions using the equality operator need to be solved to recover the matching file. The expressions to solve consist of:

- CRC32, MD5 and SHA256 check for 2 byte file fragments. Bytes must be brute-forced using known hash values. Majority of hash values can be also looked up in publicly available rainbow tables.
- Single variable equations using addition, subtraction and bitwise XOR operators (minding the endianness of values extracted using the "*`uint`*" functions)

Below in Figure 2 is a simple Python script that uses regular expressions to extract Yara condition expressions, solves extracted equations and brute-forces extracted hash values:

Python

```python
#!/usr/bin/env python3.8

import re
import sys
import struct
from zlib import crc32
from hashlib import md5, sha256

RE_FILESIZE = re.compile(r"filesize\s==\s(?P<filesize>\d{1,2})\s")
RE_HASH = re.compile(r"hash\.md5\(0,\sfilesize\)\s==\s\"(?P<hash>[0-9a-f]{32})\"")

RE_ADD8 = re.compile(
```

```
    r"uint8\((?P<idx>\d{1,2})\)\s\+\s(?P<op>\d{1,2})\s==\s(?P<val>\d{1,3})"
)
RE_XOR8 = re.compile(
    r"uint8\((?P<idx>\d{1,2})\)\s\^\s(?P<op>\d{1,2})\s==\s(?P<val>\d{1,3})"
)
RE_SUB8 = re.compile(
    r"uint8\((?P<idx>\d{1,2})\)\s-\s(?P<op>\d{1,2})\s==\s(?P<val>\d{1,3})"
)

RE_MD5 = re.compile(
    r"hash\.md5\((?P<idx>\d{1,2}),\s\d{1,2}\)\s==\s\"(?P<hash>[0-9a-f]{32})\""
)
RE_SHA256 = re.compile(
    r"hash\.sha256\((?P<idx>\d{1,2}),\s\d{1,2}\)\s==\s\"(?P<hash>[0-9a-f]{64})\""
)
RE_CRC32 = re.compile(
    r"hash\.crc32\((?P<idx>\d{1,2}),\s\d{1,2}\)\s==\s(?P<hash>0x[0-9a-f]{1,8})"
)

RE_XOR32 = re.compile(
    r"uint32\((?P<idx>\d{1,2})\)\s\^\s(?P<op>\d{1,10})\s==\s(?P<val>\d{1,10})"
)
RE_ADD32 = re.compile(
    r"uint32\((?P<idx>\d{1,2})\)\s\+\s(?P<op>\d{1,10})\s==\s(?P<val>\d{1,10})"
)
RE_SUB32 = re.compile(
    r"uint32\((?P<idx>\d{1,2})\)\s\-\s(?P<op>\d{1,10})\s==\s(?P<val>\d{1,10})"
)


def solve_add8(op, val):
    return struct.pack("B", int(val, 10) - int(op, 10))


def solve_sub8(op, val):
    return struct.pack("B", int(val, 10) + int(op, 10))


def solve_xor8(op, val):
    return struct.pack("B", int(val, 10) ^ int(op, 10))


def solve_md5(md5_hash):
    for i in range(32, 128):
        for j in range(32, 128):
            s = struct.pack("B", i) + struct.pack("B", j)
            if md5(s).hexdigest() == md5_hash:
                return s
```

```
def solve_sha256(sha256_hash):
    for i in range(32, 128):
        for j in range(32, 128):
            s = struct.pack("B", i) + struct.pack("B", j)
            if sha256(s).hexdigest() == sha256_hash:
                return s


def solve_crc32(crc32_hash):
    crc32_hash = int(crc32_hash, 16)
    for i in range(32, 128):
        for j in range(32, 128):
            s = struct.pack("B", i) + struct.pack("B", j)
            if crc32(s) == crc32_hash:
                return s


def solve_xor32(op, val):
    return struct.pack("<I", (int(val, 10) ^ int(op, 10)))


def solve_add32(op, val):
    return struct.pack("<I", (int(val, 10) - int(op, 10)))


def solve_sub32(op, val):
    return struct.pack("<I", (int(val, 10) + int(op, 10)))


FMAP = {
    RE_ADD8: solve_add8,
    RE_XOR8: solve_xor8,
    RE_SUB8: solve_sub8,
    RE_MD5: solve_md5,
    RE_SHA256: solve_sha256,
    RE_CRC32: solve_crc32,
    RE_XOR32: solve_xor32,
    RE_ADD32: solve_add32,
    RE_SUB32: solve_sub32,
}


with open(sys.argv[1], encoding="utf-8") as file:
    for line in file:
        if "filesize ==" in line:
            condition = line
```

```
flag_filesize = int(RE_FILESIZE.search(condition).group("filesize"), 10)
flag_hash = RE_HASH.search(condition).group("hash")
print(f"Flag file size: {flag_filesize}\nFlag MD5: {flag_hash}")

flag = {}

for r in [RE_ADD8, RE_XOR8, RE_SUB8, RE_XOR32, RE_ADD32, RE_SUB32]:
    match_iter = r.finditer(condition)
    for match_obj in match_iter:
        flag_slice = FMAP[r](match_obj.group("op"), match_obj.group("val"))
        flag[int(match_obj.group("idx"), 10)] = flag_slice

for r in [RE_MD5, RE_SHA256, RE_CRC32]:
    match_iter = r.finditer(condition)
    for match_obj in match_iter:
        flag_slice = FMAP[r](match_obj.group("hash"))
        flag[int(match_obj.group("idx"), 10)] = flag_slice

flag_file = b""
for offset in range(0, flag_filesize):
    if offset in flag.keys():
        flag_file += flag[offset]

assert len(flag_file) == flag_filesize
assert md5(flag_file).hexdigest() == flag_hash

print(f"Flag:\n{flag_file.decode('utf-8')}")
o = open("flag.txt","wb")
o.write(flag_file)
o.close()
```

Figure 2: Python script to extract and solve Yara condition expressions

When executed the script recovers the file with MD5 hash "b7dc94ca98aa58dabb5404541c812db2". The resulting file contains another Yara rule with the final flag:

```
Unset
$ python3.8 solve_aray.py aray.yara
Flag file size: 85
Flag MD5: b7dc94ca98aa58dabb5404541c812db2
Flag:
rule flareon { strings: $f = "1RuleADayK33p$Malw4r3Aw4y@flare-on.com" condition: $f }

$ yara -gsm aray.yara flag.txt
```

```
aray [] [description="Matches on b7dc94ca98aa58dabb5404541c812db2"] flag.txt
```

Figure 3: Running yara against the script output

Final Flag

```
Unset
1RuleADayK33p$Malw4r3Aw4y@flare-on.com
```