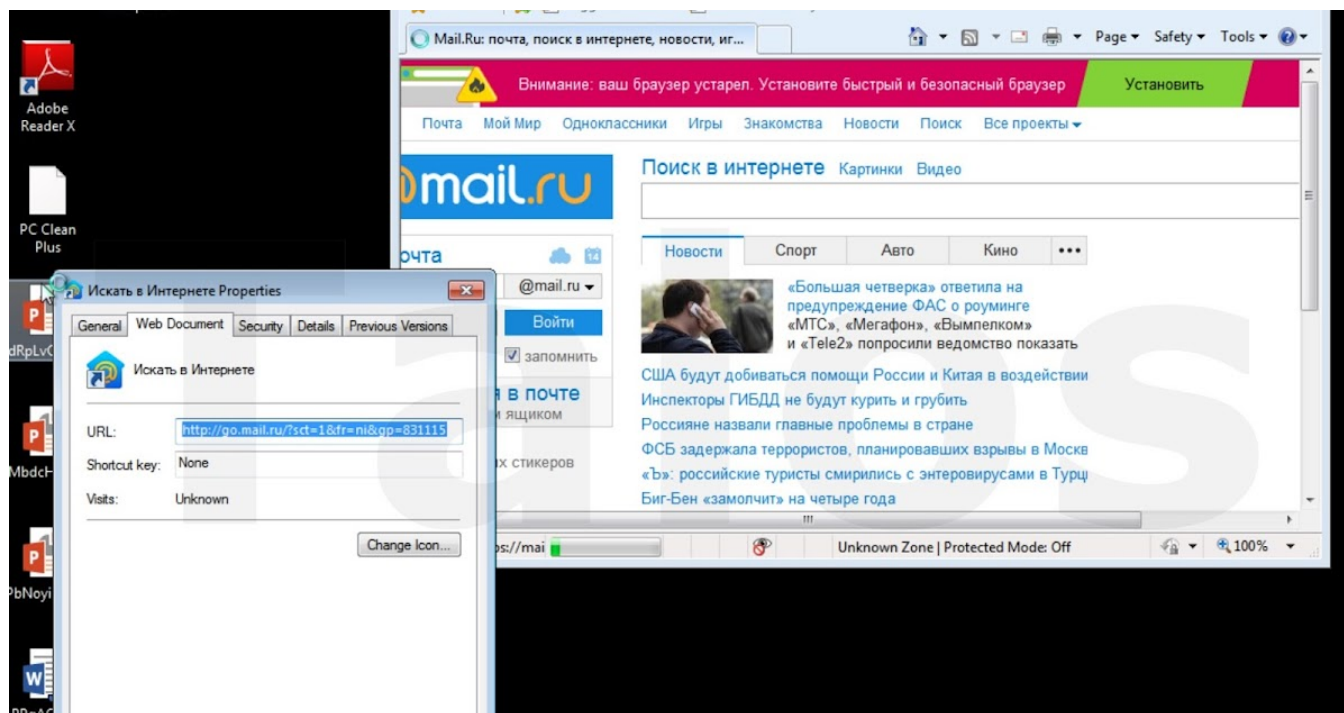


Graftor - But I Never Asked for This...

blog.talosintelligence.com/2017/09/grافتor-but-i-never-asked-for-this.html



This post is authored by [Holger Unterbrink](#) and [Matthew Molyett](#)

Overview

Free software often downloaded from large freeware distribution sites is a boon for the internet, providing users with functionality that otherwise they would not be able to use. Often users, happy that they are getting something free, fail to pay attention to the hints in the licence agreement that they are receiving additional software services bundled with the freeware they desire.

Graftor aka LoadMoney adware dropper is a potentially unwanted program often installed as part of freeware software installers. We wanted to investigate the effects this software has on a user's system. According to the analysis performed in our sandbox, Graftor and the associated affiliate files it downloads perform the following functions:

- Hijacks the user's browser and injects advertising banners
- Installs other potentially unwanted applications from partners like mail.ru
- It does not ask the user, it just silently installs these programs
- Random web page text is turned into links
- Adds Desktop and Browser Quick Launch links
- User's homepage is changed
- User's search provider is changed
- Partner adware is executed and it social engineers the user to install further software
- Checks for installed AV software
- Checks for sandbox environments
- Anti-Analysis protection
- Unnecessary API calls to overflow sandbox environments
- Creates/Modifies system certificates

