Solving the t2'14 challenge – by Vladimir Gneushev

Used tools:

HEX editor (your choice),
Text editor (your choice),
CFF Explorer http://www.ntcore.com/exsuite.php
Binwalk http://binwalk.org/
hashcat http://hashcat.net/hashcat/
OllyDbg http://www.ollydbg.de/version2.html
Python https://www.python.org/
WireShark https://www.wireshark.org/
sed https://en.wikipedia.org/wiki/Sed
dig https://en.wikipedia.org/wiki/Dig_(command)

Challenge comes as a beefy 19 megabytes large zip archive. Inside we can find the main executable called ylockpoint.exe (660kb) and a bunch of Qt framework libraries.

1. Old feelings...

Prior running any file it's always a good idea to take a look inside. Loading it in hex editor we can notice quite suspicious stuff right at the beginning. Instead of typical "This program cannot be run in DOS mode" here we can spot several messages like "Nonoptimal win" and "SHA1". Looks like our first stop is here. Let's give it a try and execute this file under DOS. By DOS I mean Dosbox, of course.

So, it appears our hunch was correct. File runs fine and shows rather familiar picture. What was it? Aha, a 15 game! We can slide blocks using numpad and our goal is to solve the puzzle? Not only to solve it, but to find an optimal solution (as was hinted above.) Let's google for "optimal 15 puzzle solver". Among the first few links we can find a nice app to perform this task: http://www.ic-net.or.jp/home/takaken/e/15pz/ download it, run, arrange the blocks just as pictured, hit "Solve" button and...

```
  2 10 13  7
 11  6 12  4
  1  3  5 15
  9 14  8  0

Optimal solution is 42moves.
Used time is 0sec.

 15  4 12 13 10  6  3  5 13 10
  6  3 11  1  5 13  8 14 13 11
 10 12  4  8 12  6  7  4  8 12
 11 10  6  7  3  2  1  5  9 13
 14 15
```

By moving the blocks using this sequence we successfully solved the puzzle and awarded with

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

message and computing the hash (it's the keys we've pressed, btw) we got our first answer:

Answer #1 : 692787112272f82c336322dd117992a8c4a36820

trivia: funny enough, game solving algorithm is called IDA*. Just like some useful tool you may heard of.

protip: when submitting hashes, make sure they are lower-cased or they won't be accepted

2. Ah you little...

Moving along. Not too far along, though. Let's check back the file once again. Right after "SHA1(" string we can notice "Rich" header. Normally it's used to "hide" Microsoft's compiler version used to compile the file and often exhibits repeating 4-byte pattern just before the "Rich" word. However, in our case the pattern is clearly irregular! Smells fishy? It surely is! So, google is our friend again. Quick query for "microsoft rich header" immediately returns detailed description of this obscure feature and also a script to decode it. http://www.ntcore.com/files/richsign.htm Using CFF Explorer from ntcore and this script http://www.ntcore.com/files/richsign/rich_sign_decr.cff we can decode the Rich header and what we see here is... oh noes, we're being rickrolled. So the little easter egg is SHA1(https://www.youtube.com/watch?v=dQw4w9WgXcQ) which is

Answer #easter1 df2c6f7afc1a58783e15f2ae0118ff039d8a4755

3. Serious stuff

All right, enough with the games. And it's time to look... inside the file once again. Last time now, my promise. Skipping through bits of code, nothing of particular interest here. Code ends, some sparse bytes and small strings like ".t2.whois-servers.org", "https:", "qrc:/main.qml" then a bunch of encrypted/packed data almost until the end of file. Seeing this "qrc:/main.qml" reference and no such file next to executable we can assume it's hidden somewhere and indeed, Qt allows storing of resources inside the executable and, more importantly, compress them too! Thankfully, it's good old deflate method, which means it's time for our next tool.

