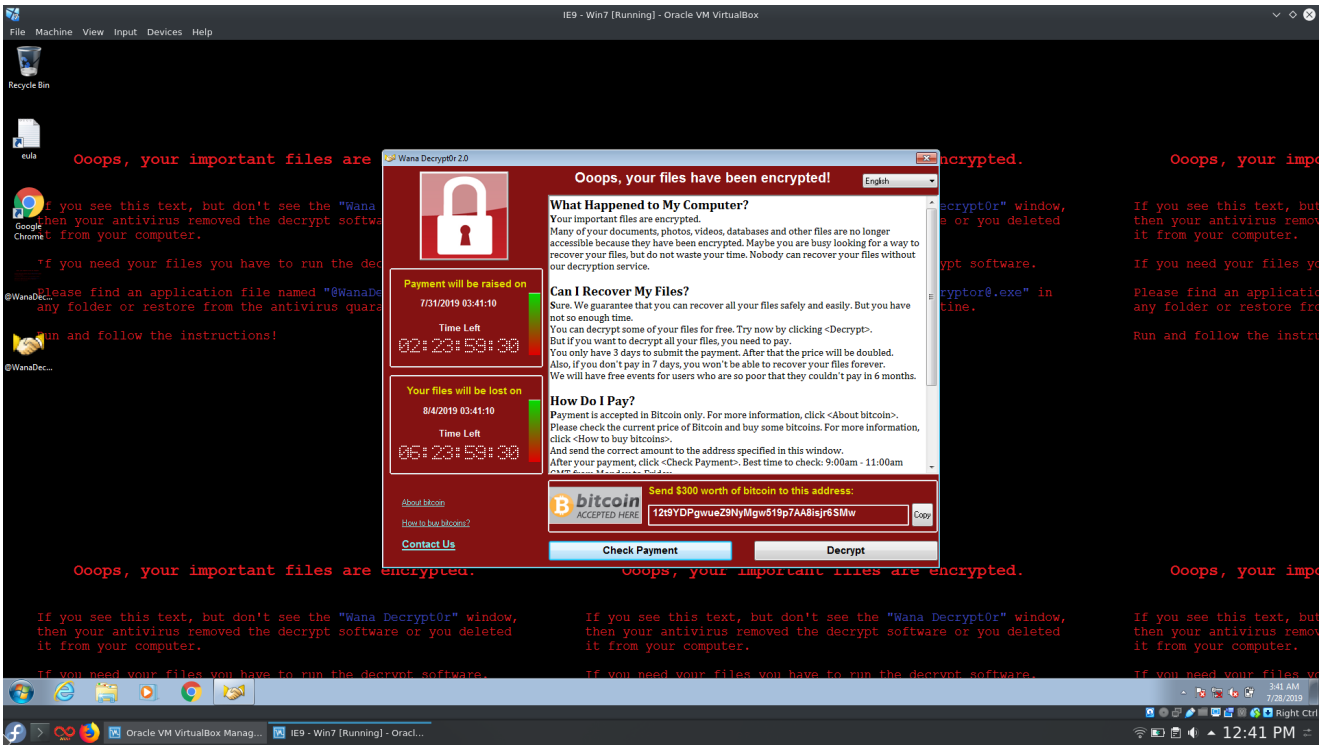
Third time's the charm? Analysing WannaCry samples

dissectingmalwa.re/third-times-the-charm-analysing-wannacry-samples.html

Sun 28 July 2019 in Ransomware

After over two years since the inital spread of the ransomware and Malwaretechs sentencing last week I got a bit nostalgic and took a second look at different samples



Since the first wave of infections in May 2017 WannaCry is basically the goto example for the whole ransomware scheme and that is actually a good thing. The potential damage that WannaCry and the variants following the original version would have been massive if it wouldn't have been for Malwaretech, 2sec4u and all the other researchers who helped to contain the spread of ransomware powered by the wormable EternalBlue exploit. Funnily enough there are still people from around the world that pay the ~300$ ransom in hopes to get their data decrypted as can be seen here.

