2IC80 - Lab on offensive computer security
Malware Analysis Report

NotPetya

Roan Hofland (1237043)
Sverre van Mulken (1235073)

2020-04-13

1 Introduction

In this report, we analyze the NotPetya malware, which gets its name from being almost identical to the GoldenEye variant of the Petya ransomware, with some tweaks to make it unique. We will first explain our setup and talk a bit about acquiring the sample. Then we perform a static analysis where we tried to analyze as many components of the malware binary as we could within the given time frame. The files used in and resulting from the analysis, along with a much more detailed log[1] file keeping track of our findings and actions can be found in a GitHub repository at `https://github.com/RoanH/NotPetya`.

1.1 Setup

In order to play it safe, all of our analysis was performed on a system running the Ubuntu Linux 18.04.4 LTS operating system. The main reason for this was to prevent any activation of NotPetya since we knew in advance it only targets systems running Microsoft Windows.

For our analysis, we primarily made use of Ghidra[2] with the OOAnalyzer[3] plugin and command line programs like wrestool[4], dd[5], zlib-flate[6], file[7] and hexedit[8].

We also had to include some missing structs and types during our analysis that did not ship with Ghidra by default. For example nothing from the `ws2_32.dll` networking library was present.

1.2 Sample

We were able to get a sample of the NotPetya malware from a GitHub repository made by Fabrizio Monaco[9]. The SHA256 hash of this sample is `027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745`. Before we started, we submitted this sample to VirusTotal to verify that it was in fact NotPetya. Part of the VirusTotal report is shown in the figure below.