Functionality

One of the first actions of the software is to install additional software on the user's desktop, and change browser settings to point to third party websites (Fig.1):

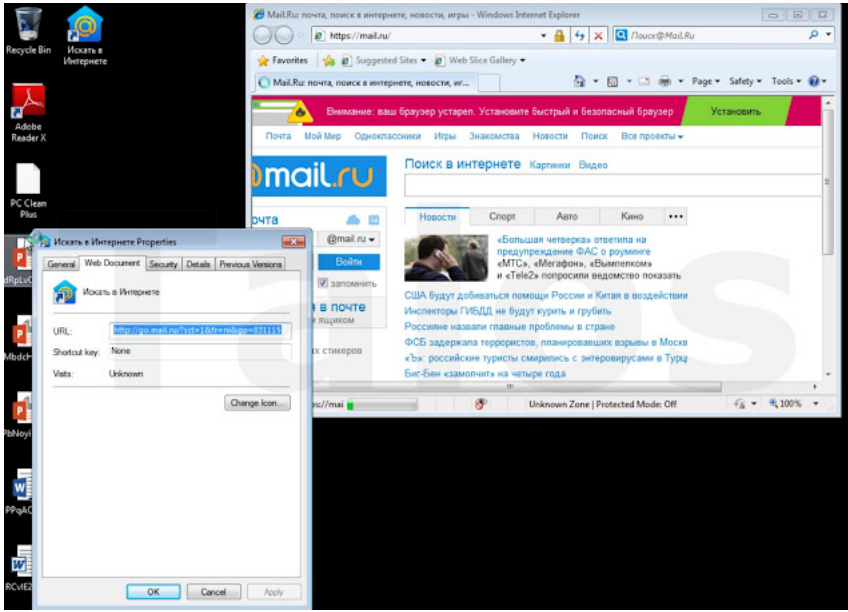


Fig. 1

Looking at the Cisco Umbrella DNS data for the CnC domain used in this campaign, we can see that the campaign only lasted for a couple of days (Fig. 2a), but affected a significant number of people. Fig. 2b and 2c show domains of two of the affiliate applications which Graftor installed during our sandbox run. It is very likely that this includes users who didn't intend to install these additional applications.

Regularfood[.]gdn (Command and Control Server Domain)

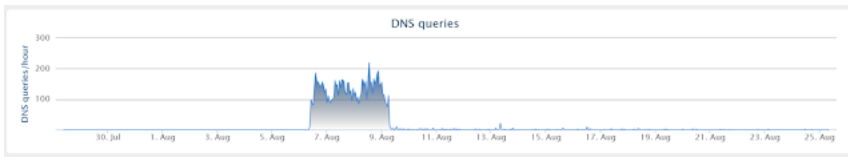


Fig. 2a

Affiliates (programs installed by Graftor):

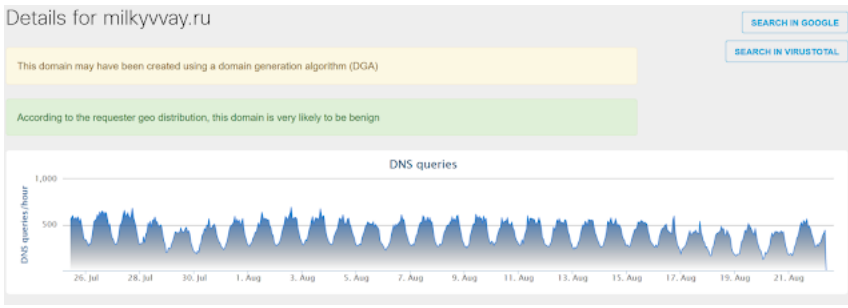


Fig. 2b

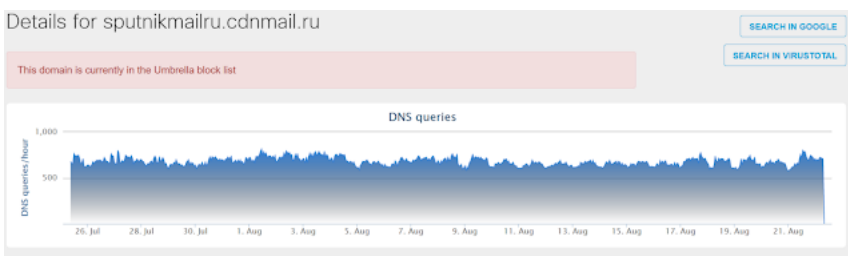


Fig. 3b

Technical Details

A few minutes after executing the original Grafter dropper (2263387661.exe), the software downloaded and installed a series of additional executables. This results in the process tree looking like this (Fig.3):