binwalk. Nifty utility that is good at searching for various stuff in files. Let's give it a try.

```
$ binwalk -Me ylockpoint.exe

Target File:   ylockpoint.exe
MD5 Checksum:  a698c7ea9bb2f02e0c7f0f3ecd1fb957
Signatures:    285

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Microsoft portable executable
30448         0x76F0          Zlib compressed data, best compression, uncompressed size
>= 65536
208033        0x32CA1         Zlib compressed data, best compression, uncompressed size
>= 65536
487003        0x76E5B         Zlib compressed data, best compression, uncompressed size
>= 65536
578362        0x8D33A         Zlib compressed data, best compression, uncompressed size
>= 12468
581880        0x8E0F8         Zlib compressed data, best compression, uncompressed size
>= 65536
```

Well, that's exactly what we expected to see. Extracted chunks are still compressed, but trivial python script finishes the job.

```
import zlib
inp = open("8D33A.zlib").read()
out = zlib.decompress(inp)
open("unpacked","w").write(out)
```

Among these files we can see a png picture, 3 wave files and qml script, which is the heart of this challenge.

It appears we have some sort of interactive shell, where we can enter various commands. "help" command lists some, the rest can be found in big "switch (arg[0].toLowerCase())" block. "view", "decode" and "infect" seems important, the rest is just for fun. Oh, "mz" actually references "Mark Zbikowski", but too late...

ok, no real answers here, time to finally run the file and try each command in action.

4. Bleedy typewriter

Let's start with "decode". According to the script, the decoder operates as follows:

```
var ch = event.text                          --- get a keypress
str = str + ch                               --- append it to a string
switch (hashes[astralbreeze.sha1(str)])      --- calculate hash of this string
case 1:                                      --- hash matches, repeat
case 2:                                      --- final hash matches, show a secret hash
default:                                     --- no matching hash, reset
```

So, in essence we need to find a growing sequence of characters that matches provided list of hashes. And for that we'll use another nice tool called hashcat.

First, put all hashes in to a file. Then run hashcat like

```
hashcat-cli64 -m100 -a3 hashes ?a
```

to crack the first character.

```
>> a0f1490a20d0211c997b44bc357e1972deab8ae3:s
```

Aha, it's "s"! Now, next one please!

```
hashcat-cli64 -m100 -a3 hashes s?a
```

```
>> c1c93f88d273660be5358cd4ee2df2c2f3f0e8e7:ss
```

more...

```
hashcat-cli64 -m100 -a3 hashes ss?a
```

```
>> 20a24593f82e573953076a0eeaf8f3cfb817a534:ssl
```

and so on, until we get the full string:

```
d42fbbb299dabe5b089d7f2dbd3303b27dc73b3b:ssl-added-and-removed-here!:)
```

By typing this string in "decode" prompt we'll receive encouraging message

```
obscure-decoder: Key sequence matches password hash
d2c4d3acc29c63a9f27c4fbf6cda4a3448590f5f
```

and our next answer:

Answer #2 : d2c4d3acc29c63a9f27c4fbf6cda4a3448590f5f

5. Astral brute

Ok, that was easy so far. What's next? How about "infect"? It looks like we need another password. This time it's just a number which is passed to astralbreeze.check() method and if the number is correct a string with " ID " part will be returned. Time for some heavy artillery! I mean, hex editor of course. Instead of checking how this astralbreeze works, let's just cheat a little. Open ylockpoint.exe in hex editor, search for "qrc:/main.qml" string (remember, we saw it earlier?) and modify it to "qrc12main.qml". This will cause the said file to be loaded from the disk instead of internally stored one. And we need this to... inject our little brute-force code right in to the script! So, let's do it:

```
  case "password: ":

  // here goes our snippet

    for(var p = 0;p < 0xFFFFFFFF;p++)
    {
       var t = astralbreeze.check(p);
       if(t.indexOf(" ID ") >= 0)
       {
         log("found pin " + p);
         text = p;
         break;
       }
    }

 // the rest continues unmodified
```

Now, run our modified executable, invoke 'infect' job, enter any number, hit Enter and... ahem. Well, it does seem to work, but waaaaay too slow. Since such dumb approach has failed, let's try something different. A debugger.