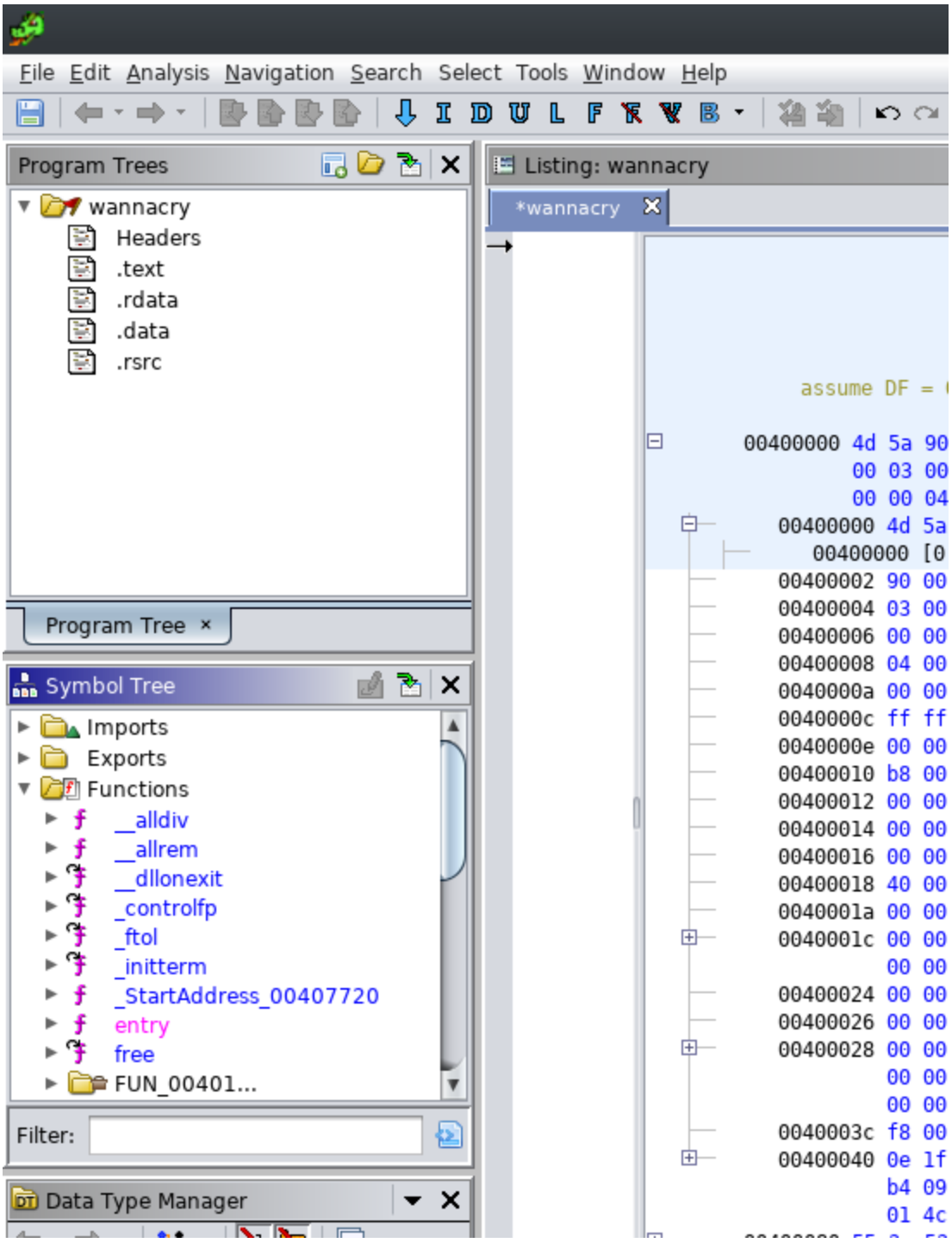
A general disclaimer as always: downloading and running the samples (especially the ones without the kill switch) linked below will lead to the encryption of your personal data, so be f$cking careful. Also check with your local laws as owning malware binaries/ sources might be illegal depending on where you live.

The three samples I'll be looking at:

Wannacry Sample #1 sometimes referred to as "dropper" available @ https://www.ghidra.ninja/samples/wannacry.zip `sha256 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c`

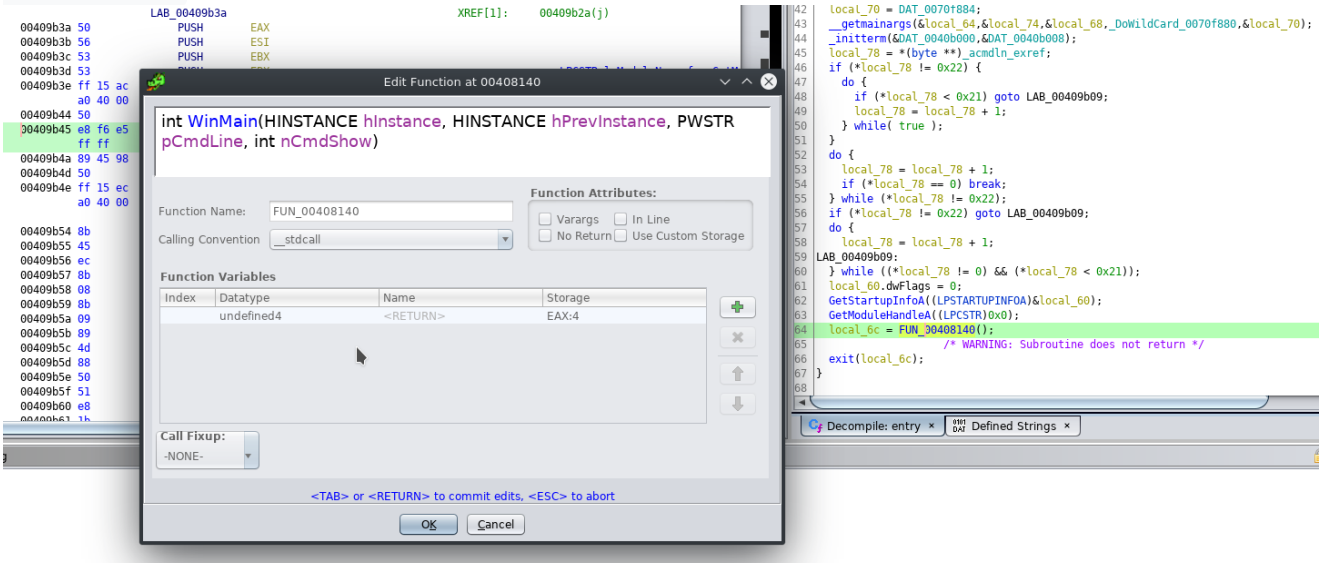
Wannacry Sample #2 sometimes referred to as "encryptor" available @ https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.WannaCry `sha256 ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa`