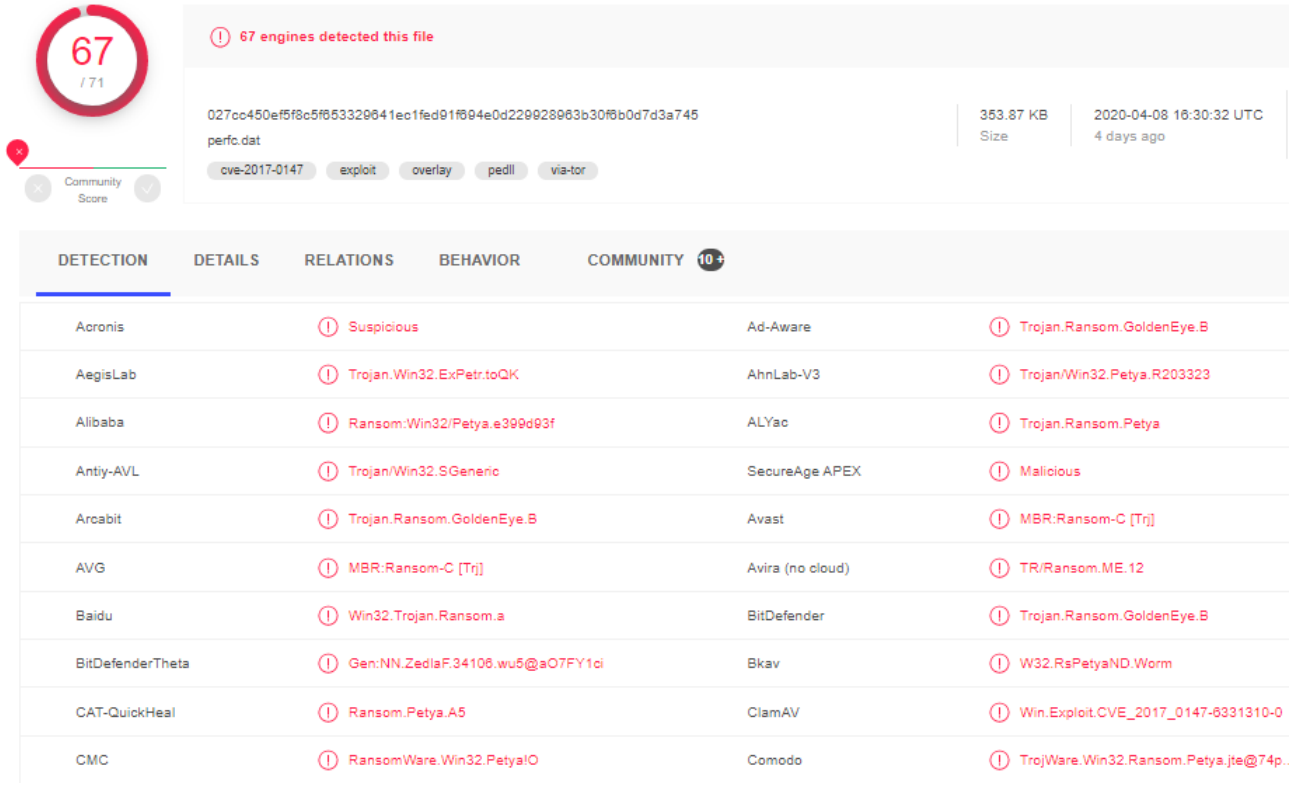


Figure 1: VirusTotal report for our NotPetya sample

[1] `https://github.com/RoanH/NotPetya/blob/master/Notes/log.md`
[2] `https://ghidra-sre.org/`
[3] `https://github.com/cmu-sei/pharos/tree/master/tools/ooanalyzer/ghidra/OOAnalyzerPlugin`
[4] `https://www.nongnu.org/icoutils/`
[5] `https://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html#dd-invocation`
[6] `http://manpages.ubuntu.com/manpages/trusty/man1/zlib-flate.1.html`
[7] `https://linux.die.net/man/1/file`
[8] `https://linux.die.net/man/1/hexedit`
[9] `https://github.com/fabrimagic72/malware-samples`

We see from these results that the sample from the GitHub does indeed get recognized as (a variant of) Petya. This means we can use the sample for our analysis. We also see that, according to VirusTotal as shown in Figure 1, out of the 71 programs that checked the file, 67 programs were able to detect it as a virus.

2 Static analysis

2.1 Entry point

When starting the static analysis we found that the entry point function did not resemble any typical main functions. We first had to fix the function signature in Ghidra before we found out that it was `DllMain`. What it ends up doing is storing a handle to the Dll module, this is less than expected.

2.2 Ordinal_1

As the entry point function turned out to not do very much, we investigated the other exported function labelled `Ordinal_1`. Through our static analysis we were later able to confirm that the `Ordinal_1` function is in fact the root function of the malware.

2.2.1 Malware setup

What we found is that the `Ordinal_1` function starts off by calling a function that performs general setup for the malware, in the end we renamed this function to `setup_privileges_antivirus_malware_copy` and it is shown below.

```
void setup_privileges_antivirus_malware_copy(void){
  BOOL success;
  DWORD result;
  uint privileges;

  if (_original_handle_freed == 0) {
    millis_since_system_start = GetTickCount();
    success = grant_privilege(L"SeShutdownPrivilege");
    privileges = (uint)(success != 0);
    success = grant_privilege(L"SeDebugPrivilege");
    if (success != 0) {
      privileges = privileges | 2;
    }
    success = grant_privilege(L"SeTcbPrivilege");
    if (success != 0) {
      privileges = privileges | 4;
    }
    granted_privileges = privileges;
    _detected_anti_virus = detect_anti_virus();
    result = GetModuleFileNameW(DLL_handle,&dll_fully_qualified_path,0x30c);
    if (result != 0) {
      copy_malware_dll_to_memory();
      return;
    }
  }
  return;
}
```

In this function we see several things happen. First the function finds out and stores the number of milliseconds since the system was started in a global variable we renamed `millis_since_system_start`. This global is later used to schedule a reboot of the system.

Then it tries to grant the current process 3 specific privileges and which of the privileges this succeeds for is stored in a global we renamed `granted_privileges`. Table 1 shows some details about these 3 privileges.

Next it calls the `detect_anti_virus` function which uses a snapshot of all running processes to compare currently running executable file names with hashes of executable file names belonging to several antivirus software, the specific hashes and

Privilege	Description	Tracker bit
SeShutdownPrivilege	Allows the user to shut down the system	1
SeDebugPrivilege	Allows the user to debug programs	2
SeTcbPrivilege	Allows the user to act as part of the operating system	3

Table 1: Privilege details

software can be found in Table 2. Which of these programs is found is then stored in the global variable `detected_anti_virus`, where the fourth bit is set to 0 if Kaspersky is detected and the third bit is set to 0 if Norton or Symantec is detected. Interestingly, whereas normally bits are set to 1 in a situation like this, here the global is set to all 1's and bits are set to 0. This global is used a lot and has a huge influence on the control flow of the program, several examples of which can be seen later on in the report.