Our choice of debugger is Ollydbg. Small, nice and packed with useful features.
http://www.ollydbg.de/odbg201.zip

Start the debugger, open challenge executable and hit Run, then go to "infect" again, type in some number (like, 123) and press enter. That's interesting. Execution is stopped at the command int 3 (if, for some reason it doesn't, check Ollydbg's Options - Exceptions settings, Ignore INT3 breaks must be clear.) Also, EAX register seems to hold the password number we just entered. However, if we just attempt to single-step the code from here, all we get is the message

```
Attempt to run with DEWSWEEPER. Security policy violation. Sop.
```

Not good. So it's seems that int 3 command triggers some hidden action, which is skipped if debugger handles int 3 by itself. Let's try it again. Put a regular breakpoints at the beginning and end of this function (like, at 404351 and 4043C0) so we won't miss it again. Now go to Options - Exceptions and set all "Ignore the following exception" options and also set "Report ignored exceptions to log". Apply changes and resume execution. Go to "infect", enter some number again, hit Enter. After hitting our breakpoint at the beginning of the function, continue execution. Something's changed? Yes, we've skipped int 3 opcode successfully and stopped at our second breakpoint at the end of function.

Let's check the log window (View -> Log)

```
00404351  Breakpoint at ylockpoint.00404351
0040437C  INT3 command at ylockpoint.0040437C - passed to application
00404381  INT3 command at ylockpoint.00404381 - passed to application
00404388  Break on single-step trap or INT1 set by application - passed to application
00403AC5  Break on single-step trap or INT1 set by application - passed to application
00403FB6  Break on single-step trap or INT1 set by application - passed to application
              ....
004034D3  Break on single-step trap or INT1 set by application - passed to application
00404389  Break on single-step trap or INT1 set by application - passed to application
004043C0  Breakpoint at ylockpoint.004043C0
```

So it appears application handles int 3 breaks by itself and also use some sort of self-tracing. And if we check the code at the adress where that self-tracing occurs we can see that it's filled with some garbage. Since int 1 breaks occur right after the command execution, it must be located just before the address from the log window.

Time to use another Ollydbg's excellent feature. If we follow one of the int 1 addresses in Debugger's CPU Dump window and scroll around a bit, we can recognize the beginning (4031B0) and end (4041AF) of this "junk" area. Now select this whole area with the mouse, click the right button and choose Breakpoint - Memory log. Check "Execution", Pause program - Never, Log value of expression - Always. Apply. Next go to Trace - Set protocol. Select "Protocol only the following EIP ranges", "Add range" and enter same addresses here. Go to Trace again, select "Open run trace" (new window will appear) and resume execution of the process. Again, "infect", some number, Enter. Continue after first breakpoint and stop at the second. If we check the Run trace window now, we'll see it full of executed commands. Let's save it to a file (Right click, Log to file, Add available contents).

```
main  00403AC2                  SUB ESP,50                                EAX=0000007B
main  00403AC5                  INS BYTE PTR ES:[EDI],DX                  EAX=0000007B
main  00403FB2                  MOV DWORD PTR SS:[ESP+40],EDX             EAX=0000007B
main  00403FB6                  CDQ                                       EAX=0000007B
main  00403CA3                  MOV DWORD PTR SS:[ESP+44],ESI             EAX=0000007B
main  00403CA7                  CMPS BYTE PTR DS:[ESI],BYTE PTR ES:[EDI];EAX=0000007B
main  00403875                  MOV DWORD PTR SS:[ESP+48],EDI             EAX=0000007B
...
main  00403DC4                  FCOM QWORD PTR DS:[ESI-77]                EAX=00000105
main  00403A80                  MOV EBP,DWORD PTR SS:[ESP+4C]             EAX=00000105
main  00403A84                  TEST ESP,EDI                              EAX=00000105
main  00403DA5                  ADD ESP,50                                EAX=00000105
main  00403DA8                  AND EAX,62435BD0                          EAX=00000105
main  004034D0                  JNE SHORT 004034D3                        EAX=00000105
```