Fig. 3

We analysed the Grafter dropper/downloader (2263387661.exe). It comes with multiple stages of obfuscation. The first unpacking stage of the executable uses a heavily obfuscated but fairly simple unpacking algorithm which we will describe in the following section. This algorithm is obfuscated in the *WinMain* function distributed over several sub functions. Fig.4 shows you the complexity of the *WinMain* function in IDA, many of these building blocks are combined with further sub functions, jumping back and forth, which makes analysis particularly challenging.

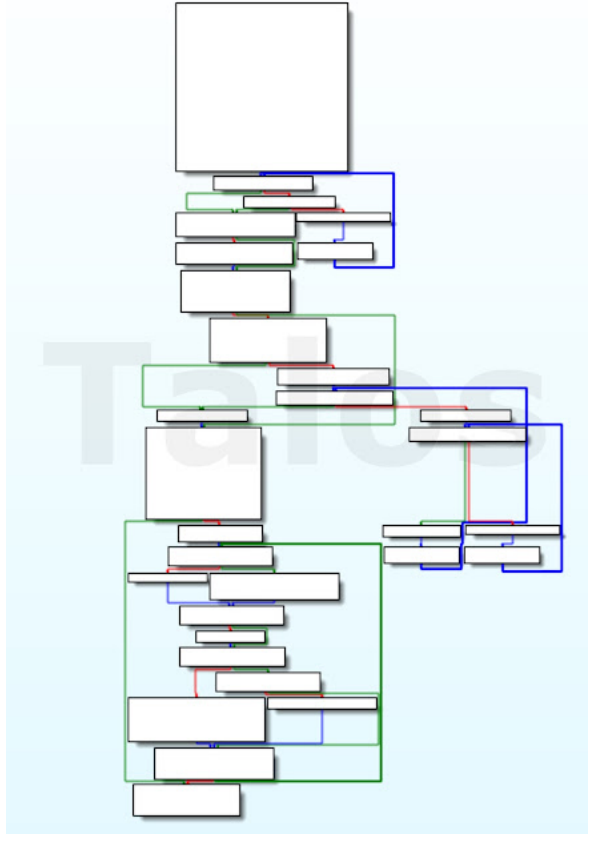


Fig. 4

First, a new buffer is allocated (see Fig.5 at 00401395) :

```

00401358 FF 15 34 20 40 00 call ds:ReleaseSemaphore ; Indirect Call Near Procedure
0040135E E8 9D FD FF FF call sub_401100 ; Call Procedure
00401363 E8 3B FD FF FF call DoNonsenseClearEAX ; Call Procedure
00401368 68 18 E6 41 00 push offset ProcName ; "VirtualAllocEx"
0040136D 68 28 E6 41 00 push offset LibFileName ; "kernel32.dll"
00401372 FF 15 30 20 40 00 call ds:LoadLibraryW ; Indirect Call Near Procedure
00401378 50 push eax ; hModule
00401379 FF 15 2C 20 40 00 call ds:GetProcAddress ; Indirect Call Near Procedure
0040137F 8B 5D CC mov ebx, [ebp+var_34]
00401382 6A 40 push 40h
00401384 8B 45 E4 mov eax, [ebp+var_1C]
00401387 50 push eax
00401388 FF 75 C0 push [ebp+var_40]
0040138B 33 C0 xor eax, eax ; Logical Exclusive OR
0040138D 50 push eax
0040138E B8 FF FF FF FF mov eax, 0FFFFFFh
00401393 50 push eax
00401394 58 pop eax
00401395 FF D3 call ebx ; VirtualAlloc
00401397 A3 70 E6 41 00 mov AllocBuffer, eax

```

Fig. 5

Then the bytes from 00416B6A (see Fig. 9 below) are decoded by different sub functions within the *WinMain* function. For example see *loc_4013EC* in Fig.6.