Wannacry Plus available @ https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.WannaCry_Plus `sha256 55504677f82981962d85495231695d3a92aa0b31ec35a957bd9cbbef618658e3` The first thing we're going to take a look at is the symbol tree. Stepping into the function called *entry*: we notice that it is in fact not the main / WinMain function, but rather a preparing function that will call WinMain at the end (this might acutally be an artifact of Ghidra's decompiler).

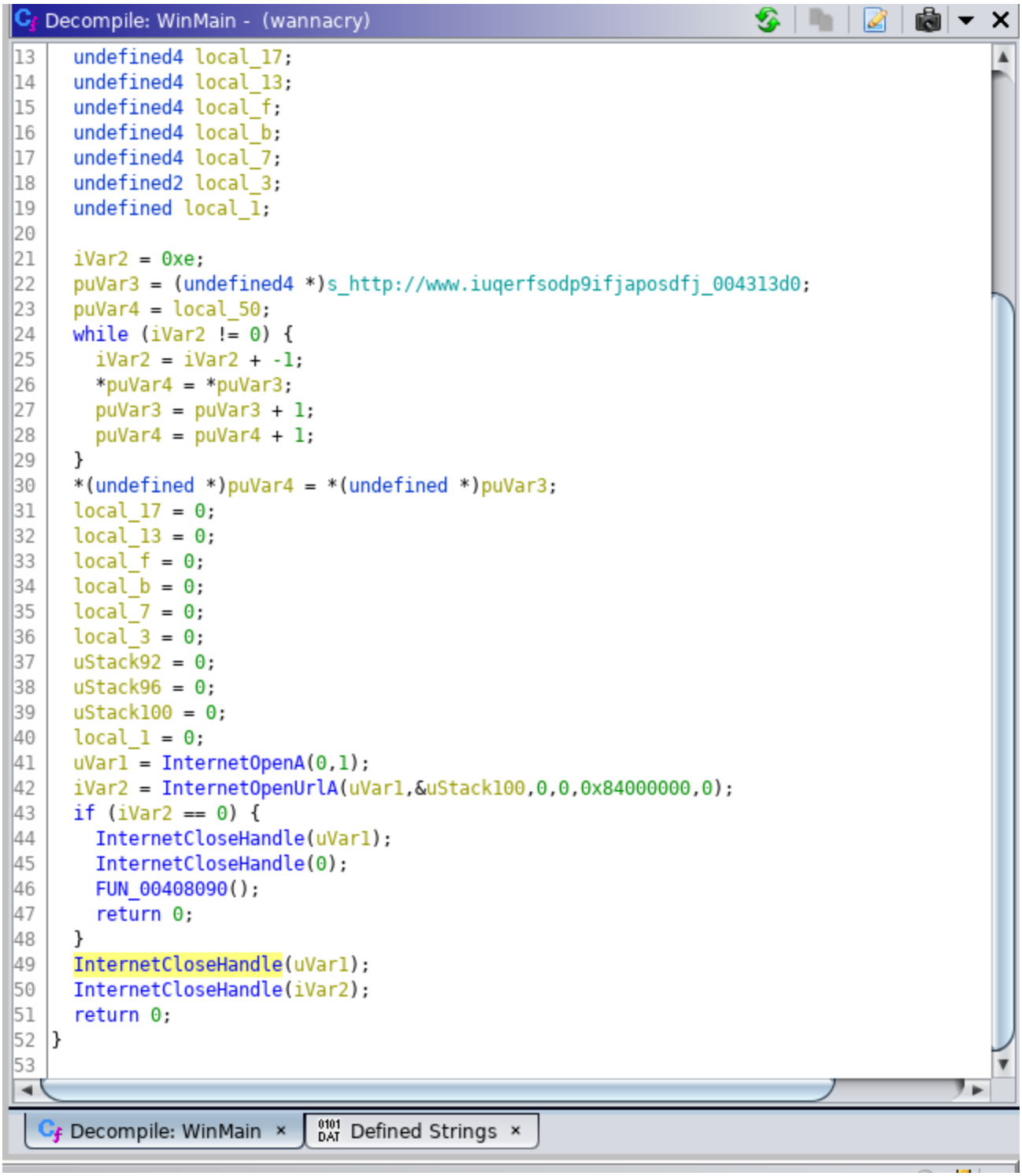


Because the decompilation result in our WinMain function is not that pretty yet we will edit its function signature to match the one described in the Win32 API Reference. `int WINAPI`

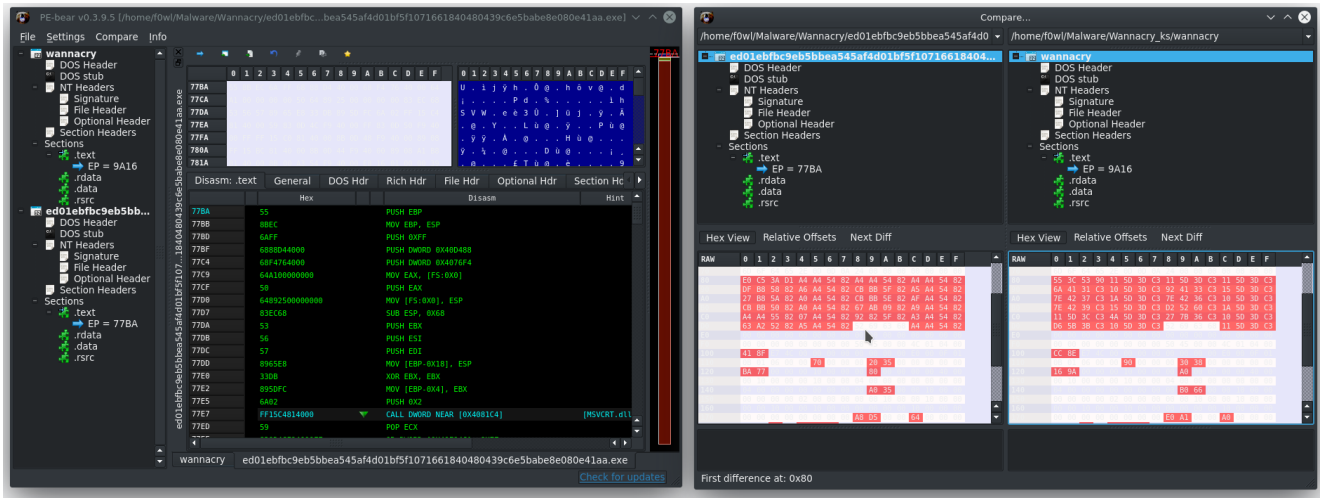
```
wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int
nCmdShow);
```



After this is done the decompilation result will be much better and easier on the eyes. One of the first things you will spot is the famous *Kill Switch URL* registered by Malwaretech after the inital outbreak which is to this day pointing to the Kryptos Logic sinkhole. After Line 41 you are also able to see multiple InternetOpen etc. function calls that will check if the aforementioned URL is registered and reachable. If that is the case it will close the connection socket and exit to WinMain before the encryption processs even started. Of course this also means that if the infected PC is not connected to the Interwebs (remember it propagates via SMB over local networks as well) or is unable to resolve the domain name the ransomware will go to town with the user's files. Looking into sample #2 there is acutally no such kill switch which means that it is one of the later versions following the inital outbreak.