Software	Hash	Executable	Tracker bit	Possible Global Field Values
Kaspersky	`0x2e214b44`	avp.exe	3	`0xFFFFFFF7` `0xFFFFFFF3`
Norton Security	`0x651b3005`	NS.exe	4	`0xFFFFFFFB` `0xFFFFFFF3`
Symantec	`0x6403527e`	ccSvcHst.exe	4	`0xFFFFFFFB` `0xFFFFFFF3`

Table 2: Details on antivirus software tracking

Lastly, it calls a function we renamed to `copy_malware_dll_to_memory`. Here it opens the malware DLL file and stores a copy of itself in memory.

After looking at `setup_privileges_antivirus_malware_copy`, we investigated a function we renamed to `possible_restart_from_in_memory_copy`, this function invokes `Ordinal_1` after first relaunching the malware from memory, which we saw being prepared by `copy_malware_dll_to_memory`, and executing some initialisation subroutines.

One of these subroutines, `delete_dll_and_invoke_Ordinal_1`, overwrites the malware DLL file, essentially wiping it's contents. It also removes the malware DLL file, making it harder to recover the contents using disk forensics. Take note here that it wipes and removes the original file and that a copy of this file was loaded in memory by `copy_malware_dll_to_memory` as previously discussed.

2.2.2 Command line argument handling

A function we renamed `handle_cmd_args` is responsible for handling any command line input passed to NotPetya. This function invokes a function renamed to `handle_h_flag` for command line arguments that start with `-h`. It also passes any arguments that contain `:` to a function we renamed `handle_colon_args`. We discovered later on that the arguments passed here are user credentials in the format `username:password`.

Both of these functions eventually make use of three functions we renamed `possible_lock_and_wait_check_args`, `possible_lock_and_wait` and `possible_lock`. We later figured out that these functions are used to pass information between different parts of the program, especially between different threads. These three functions also always take as an additional parameter one of three critical sections that are created right before `handle_cmd_args` is called. These critical sections are also all stored in a global variable.

2.2.3 Killswitch

Eventually we came across the following code fragment:

```
if ((granted_privileges & 2) != 0) {
    create_c_windows_file_or_exit();
    destroy_boot_and_write_custom_bootloader();
}
```

Here we see the functions `create_c_windows_file_or_exit` and `destroy_boot_and_write_custom_bootloader` being invoked, only when the process has been granted a specific privilege by the `setup_privileges_antivirus_malware_copy` function we discussed previously. In particular, this is the `SeDebugPrivilege` which allows NotPetya to modify any running process,

including programs running under the `SYSTEM` account.

The second function is discussed in the next section. However the first function, namely `create_c_windows_file_or_exit`, acts as a type of killswitch. It checks if the file `C:\Windows\dllname` exists, where `dllname` is the original name of the NotPetya binary, if such a file already exists, then NotPetya will exit. Otherwise it will create the file and continue execution.

2.2.4 Boot handling

When we take a look at `destroy_boot_and_write_custom_bootloader` we see that this function overwrites the second sector of the `C:` drive with some (garbage) data. Leaving the MBR (Master Boot Record) on the first sector intact. In a call to a function we renamed `write_custom_bootloader` it ends up writing the custom bootloader data to the disk, overwriting the existing bootloader. The data being written includes the ransomware boot screen text shown in Figure 2 among other things.

Finally it ends up continuing in one of two ways: if it previously detected the presence of Kaspersky using the `detect_anti_virus` function or if the process of writing the custom bootloader fails, it wipes the first 10 sectors of `PhysicalDrive0`, which generally stores the MBR and part of the initial program loader, making it impossible to boot the system from this drive.

Next we look at a function we renamed `schedule_reboot` which is used to schedule a reboot of the system at a random time up to an hour into the future, with a minimum of 13 minutes from the current time. Using a subroutine we renamed `running_win_8_or_higher`, it checks if the Windows version of the current system is below Windows 8 or not. On systems running Windows 8 or above, it checks if the `setup_privileges_antivirus_malware_copy` function was able to grant the `SeTcbPrivilege` privilege, which gives the ability to run tasks as any user. While it will always schedule a reboot of the system, if this privilege is granted it will specifically execute this task as the `SYSTEM` account.

