Solving the T2 '14 challenge – by Ludvig Strigeus

The challenge is in the form of a 19MB file called `ylockpoint-final.zip`. It's contains a Win32 application written with the QT[1] toolkit, that displays a full screen interface with a command prompt. The task of the challenge is to extract a set of hexadecimal SHA hashes that the challenge will present in one way or another.

This year would be more focused on reverse engineering, so I loaded up the binary in IDA[2] and did some basic static analysis. Most of the logic appeared to be handled by the QT/QML[3] code that gets loaded into the main window (through a resource named `main.qml`). At the same time, I noticed that the EXE-file had a nonstandard DOS header appearing to belong to the challenge.

The static analysis had revealed a call early on to QT's `showFullscreen` which I patched into `showNormal` using a hex editor. This runs the program as a normal windowed application instead, to facilitate debugging.

At the end of the binary is a large data blob containing embedded application resources, stored using zlib[4] compression in QT's own resource format[5]. To unpack them, I wrote a Python[6] tool that scans for zlib headers and tries to decompress. I found five files: Three audio files, one qml file and one image. The audio files produce various keypress sounds, and the image is the background image of the application's interface. Through ocular inspection, I found no secret information hidden inside the media files.

```
# Decoder code. 0x76e8 is the offset of the resource section start in the EXE.
data, ctr = file('d:\\t2\\ylockpoint.exe', 'rb').read()[0x76e8:], 0
for i in xrange(len(data) - 10):
  # Zlib header magic numbers
  if data[i] == '\x78' and data[i+1] in ('\xda', '\x9c', '\x01'):
    try:
      s = zlib.decompressobj().decompress(data[i:])
      file('d:\\t2\\out\\%d.dat' % ctr, 'wb').write(s)
      ctr += 1
    except zlib.error:
      pass
```

The program, which has a PI inspired version number 3.14.15.92, accepts the following commands:

```
Available commands:
(h) help Show help
(d) decode Acticate obscure key decoder
(v) view Connect to quntaview live stream
(i) infect Install Astral Breeze monitoring
(q) quit Quit program
```

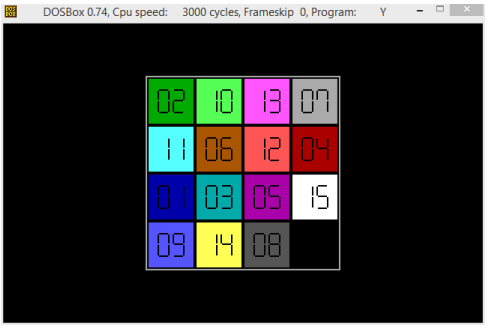
I found some other ones too:

```
y-lockpoint$ make blabla
make: don't know how to make blabla. Stop.
y-lockpoint$ m
mz: Mark Zbikowski is still alive!
```

[1] http://qt-project.org/
[2] https://www.hex-rays.com/products/ida/
[3] http://en.wikipedia.org/wiki/QML
[4] http://en.wikipedia.org/wiki/Zlib
[5] http://qt-project.org/doc/qt-5/resources.html
[6] https://www.python.org/

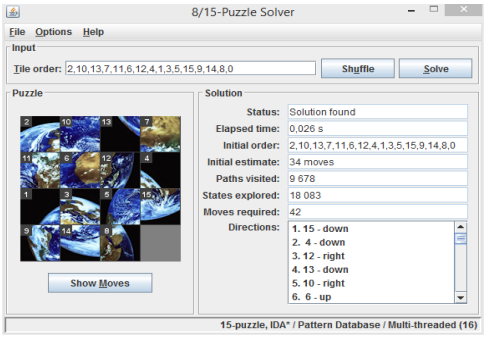
Puzzle of old times

When starting ylockpoint.exe in DOSbox[7], a sliding puzzle game appears. It's controlled using the arrow keys. This game is written in 16-bit DOS assembly and embedded in the DOS header that normally just prints that the application requires Windows ☺.



The static disassembly shows various strings (one including the word SHA) that can be printed as a result of running the game, so probably if you solve the puzzle with the least number of moves the secret string will get printed.

I found a puzzle solver[8] on the Internet that I ran to produce the answer that consisted of 42 steps. When inputting this in the DOS based game it says:




```
SHA1(226268842268684486224486224886624266888444)
after 42 moves.
```

Hashing that string with Python to get the answer:

```
>> s = '226268842268684486224486224886624266888444'
>> sha.new(s).hexdigest()
'692787112272f82c336322dd117992a8c4a36820'
```

Additionally, I found an encrypted embedded string, encrypted using XOR encryption[9] with the key 0xDEADBEEF. It was obvious that the key was 0xDEADBEEF because of embedded NUL characters that got encrypted into the key itself. When decrypting the blob, with NUL characters replaced with `?`, I get:

`DanS??????????SHA1(https://www.youtube.com/watch?v=dQw4w9WgXcQ)???`

Hashing and entering the URL on the web page just says that I found an Easter egg:
`df2c6f7afc1a58783e15f2ae0118ff039d8a4755`
The video is a very motivating video named: Rick Astley – **Never** Gonna **Give** You **Up**.