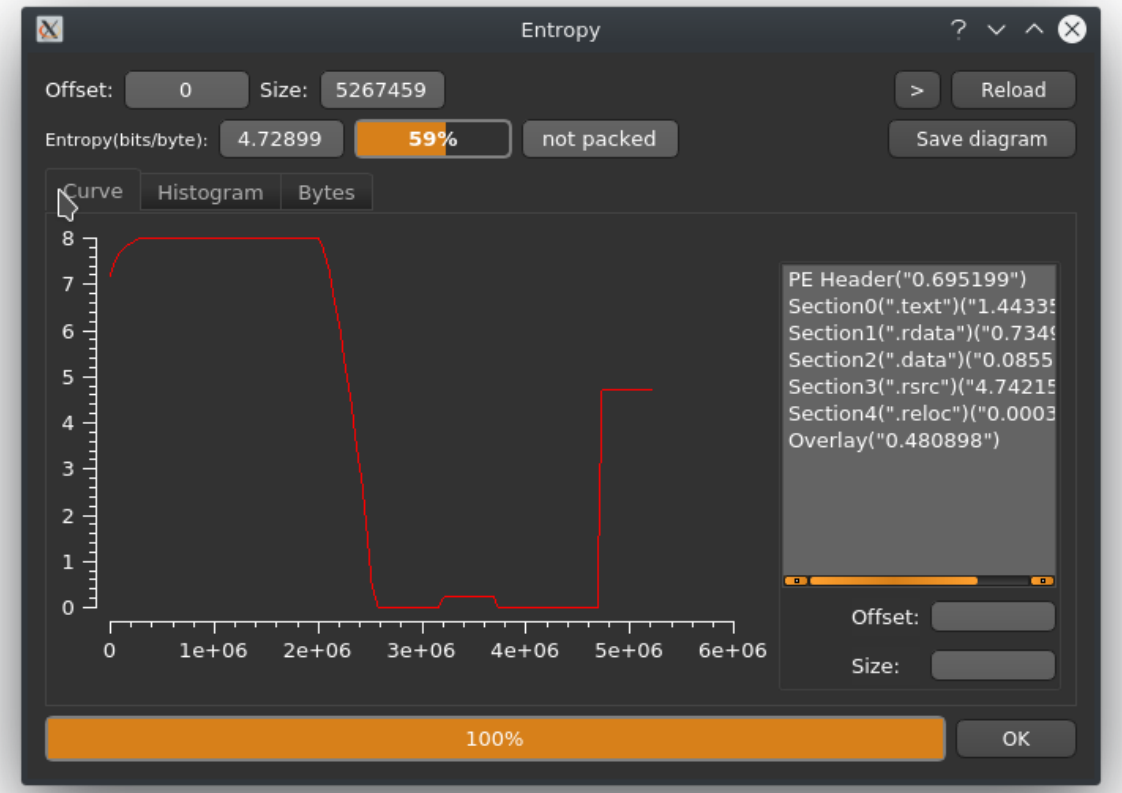


To show the differences between the kill switched first sample and the second rambo version I fired up hasherezade's awesome PE-Bear and loaded Sample #2 and #1. This indeed confirms that the samples are basically the same, but version #2 is missing the notorious kill switch.