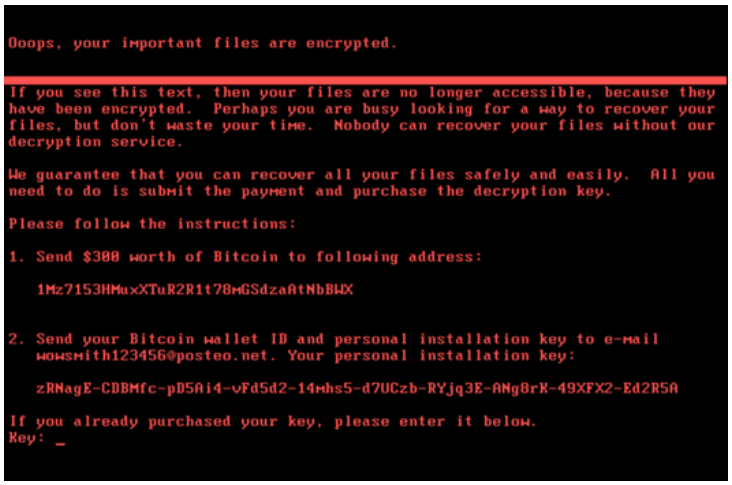


Figure 2: NotPetya custom boot text

2.2.5 Network Scanning

During our analysis we came across a thread that employs various methods to identify other hosts. The code coordinating this can be seen below.

```
void find_infection_candidates_on_network(void){
  bool executed;
  LPCRITICAL_SECTION critical_secton;
  BOOL success;
  WCHAR net_bios_name [260];
  DWORD buffer_size;

  critical_secton = critical_section_no_extra_debug;
  possible_lock_and_wait_check_args(critical_section_no_extra_debug,L"127.0.0.1",1);
  possible_lock_and_wait_check_args(critical_secton,L"localhost",1);
  buffer_size = 0x104;
  success = GetComputerNameExW(ComputerNamePhysicalNetBIOS,net_bios_name,&buffer_size);
  if (success != 0) {
    possible_lock_and_wait_check_args(critical_secton,net_bios_name,1);
  }
  CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,find_infection_candidates,critical_secton,0,(LPDWORD)0x0);
  executed = false;
  do {
    find_remote_infection_candidates(critical_secton);
    find_infection_candidates_arp(critical_secton);
    if (!executed) {
      find_infection_candidates_via_domain(critical_secton,0x80000000,0);
      executed = true;
```

```
    }
    Sleep(180000);
  } while( true );
}
```

From this function we found that there are 5 main scans being performed

1. **Localhost:** We see that the function passes `localhost`, `127.0.0.1` and the `NetBIOS Name` for the local computer to `possible_lock_and_wait_check_args`.
2. **Adapter:** Using the function we renamed to `find_infection_candidates` we see all the network adapters in the system being retrieved. If the current host is a DHCP server for one of the adapters, we see all of its connected clients being checked for the SMB ports being open. We also always see hosts within the subnet being scanned for open SMB ports. These hosts are then passed to `possible_lock_and_wait_check_args`.
3. **Remote connections:** A function we renamed `find_remote_infection_candidates` gets a list of all connections in the TCP table and passes the remote addresses to `possible_lock_and_wait_check_args`.
4. **ARP Table:** The ARP table is retrieved by a function we renamed `find_infection_candidates_arp` which proceeds to pass all IP addresses in it to `possible_lock_and_wait_check_args`.
5. **Domain:** A function we renamed `find_infection_candidates_via_domain` is invoked which recursively traverses servers of the type `SV_TYPE_DOMAIN_ENUM` and passes all servers of the type `SV_TYPE_WORKSTATION` and `SV_TYPE_SERVER` that are running Windows 2000 or later to `possible_lock_and_wait_check_args`.

Scan type 3, 4 and 5 are performed every 2 minutes, however 5 is only allowed to execute once in total.

2.2.6 Resource Extraction

Several resources are extracted from the malware DLL by functions we renamed to `extract_and_run_resource_1_or_2` and `extract_resource_3`. Using wrestool we were able to find out all the resources that were present in the file, the result is shown in Table 3. Resource 4 is used to exploit EternalBlue and not discussed here.

Type	Name	Language	Offset	Size
10 (rcdata)	1	1033	0x200e8	24958
10 (rcdata)	2	1033	0x26268	27426
10 (rcdata)	3	1033	0x2cd8c	191605
10 (rcdata)	4	1033	0x5ba04	3379

Table 3: Resources found in DLL file

We first discuss `extract_and_run_resource_1_or_2`. This function continues with resource 1 if the host system is a 32bit system and with resource 2 if it is a 64bit system.