Hashed keypresses

When using this command of `ylockpoint`:
`(d) decode  Acticate obscure key decoder`
A loop gets started that accepts key presses. Analyzing `main.qml` that I decompressed earlier shows that the entered keys are accumulated into a password string one by one. For every typed character, the partial string is hashed and compared a list of valid hashes. If correct so far, a special sound is played. You could solve this by testing one character at a time by hand and listening to the sounds that get played, but I made a Python solver:

```
def nextchar(s = ''):
    for c in "qwertyuiopasdfghjklzxcvbnm-!:()":
        if sha.new(s + c).hexdigest() in HASHLIST:
            return nextchar(s + c)
    print 'Answer: ' + s
nextchar()
```

Answer: `ssl-added-and-removed-here!:)`
`SHA1("ssl-added-and-removed-here!:)") = d42fbbb299dabe5b089d7f2dbd3303b27dc73b3b`

[7] http://www.dosbox.com/
[8] http://www.brian-borowski.com/software/puzzle/
[9] http://en.wikipedia.org/wiki/XOR_cipher

One step at a time

When using this command of ylockpoint:

```
(i) infect   Install Astral Breeze monitoring
```

A password prompt is shown where you can enter a numeric pin code. This is then passed to a C++ function for verification through QT's JavaScript bindings.

The C++ code has installed a vectored exception handler[10] that is used to single-step the CPU instructions of the PIN code verifier by using the Trap Flag[11], and then a string is decrypted and returned to `main.qml`. After every instruction the instruction pointer is modified to a new location by looking up a value to add from an array. This means that the instructions of the verifier function are not stored linearly in the executable but scattered inside a data blob in the executable. After every instruction: `IP += TABLE_OF_OFFSETS[current_offset++]`

To be able to analyze it, I wrote an IDAPython[12] script that computes the addresses of all the run instructions and linearizes them into a single function in IDA. The resulting function takes the PIN code integer as input, hashes it using SHA1, converts the hash to a hex string, verifies that the hash is correct, and then produces the solution string if the hash is correct.

I copied all the instructions into an inline-assembly function in a new C++ program, and wrote a multithreaded bruteforcing loop that tests all 2^32 = 4 billion combinations against the valid hash. After a short while, my program had found the answer.

```
bool __declspec(naked) _cdecl doit(int input) {
  _asm {
    // All the instructions from the linearized function but changed
    // to return true for a valid hash.
  }
}

DWORD WINAPI Start(PVOID arg) {
  int i = (int)arg << 29, count = 1 << 29;
  do {
    if (doit(i)) { printf("The PIN is: %u\n", i); break; }
  } while (i++, --count);
  return 0;
}

int main(int argc, char* argv[]) {
  DWORD id;
  for (int i = 0; i < 8; i++)
    CreateThread(NULL, 0, &Start, (void*)i, 0, &id);
  while (true)
    Sleep(10);
  return 0;
}
```

The PIN code is `233811181`. Entering this in ylockpoint prints:

```
Access via Belgacom, ID e51b27ad14a0aa4b083eb2fad55b065502c6154e
```

The single step function contains another magic PIN code, `42424242`. This prints:

```
PIN valid for BLACKHEART operations only. Installation aborted.
```

[10] http://msdn.microsoft.com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx
[11] http://en.wikipedia.org/wiki/Trap_flag
[12] https://code.google.com/p/idapython/

The secret location

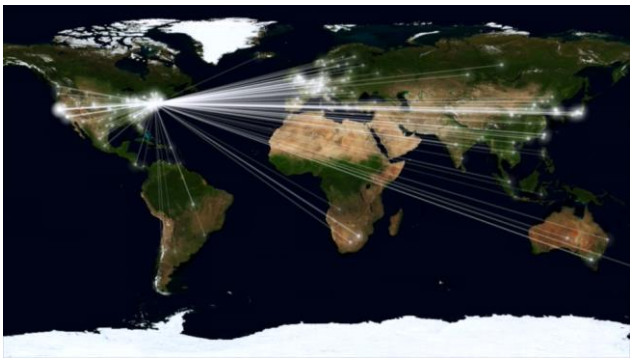
The final challenge uses

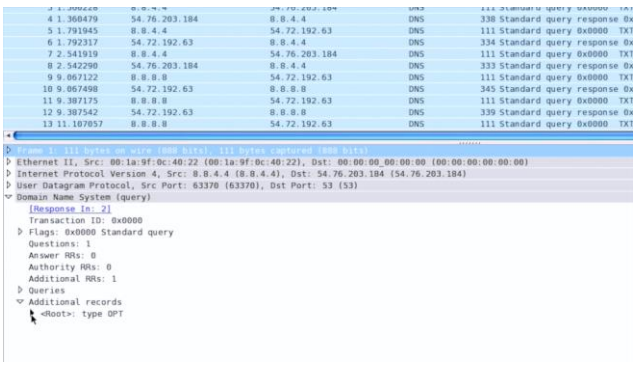
`(v) view Connect to quntaview live stream`