Looks very interesting, but a bit dirty. It appears Ollydbg logs the command that just executed and the next one (which is a garbage) after it. Let's clean the list by throwing away every second line.

```
sed -n '1~2p' tracelog.txt > clean.txt
```

```
main  00403AC2                  SUB ESP,50                                EAX=0000007B
main  00403FB2                  MOV DWORD PTR SS:[ESP+40],EDX             EAX=0000007B
main  00403CA3                  MOV DWORD PTR SS:[ESP+44],ESI             EAX=0000007B
main  00403875                  MOV DWORD PTR SS:[ESP+48],EDI             EAX=0000007B
...
```

Much better now.

But what is it? Those magic numbers are really familiar... Aha! It's just another SHA1 calculation over single block. First 4 bytes of it is our password, next four is 0xB16B00B5 and the rest is regular SHA1 padding/size bits. Correct hash value must be:

```
main  004034E1                  CMP ESI,E51B27AD                        EAX=00000000
main  004037C3                  SETNE AH                                EAX=00000000
main  00403725                  ADD AL,AH                               EAX=00000100
main  00403591                  CMP EBX,14A0AA4B                        EAX=00000101
...

E51B27AD14A0AA4B083EB2FAD55B065502C6154E
```

Now we can easily bruteforce the input value.

```
import struct
import hashlib

data=bytearray(8)
data[4:8]=struct.pack(">I", 0xB16B00B5)

for i in xrange(0, 0xFFFFFFFF):
       data[0:4]=struct.pack(">I", i)
       if hashlib.sha1(data).hexdigest() ==
"e51b27ad14a0aa4b083eb2fad55b065502c6154e":
             print "found pin ", i
             break
```

It's turns out to be 233811181 (or 0xDEFACED) which awards us with another message:

```
infect-astral-breeze: Access via Belgacom, ID e51b27ad14a0aa4b083eb2fad55b065502c6154e
```

and that's our goal.

Answer #3 : e51b27ad14a0aa4b083eb2fad55b065502c6154e

behind the scenes: Also try password 42424242 for more NSA jokes.

6. Watch me carefully

All right. Final task. According to the qml script, "view" command accepts extra parameter. When omitted, "dynamic" is used instead. It's passed to quantumsource.lookupInfo() method which then invokes callback function with two additional arguments: url and msg. Url later used to load and play a video file. Msg displayed as "location" and, if contains "SHA1" substring is our goal. Since this part of the challenge seems to be network-related, let's run our trusty Wireshark and see what's going on here.

Aha, it looks like "view" command sends TXT DNS query to dynamic.t2.whois-servers.org or another sub-domain if supplied. And returned answer contains url of a video file.

Let's replicate the query using dig tool:

```
$ dig TXT dynamic.t2.whois-servers.org

;; QUESTION SECTION:
;dynamic.t2.whois-servers.org.        IN      TXT

;; ANSWER SECTION:
dynamic.t2.whois-servers.org. 5       IN      CNAME   ru.t2.whois-servers.org.
ru.t2.whois-servers.org. 5    IN      TXT     "Russian Federation"
ru.t2.whois-servers.org. 5    IN      TXT
        "https://dl.dropboxusercontent.com/s/2ow9on75z3h5r46/ylockpoint-6891.mp4?dl=1"
```

So far so good, but nowhere close to the goal. In the video we can see some sort of a network monitoring log, with a lot of DNS TXT request from all around the world like we just did ourselves.