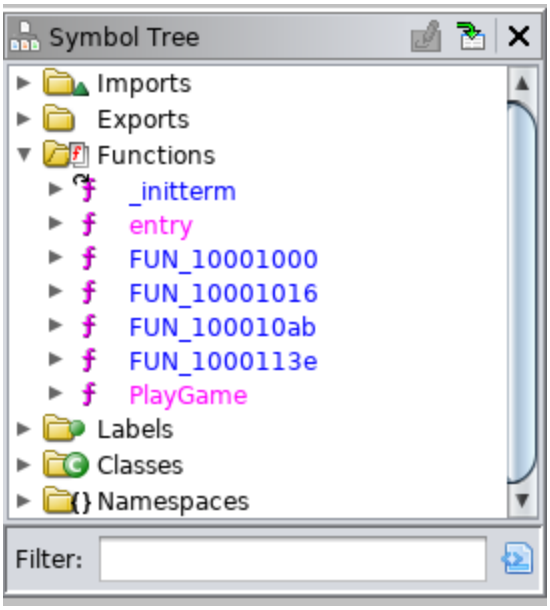


WannaCry Plus

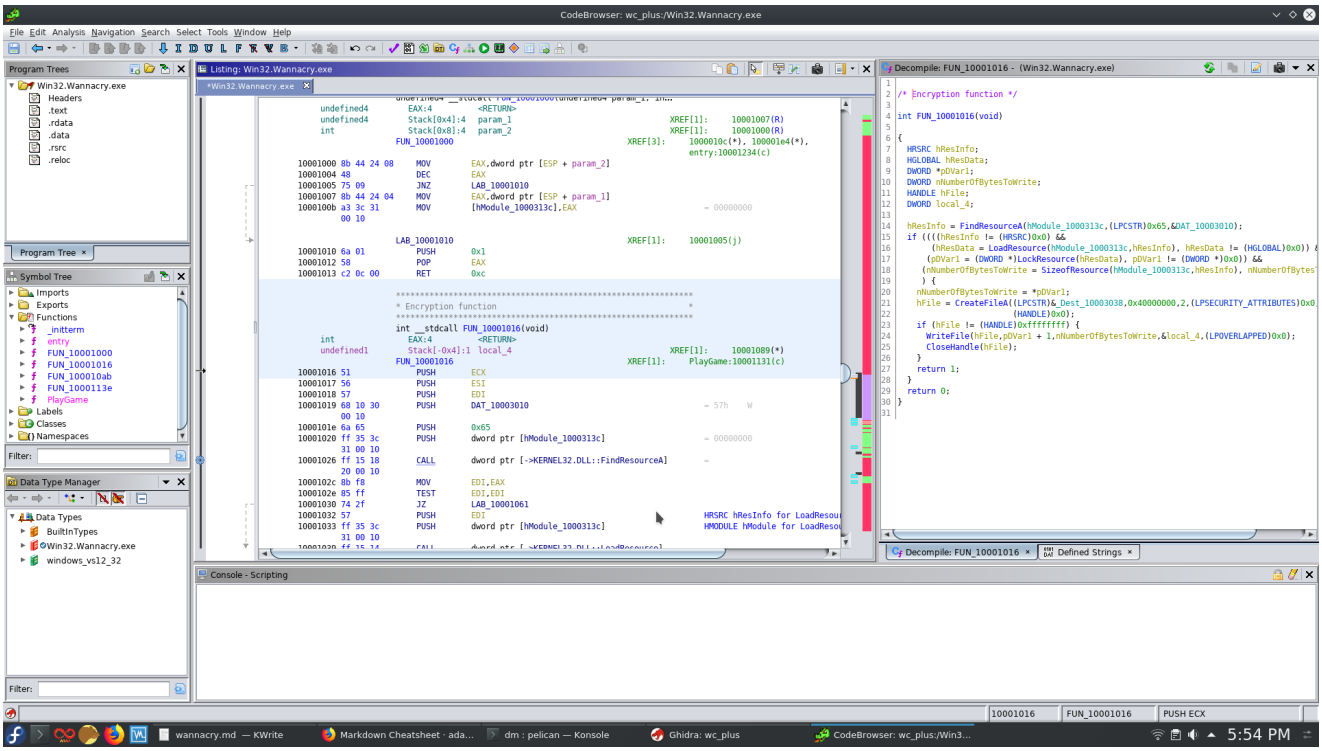
I haven't heard of this strain/ variant before, but it got it's own subfolder in ytisf's TheZoo so it has to be special in some way, right? Let's first check the entropy of the binary with "Detect it easy" to see if it is packed or obfuscated in any way:



Looking at the entropy graph we can pretty comfortably say that the PE is neither packed nor obfuscated (which would have been out of the ordinary for a WannaCry sample anyway). Looking at the symbol tree we are greated with a new function called PlayGame. Please no Fortnite ransomware kthxbye :D We'll have a look into that later..