In the function the resource was passed to we were able to find both a `GZIP_MAGIC` constant and a bunch of error messages. Using these we were able to identify several functions from the inflate implementation in zlib[10], a software library used for data compression. This meant that the resource passed to this function is in fact compressed.

The inflated resource data is then written to a temporary file in the `%TEMP%` directory and executed as a new process. A named pipe is used to communicate with the process and any data received on the pipe containing a `:` is passed on to the `handle_colon_arg` function we saw before. After 60 seconds, the process is terminated and the file it was written to is deleted.

The `extract_resource_3` function simply extracts resource 3 and writes it to disk. If `setup_privileges_antivirus_malware_copy` was able to grant either the `SeDebugPrivilege` or the `SeTcbPrivilege`, the resource is written to `C:\Windows\dllhost.dat`. If neither of the privileges were granted, then it writes the resource to `%APPDATA%\dllhost.dat`.

2.2.7 Admin share

NotPetya comes with a mechanism to spread itself to new targets using the admin share. The admin share is present on all Windows systems at the `\\hostname\admin$` address and points to the `C:\Windows` folder. Normally it is used to perform system updates via the network. NotPetya takes advantage of this share to spread itself, this being a 4 step process.

[10] `https://www.zlib.net/`

1. A function we renamed `find_victims_via_shares` recursively parses the tree of network resources and retrieves the remote name of each resource in the network. The remote names are then passed to `possible_lock_and_wait_check_args`.
2. In a function we renamed `obtain_user_credentials`, `CredEnumerateW` is used to get all of the credentials in the user's credential set. Credentials of type `CRED_TYPE_DOMAIN_PASSWORD` are then passed to `possible_lock_and_wait_check_args` while all other credentials with the exception of credentials of type `CRED_TYPE_GENERIC` are passed to `handle_colon_args`.
3. A function we renamed `infect_hosts_via_admin_share` connects to the admin share of a victim using the obtained credentials. It is then checked if a file exists at `\\hostname\admin$\malwaredll` and if this is the case the operation is aborted. This again demonstrates how creating a file in this location with the same name as the malware file can prevent infection with NotPetya. If no file exists at this location then the malware copies itself over.
4. Finally the copy of NotPetya is started using one of two possible commands, one of them using resource 3 which was extracted earlier. Both commands also get all credentials gathered so far appended to them.

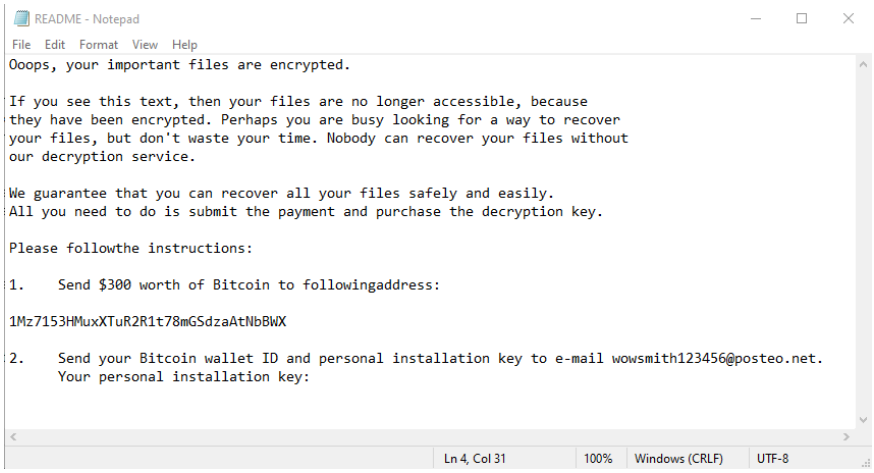
   - `%dllhost \\%host -accepteula -s -d:\Windows\System32\rundll32.exe "C:\Windows\%dll",#1`
   - `C:\Windows\wbem\wmic.exe /node:"%host" /user:"%username" /password:"%password" process call create "C:\Windows\System32\rundll32.exe\"C:\Windows\%dll\" #1`

   The command used depends on the privileges obtained so far. This also clearly revealed that `Ordinal_1` is the function invoked to start NotPetya.