The C++ code / QT binding does a TXT[13] DNS lookup against the hostname `dynamic.t2.whois-servers.org`, and extracts a URL and a text string from the reply. The URL points at a movie file that gets played, while the string is a message that is displayed on the screen.

```
[ludde@freebsd ~]$ host -t txt dynamic.t2.whois-servers.org
dynamic.t2.whois-servers.org is an alias for se.t2.whois-servers.org.
se.t2.whois-servers.org descriptive text "Sweden"
se.t2.whois-servers.org descriptive text "https://dl.dropboxusercontent.com/s/2ow9on75z3h5r46/ylockpoint-6891.mp4?dl=1"
```

This is basically a GeoIP[14] service that maps the user's IP address into a country and a video file. The video itself is well made and gives a hacker feeling with a bunch of packet capture screenshots and countries with blinking lines and funny beeps. Quite cool.





I perform a DNS query on all the 26x26 two-letter country codes to see if any countries get special treatment and I found Afghanistan (af) and Bahamas (bs):

```
af.t2.whois-servers.org descriptive text "youtube:XlWQsVwsTyY"
af.t2.whois-servers.org descriptive text
https://dl.dropboxusercontent.com/s/8bdo5s2y45azv6n/66f0a1a1c7cdb6a7b66cbf7d10772ae8c1238809.mp4?dl=1
af.t2.whois-servers.org descriptive text "Afghanistan. Close, but no cigar. You are not there!"
```

The video is made by Thomas Dolby[15]. Searching for him doesn't reveal anything interesting except for a twitter mentioning NSA. At this point I'm pretty much stuck and I guess various other strings to lookup instead of the country codes, but I find nothing.

Eventually, after a hint was published, I realize that the solution really is to be found in the original video somewhere. At the beginning of the video it shows two outlines of Mexico and Afghanistan. At the very end, the video displays a DNS packet using the Client Subnet[16] extension. I craft a DNS packet using this extension pretending that the query originates from an IP inside of Afghanistan. That did the trick: Now the CNAME points at `somalget-ftw-in-2014.t2.whois-servers.org` with this string:
`SHA1("Connection hijacked by E.S.")` `9afe35a90de076fd497787811d469264b85bd204`

Or use this command to see it live inside of the ylockpoint console:

```
>> v somalget-ftw-in-2014
```

---

[13] http://en.wikipedia.org/wiki/TXT_Record
[14] https://www.maxmind.com/en/geoip2-services-and-databases
[15] http://en.wikipedia.org/wiki/Thomas_Dolby
[16] http://www.ietf.org/archive/id/draft-vandergaast-edns-client-subnet-02.txt

Appendix A – IDAPython Code to linearize the scattered PIN verifier

```
import array, idc
data = file('d:\\t2\\ylockpoint.exe', 'rb').read()
a = array.array('i', data[0x5df8:])
OUT = 0x500000
PC = 0x00403AC2
CNT = 0
idc.MakeUnknown(OUT, 65536, 0)
for i in xrange(4, 2000):
    idc.MakeUnknown(PC, 16, 0)
    len = idc.MakeCode(PC)
    if len <= 0:
        break
    for j in xrange(len):
        idc.PatchByte(OUT, Byte(PC + j))
        OUT += 1
    PC += a[i] + len
    CNT += 1
idc.MakeCode(0x500000)
```

Appendix B – Code to craft a Client Subnet DNS packet

```
import socket

ip = (61,5,192,0) # belongs to Afghanistan

a = "00 00 00 00 00 01 00 00 00 00 00 01".replace(' ', '').decode('hex')
b = "\x07dynamic\x02t2\x0dwhois-servers\x03org\x00"
c = "\x00\x10\x00\x01"
d = "\x00\x00\x29\x0b\xb8\x00\x00\x80\x00\x00\x0c"
e = "\x00\x08\x00\x08\x00\x01\x18\x00"+"".join(chr(v) for v in ip)

m=a+b+c+d+e
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
s.sendto(m, ('54.72.192.63', 53))

# The reply can be seen in a packet capture such as Ethereal.
```

Appendix C – Code to decode the easter egg in the DOS header

```
import array, sha
data = file('d:\\t2\\ylockpoint.exe', 'rb').read()
r = ''
for i in xrange(76):
    c = ord(data[0x250 + i]) ^ ord("\xEF\xBE\xAD\xDE"[i%4])
    r += chr(c) if c >= 32 else '?'
print r
print sha.new('https://www.youtube.com/watch?v=dQw4w9WgXcQ').hexdigest()
```