Oh, we also have a hint! Two in fact. "Pay close attention to first 4 seconds and last 2 seconds of the video." In first few seconds of the video we can see three irregular geometric shapes, looks like someone spilled black ink. Let's find out what it is. Screengrab the image, crop individual shapes and upload them to google image search. That was easy. That's not the ink, but rather world's map cut-outs! Rightmost one is Mexico, middle is Afghanistan and leftmost one is Bahamas. Let's try it again. Mexico zone is mx, Afghanistan is af, and Bahamas is bs, according to https://en.wikipedia.org/wiki/ISO_3166-1.

```
$ dig TXT mx.t2.whois-servers.org

mx.t2.whois-servers.org. 5    IN      TXT     "Mexico"
mx.t2.whois-servers.org. 5    IN      TXT
        "https://dl.dropboxusercontent.com/s/2ow9on75z3h5r46/ylockpoint-6891.mp4?dl=1"

$ dig TXT af.t2.whois-servers.org

af.t2.whois-servers.org. 5    IN      TXT     "Afghanistan. Close, but no cigar. You are
not there!"
af.t2.whois-servers.org. 5    IN      TXT     "youtube:XlWQsVwsTyY"
af.t2.whois-servers.org. 5    IN      TXT
        "https://dl.dropboxusercontent.com/s/8bdo5s2y45azv6n/66f0a1a1c7cdb6a7b66cbf7d10772
ae8c1238809.mp4?dl=1"

$ dig TXT bs.t2.whois-servers.org

bs.t2.whois-servers.org. 5    IN      TXT     "Bahamas. Close, but no cigar. You are not
there!"
bs.t2.whois-servers.org. 5    IN      TXT     "youtube:XlWQsVwsTyY"
bs.t2.whois-servers.org. 5    IN      TXT
        "https://dl.dropboxusercontent.com/s/8bdo5s2y45azv6n/66f0a1a1c7cdb6a7b66cbf7d10772
ae8c1238809.mp4?dl=1"
```

That's better, but no cigar. Time for another clue!

Last few seconds of the video show close-up of several DNS TXT requests from google's public DNS server 8.8.8.8. Nothing particularly interesting here, except... Let's rewind the video back to 0:13. In the long list of request we can see the same requests from 8.8.8.8 ip. But what's more important, if we check the rightmost column, we can also notice that despite requests arriving from the same IP and to the same dynamic.t2. server, the resolved geographic subdomain is different every time. So this means it's possible to somehow specify the desired zone in request itself. Now go forward to the last few sections of the video again. Here we can see some "CSUBNET - Client subnet" option. Let's try to send such request. (Of course, we can try to find some Bahama's vpn/proxy, but...)

Bummer. It appears by default dig doesn't provide such functionality. Time to ask google for help again. Query for "dig client subnet" easily finds the solution. https://www.gsic.uva.es/~jnisigl/dig-edns-client-subnet.html

Let's follow the instructions, recompile the dig with edns patch, and run it again with some IP from Afghanistan. Another query to google for "afghanistan ip range" brings us to this table http://www.nirsoft.net/countryip/af.html with suggested IP range 58.147.128.0 through 58.147.159.255 . Let's try it now.

```
$ ./dig TXT dynamic.t2.whois-servers.org +client=58.147.128.123 @8.8.8.8

;dynamic.t2.whois-servers.org.       IN     TXT

;; ANSWER SECTION:
dynamic.t2.whois-servers.org. 299 IN  CNAME   somalget-ftw-in-2014.t2.whois-servers.org.
somalget-ftw-in-2014.t2.whois-servers.org. 299 IN TXT "^C^CSHA1(\"Connection hijacked by
E.S.\") 9afe35a90de076fd497787811d469264b85bd204"
somalget-ftw-in-2014.t2.whois-servers.org. 299 IN TXT "youtube:o66FUc61MvU"
somalget-ftw-in-2014.t2.whois-servers.org. 299 IN TXT
"https://dl.dropboxusercontent.com/s/tgyy82dv1uujajq/f184a821ad6eb33fdb4446e9b7d5a712f55f
1d40.mp4?dl=1"
```

A-and we're done here.

Answer #4: 9afe35a90de076fd497787811d469264b85bd204

Tha't all folks!