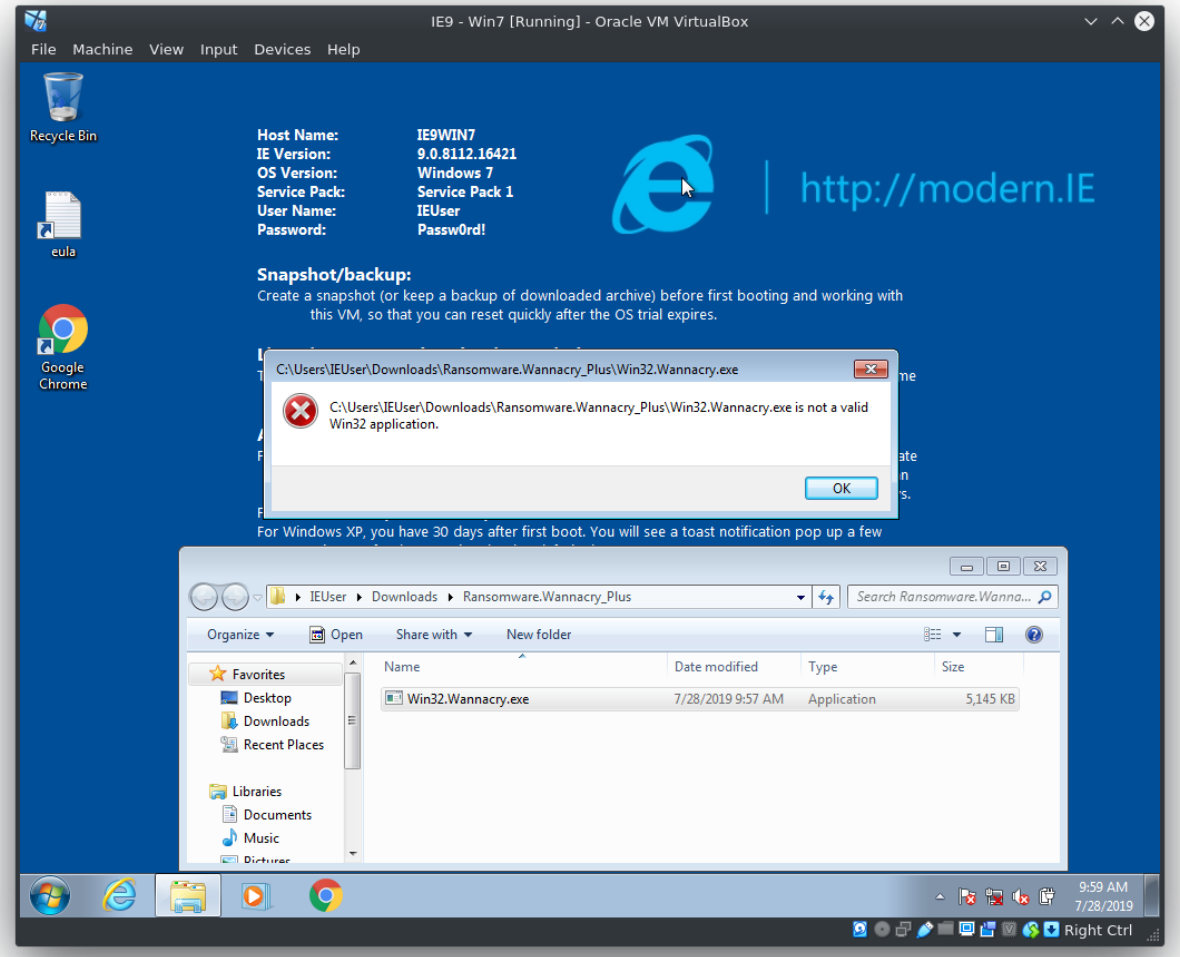


Jumping into the entry function things are looking quite different compared to first two samples. Following the procedure we are dropped into FUN_10001016 which is what i presume the file encryption function. This pretty easy to spot through the rather characteristic combination of *FindResourceA*, *CreateFileA* and *WriteFile*.