The code avoids calling functions by address values, but instead calls them via the values stored in registers or variables. For example the *call ebx* instruction in Fig. 5 at 00401395 results in a *VirtualAlloc* call. This makes the static analysis of the code harder. E.g without deeper analysis it is difficult to identify the destination of the *call* at 00401395 shown in Fig. 5.

```

004013EC
004013EC loc_4013EC:
004013EC C7 45 AC 00 00 00+mov [ebp+var_54], 0
004013F3 8B 5D F8 mov ebx, [ebp+encoded_byte2]
004013F6 8B 1D 70 E6 41 00 mov ebx, AllocBuffer
004013FC 8B 55 F8 mov edx, [ebp+encoded_byte2]
004013FF 03 55 E8 add edx, [ebp+Target] ; Target=Target+1 (per round) 0,1,2,3,4,...
00401402 8A 02 mov al, [edx]
00401404 8B 45 DC mov [ebp+encoded_byte], al

```

Fig. 6

Finally the decoded bytes are handed over to a function (Fig. 7 *write_unpkd_bytes2buf*), which writes these bytes into a buffer. This is the buffer which was allocated in Fig.5 at 00401395. The decoding loop starts again until all bytes are decoded:

```

00401446 08 00 push 0
0040144E 68 68 E6 41 00 push offset AllocBuffer2
00401453 6A 00 push 0
00401455 6A 03 push 3
00401457 E8 A4 FB FF FF call write_unpkd_bytes2buf ; decode byte (add 0x4e) and write to buffer
0040145C 83 C4 18 add esp, 18h ; Add

```

Fig. 7

Fig. 8 shows the *write_unpkd_bytes2buf* function itself:

```

00401000 55 push ebp
00401001 8B EC mov ebp, esp
00401003 83 EC 02 sub esp, 02h ; Integer Subtraction
00401005 C7 45 F4 00 00 00+mov [ebp+var_C], 0
0040100D C6 45 F8 00 mov [ebp+var_8], 0
00401011 8B 45 18 mov eax, [ebp+arg_encoded_byte] ; 42
00401014 8A 00 mov cl, [eax]
00401016 8B 4D FC mov byte ptr [ebp+var_4], cl
00401019 8D 55 FC mov edx, [ebp+var_4]
0040101C 81 E2 FF 00 00 00 and edx, 0FFh ; Logical AND
00401022 8B 45 1C mov eax, [ebp+arg_14]
00401025 8D 4C 02 4E lea ecx, [edx+eax+4Eh] ; (edx = encoded_byte = A2) * (arg14 = 0) * 4E = 0x90 = decoded byte
00401029 8B 55 10 mov edx, [ebp+arg_allocBuffer]
0040102C 8B 02 mov eax, [edx] ; AllocBuffer Addr
0040102E 8B 55 14 mov edx, [ebp+arg_C] ; 0
00401031 8B 0C 10 mov [eax+edx], cl ; write unpacked code byte
00401034 68 10 E6 41 00 push offset FileName ; "hex"
00401039 FF 15 50 20 40 00 call ds:DeleteFileW ; Indirect Call Near Procedure
0040103F 32 C0 xor al, al ; Logical Exclusive OR
00401041 8B E5 mov esp, ebp
00401043 5D pop ebp
00401044 C3 retn ; Return Near From Procedure
00401044 write_unpkd_bytes2buf endp

```

Fig. 8

The end result is that despite all of the complexity and obfuscation, the unpacking algorithm is remarkably simple and translates to the following pseudo-code (see Fig. 9 comments):

```

.data:00410000 uu 0
.data:00410001 db 42h ; B ; var x = 0x4e
.data:00410002 ; while(1):
.data:00410003 ; byte * x = decoded_byte #e.g. 0x90
.data:00410004 ; x = x + 1
.data:00410005 db 41h ; 0
.data:00410006 db 40h ; 0
.data:00410007 db 3Fh ; ?
.data:00410008 db 3Eh ; >
.data:00410009 db 3Dh ; =
.data:0041000A db 3Ch ; <
.data:0041000B db 3Bh ; ;
.data:0041000C db 3Ah ; ;
.data:0041000D db 39h ; 9
.data:0041000E db 38h ; 8
.data:0041000F db 37h ; 7
.data:00410010 db 0Fh ; 0
.data:00410011 db 30h ; 0
.data:00410012 db 90h ; E
.data:00410013 db 2Ah ; <

```

Fig. 9

This **first stage** of unpacking extracts the code into memory. After successfully unpacking this code it is executed via `call ecx` (see Fig. 10) - the **second stage** of the unpacker:

Fig. 10

This second stage code is position independent. It is loaded into a random address space picked by the operating system. The *VirtualAlloc* function in Fig.5 which we have mentioned above, is called with *LPVOID lpAddress* set to *NULL*, which means that the system determines where to allocate the memory region. This second stage is even more obfuscated by spaghetti code than the first stage. It's main task is to rebuild the *Import Address Table (IAT)* and resolve the addresses of certain library functions (Fig. 11), plus modify the original PE file.

```

00230009 90 nop
0023000A 90 nop
0023000B 90 nop
0023000C 55 push ebp
0023000D 8B EC mov ebp, esp
0023000E 81 EC E8 01 00 00 sub esp, 1C8
0023000F C7 85 MC FF FF 00 00 00 mov dword ptr ss:[ebp-154], 0
00230010 C7 85 08 FF FF 00 00 00 mov dword ptr ss:[ebp-94], 0
00230011 C7 85 34 FF FF C0 20 00 00 mov dword ptr ss:[ebp-C4], 20C0
00230012 C7 45 E4 20 12 00 00 mov dword ptr ss:[ebp-1C], 1220
00230013 C6 85 78 FE FF 56 mov byte ptr ss:[ebp-188], 56
00230014 C6 85 79 FE FF 69 mov byte ptr ss:[ebp-187], 69
00230015 C6 85 7A FE FF 72 mov byte ptr ss:[ebp-186], 72
00230016 C6 85 7B FE FF 74 mov byte ptr ss:[ebp-185], 74
00230017 C6 85 7C FE FF 75 mov byte ptr ss:[ebp-184], 75
00230018 C6 85 7D FE FF 61 mov byte ptr ss:[ebp-183], 61
00230019 C6 85 7E FE FF 6C mov byte ptr ss:[ebp-182], 6C
0023001A C6 85 7F FE FF 50 mov byte ptr ss:[ebp-181], 50
0023001B C6 85 80 FE FF 72 mov byte ptr ss:[ebp-180], 72
0023001C C6 85 81 FE FF 6F mov byte ptr ss:[ebp-179], 6F
0023001D C6 85 82 FE FF 74 mov byte ptr ss:[ebp-178], 74
0023001E C6 85 83 FE FF 65 mov byte ptr ss:[ebp-177], 65
0023001F C6 85 84 FE FF 63 mov byte ptr ss:[ebp-176], 63
00230020 C6 85 85 FE FF 74 mov byte ptr ss:[ebp-175], 74
00230021 C6 85 86 FE FF 00 mov byte ptr ss:[ebp-174], 0
00230022 C7 85 70 FF FF 00 00 00 mov dword ptr ss:[ebp-90], 0
00230023 C6 45 84 53 mov byte ptr ss:[ebp-4C], 53
00230024 C6 45 85 43 mov byte ptr ss:[ebp-4B], 43
00230025 C6 45 86 61 mov byte ptr ss:[ebp-4A], 61
00230026 C6 45 87 72 mov byte ptr ss:[ebp-49], 72
00230027 C6 45 88 64 mov byte ptr ss:[ebp-48], 64
00230028 C6 45 89 49 mov byte ptr ss:[ebp-47], 49
00230029 C6 45 8A 6E mov byte ptr ss:[ebp-46], 6E
0023002A C6 45 8B 74 mov byte ptr ss:[ebp-45], 74
0023002B C6 45 8C 72 mov byte ptr ss:[ebp-44], 72
0023002C C6 45 8D 6F mov byte ptr ss:[ebp-43], 6F
0023002D C6 45 8E 64 mov byte ptr ss:[ebp-42], 64
0023002E C6 45 8F 75 mov byte ptr ss:[ebp-41], 75
0023002F C6 45 C0 63 mov byte ptr ss:[ebp-40], 63
00230030 C6 45 C1 65 mov byte ptr ss:[ebp-3F], 65
00230031 C6 45 C2 43 mov byte ptr ss:[ebp-3E], 43
00230032 C6 45 C3 61 mov byte ptr ss:[ebp-3D], 61