2.2.8 Encrypting Drives

The encryption of the drives, done by executing the `encrypt_all_drives` function, works as follows. All currently available disk drives are retrieved and a dedicated thread is started for each of the drives. Each thread generates its own encryption key and goes through each file on the drive encrypting it, avoiding the `C:\Windows\` directory and any files with an extension not in Table 3b. It then creates a new file called `README.TXT` at the root of the drive, which contains a message similar to the boot message we saw earlier and a personal installation key. The contents of the message are shown in Figure 3a.

The personal installation key is the 128bit AES key that was used to encrypt all the files, encrypted with the 2048bit RSA public key of the attacker. Note that because the file in the image is a reconstruction of the file found through our analysis, no actual installation key is present.



(a)

.3ds	.7z	.accdb	.ai	.asp
.aspx	.avhd	.back	.bak	.c
.cfg	.conf	.cpp	.cs	.ctl
.dbf	.disk	.djvu	.doc	.docx
.dwg	.eml	.fdb	.gz	.h
.hdd	.kdbx	.mail	.mdb	.msg
.nrg	.ora	.ost	.ova	.ovf
.pdf	.php	.pmf	.ppt	.pptx
.pst	.pvi	.py	.pyc	.rar
.rtf	.sln	.sql	.tar	.vbox
.vbs	.vcb	.vdi	.vfd	.vmc
.vmdk	.vmsd	.vmx	.vsdx	.vsv
.work	.xls	.xlsx	.xvd	.zip

(b)

Figure 3: The README.txt file stored at the root of each drive (a) and file extensions targeted for encryption (b).

2.2.9 EternalBlue

Besides spreading via the admin share, NotPetya also attempts to spread itself using the infamous exploit EternalBlue. Before attempting to do this some data like credentials and command line arguments are prepared. Most interesting however is that `detected_anti_virus` is checked, if Kaspersky turns out to be detected, we see that the EternalBlue functionality is not executed.

Inside the main function we see a couple of things. We see it sending a large number of SMB messages of several types, 12 of which we were able to identify. We also see that the payload for these SMB messages is often decoded by XOR'ing them

with some value, which makes the payload harder to identify when analyzing the global data fields. The main function uses a subroutine we renamed `eternal_blue_load_payload` to load and extract the fourth resource contained inside the malware. This resource is decoded by XOR'ing it with `0x86868686`. Finally a function we renamed `eternal_send_main_payload` is invoked to send the main payload using several `SMB_COM_TRANSACTION2` messages.

2.2.10 Final Cleanup

At the end of `Ordinal_1` we see the following command being created and executed:

```
wevtutil cl Setup & wevtutil cl System & wevtutil cl Security & wevtutil cl Application
& fsutil usn deletejournal /D %dll:
```

This clears all of the event logs as well as all of the disk events related to NotPetya. After this, various calls are issued to forcefully reboot the system.

3 Conclusion

This report presents the analysis of a sample of the NotPetya malware. The functionality of NotPetya can be summarized as follows. First it actively tries to gain more privileges on the host system and constantly scans its surroundings for other hosts. Given enough privileges it then tries to spread using the admin share and by exploiting EternalBlue. Finally, it also encrypts all the drives and writes a custom bootloader to the primary disk which shows the infamous ransomware demand after a reboot.

In the end we were able to analyze a large part of the NotPetya sample. In Table 4 we show the number of functions and instructions we covered in each of the binaries. Something interesting to note is that the code for the EternalBlue exploit that is included, entails around 40% of all the instructions in the entire Main DLL.

Binary	**Functions**				**Instructions**			
	Total	Covered	Missed	%	Total	Covered	Missed	%
Main DLL	178	166	12	93,26	47847	46061	1786	96,27
Resource 1	32bit version of resource 2							
Resource 2	151	92	59	60,93	28995	17566	11429	60,58
Resource 3	Remote code execution program							
Resource 4	SMB Payload							

Table 4: Analysis Statistics

While we unfortunately, due to time constraints, were not able to look deeper into all of the resources included or the custom bootloader, we were still able to figure out most, if not all, of the main functionality of the malware. Including the encryption mechanism, the way it spreads to other hosts and the initial setup before the inevitable reboot.

We also know roughly what the embedded resources were used for, namely.

- **Resource 1 & 2**: Some executable that obtains user credentials which are then sent back to the main process. The only reason we covered a fair number of instructions in this binary is due to Ghidra recognizing a lot of functions. We also only extracted, restored the headers and decompressed resource 2.
- **Resource 3**: Some executable that allows execution of commands on a remote system.
- **Resource 4**: The largest SMB payload used to exploit EternalBlue.