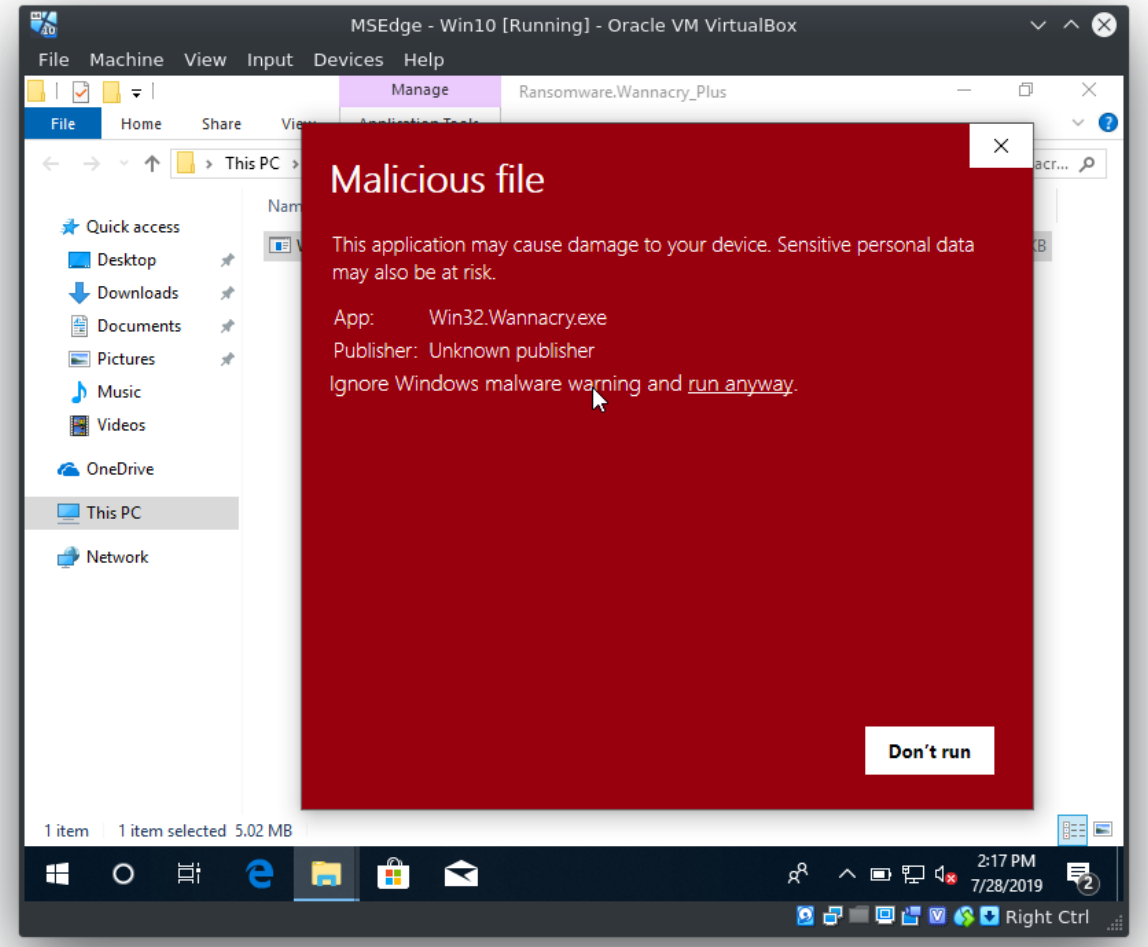


To see what happens if I run the malware I fired up a Windows 7 x86 VM in VirtualBox provided by https://modern.ie/. After seeing the error message below I thought the executable might actually be a x86_64 one since it refuses to run on the 32-bit Windows 7