```

Fig. 11

It stores the function addresses in different local variables. These are passed as arguments to several setup functions, for example: change memory region 0x400000 - 0x59C000 to read/write/execute (see Fig. 12). In other words, change the whole `.text`, `.rdata`, `.data`, and `.rsrc` section of the original PE file to read/write/execute. This enables the dropper to modify and execute the code stored in these regions. As we have already seen, in order to frustrate static analysis, most calls are obfuscated by either calling registers or variables (Fig.12).

```

002A1248 52 push edx
002A1249 6A 40 push 40
002A124E 8B 45 D8 mov eax, dword ptr ss:[ebp-28]
002A124C 8B 48 50 mov ecx, dword ptr ds:[eax+50]
002A1251 51 push ecx
002A1252 8B 55 14 mov edx, dword ptr ss:[ebp-14]
002A1255 52 push edx
002A1253 FF 35 1C call dword ptr ss:[ebp-1C]
002A1259 8B 45 14 mov ecx, dword ptr ss:[ebp-14]
0018FC60 PAGE_EXECUTE_READWRITE
[ebp-28]: "PE"
0019C000
00400000
[ebp-1C]: VirtualProtect

```

Fig. 12

Next step at 002A14F6 is to allocate a buffer located at 01DC0000:

```
002A14F6 FF 55 10 call dword ptr ss:[ebp-10] [ebp+10]:VirtualA1loc
```

Fig. 13

This buffer is filled with the bytes copied from 0042d049 from the original packed PE file:

```

* 002A0F03 03 45 FC add eax,dword ptr ss:[ebp-4]
* 002A0F06 8A 08 mov cl,byte ptr ds:[eax]
* 002A0F08 88 0A mov byte ptr ds:[edx],cl
** 002A0F0A EB DD jmp 2A0EE9

```

Fig. 14

0042d049	91 A5 14 95 5F EB C2 D7 48 FB 84 95 03 04 C2 D7	ü...èÄxHü...Äx
0042d059	BC FB 84 95 5C EB C2 D7 84 FB 84 95 5C EB C2 D7	%ü...èÄx.ü...èÄx
0042d069	44 FB 84 95 5C EB C2 D7 44 FB 84 95 5C EB C2 D7	Dü...èÄxDü...èÄx
0042d079	44 FB 84 95 5C EB C2 D7 44 FB 84 95 0C EB C2 D7	Dü...èÄxDü...èÄx
0042d089	4E E0 3E 93 5C 3F BB 14 25 43 83 51 11 CA 6E 6F	Nä»...?».%.Q.Éno
0042d099	6D 8C A4 85 8E 94 9D 89 65 8E A4 76 7D 95 94 76	n...e...e...v
0042d0A9	70 0E 6E 74 3F 74 9F 7C 74 03 73 05 80 04 71 07	wfö...e...e...v

Fig. 15

This data is an encoded PE file. After copying the bytes to memory, it decodes them and writes them back to the buffer (Fig. 16a) at 01DC0000 (Fig. 16b)

```

002A1594 88 40 A4 mov ecx,dword ptr ss:[ebp-54]
002A1597 83 C1 01 add ecx,1
002A159A 89 40 A4 mov dword ptr ss:[ebp-54],ecx
002A159D 88 55 A8 mov edx,dword ptr ss:[ebp-58]
002A15A0 83 C7 01 add edx,1
002A15A3 89 55 A8 mov dword ptr ss:[ebp-58],edx
002A15A6 83 70 A4 08 cmp dword ptr ss:[ebp-5C],8
002A15A9 73 49 jmp 2A1595
002A15AC 88 45 A8 mov eax,dword ptr ss:[ebp-58]
002A15AF 38 45 F4 cmp eax,dword ptr ss:[ebp-C]
002A15B2 73 41 jmp 2A1595
002A15B4 88 45 E0 mov eax,dword ptr ss:[ebp-20]
002A15B7 88 45 F8 mov eax,dword ptr ss:[ebp-8]
002A15BA 88 4D E0 mov ecx,dword ptr ss:[ebp-20]
002A15BD 03 40 A8 add ecx,dword ptr ss:[ebp-58]
002A15C0 0F B6 11 movzx edx,byte ptr ds:[ecx]
002A15C3 88 45 A4 mov eax,dword ptr ss:[ebp-5C]
002A15C6 0F BE 4C 05 C0 movzx ecx,byte ptr ds:[ebp+eax-40]
002A15C9 03 40 A4 add ecx,dword ptr ss:[ebp-5C]
002A15CE 2B D1 sub edx,ecx
002A15D0 88 45 E0 mov eax,dword ptr ss:[ebp-20]
002A15D3 03 45 A8 add eax,dword ptr ss:[ebp-58]
002A15D6 88 10 mov byte ptr ds:[eax],dl
002A15D8 88 40 E0 mov ecx,dword ptr ss:[ebp-20]
002A15DB 03 40 A8 add ecx,dword ptr ss:[ebp-58]
002A15DE 0F B6 11 movzx edx,byte ptr ds:[ecx]
002A15E1 88 45 A4 mov eax,dword ptr ss:[ebp-5C]
002A15E4 0F BE 4C 05 C0 movzx ecx,byte ptr ds:[ebp+eax-40]
002A15E7 33 D1 xor edx,ecx
002A15E8 88 45 E0 mov eax,dword ptr ss:[ebp-20]
002A15EB 03 45 A8 add eax,dword ptr ss:[ebp-58]
002A15F1 88 10 mov byte ptr ds:[eax],dl
002A15F3 EB 9F jmp 2A1594