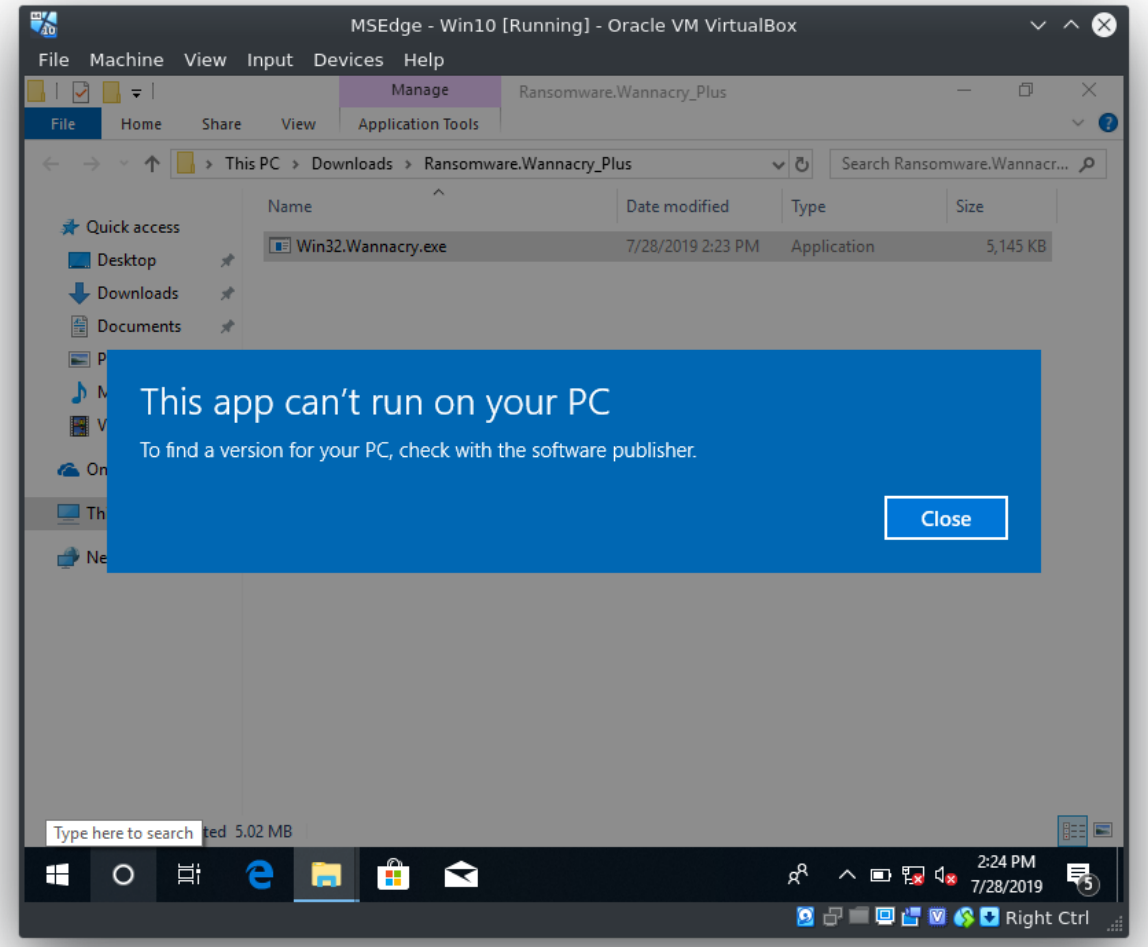
System. Even these days it is actually quite unusual for malware to be compiled for x64 systems only since it'll cut out a lot of the old and vulnerable systems running x86 XP for example (which is kind of a no-brainer since the potatohead holding PCs for ransom would want to maximise the attack surface and earnings).



Kudos to Microsoft in this case: Their Defender and SmartScreen really stepped up their game. For an attacker and (sadly) for a malware reverse engineer it is actually quite difficult to circumvent or disable the built in Mal-/Ransomware Protection. You are constantly greeted with Pop-Ups about a detected ransomware executable and the Defender will even go as far as simply deleting your precious sample :(



But even after calming down the Windows Defender I couldn't get the malware to encrypt anything :S



Looking at the Anyrun Sandbox Analysis over here we see the same error message but it seems to drop another executable called "SearchProtocolHost.exe" which is probably RunPE Process Hollowing at play. The next step will probably be manual debugging, so stay tuned!

IOCs

Wannacry (SHA256)

```
24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c
ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa
55504677f82981962d85495231695d3a92aa0b31ec35a957bd9cbbef618658e3
32f24601153be0885f11d62e0a8a2f0280a2034fc981d8184180c5d3b1b9e8cf
697158bcade7373ccc9e52ea1171d780988fc845d2b696898654e18954578920
ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa
```