```

Fig. 16a

Address	Hex	ASCII
01dc0000	40 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
01dc0010	B8 00 00 00 00 00 00 00 40 00 00 00 44 FD 18 00@...Dy..
01dc0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01dc0030	E4 FD 18 00 00 00 00 00 00 00 00 00 F0 00 00 00	äY.....ð.....
01dc0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!...L!Th
01dc0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
01dc0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
01dc0070	6D 6F 64 65 2E 00 00 04 24 00 00 00 00 00 00 00	mode...\$.....
01dc0080	13 AD C9 63 57 CC A7 30 57 CC A7 30 57 CC A7 30	..ÉcxI\$0wI\$0
01dc0090	5E B4 32 30 48 CC A7 30 5E B4 23 30 68 CC A7 30	^`20kI\$0A`#0kI\$0
01dc00A0	5E B4 24 30 DC CC A7 30 70 0A C9 30 56 CC A7 30	^`\$0uI\$0p.É0vI\$0
01dc00B0	70 0A CA 30 51 CC A7 30 70 0A DC 30 70 CC A7 30	p.É0qI\$0p.Ú0pI\$0
01dc00C0	57 CC A6 30 38 CD A7 30 5E B4 2D 30 4B CC A7 30	wI`08I\$0A`-0kI\$0
01dc00D0	5E B4 33 30 56 CC A7 30 5E B4 36 30 56 CC A7 30	^`30vI\$0A`60vI\$0
01dc00E0	52 69 63 68 57 CC A7 30 00 00 00 00 00 00 00 00	RtchwI\$0.....

Fig. 16b

This stage is protected with an Anti-Debugging technique. The executable uses the following two GetTickCount calls to measure the time between the two calls (Fig. 17a and 17b). If it takes too long the executable will crash.

```
002A07D2 FF 95 A8 FE FF FF call dword ptr ss:[ebp-158] [ebp-158]:GetTickCount
002A07D5 80 81 2A FE EE EE mov dword ptr ss:[ebp-154]
```

Fig. 17a

```
002A12C2 FF 55 24 call dword ptr ss:[ebp+24] [ebp+24]:GetTickCount
```

Fig. 17b

After resolving more library function addresses and fixing the IAT of the PE file in memory, it sleeps for 258 milliseconds and jumps back to 004897D3, which we will call the **third stage** from now on.

```

002A0C29 ^ EB E3 jmp 2A0CDE
002A0C2B 68 58 02 00 00 push 258
002A0C30 FF 55 F8 call dword ptr ss:[ebp-8] [ebp-8]:Sleep
002A0C33 ^ FF A5 AC FE FF FF jmp dword ptr ss:[ebp-154] [ebp-154]:jmp 004897D3
002A0C35 33 70 mov esi,esi

```

Fig. 18

The 2nd unpacking stage, the one we have just discussed, also decodes the URL which is later used to contact the command and control server. First it allocates a buffer e.g. at 002B0000 (Fig. 19a) and reads the encrypted URL from the original sample at 004020c0, decodes it and stores it in the allocated buffer i.e. 002B0000 again (Fig. 19b).

```
002A0927  FF 95 70 FE FF FF  call dword ptr [ebp,190]  [ebp-190]:VirtualAlloc
```

Fig.19a

```
002A093E  8B 85 40 FE FF FF  mov eax,dword ptr [ebp-1C0]
002A0945  83 C0 01          add eax,1
002A0948  89 85 40 FE FF FF  mov dword ptr [ebp-1C0],eax
002A094E  8B 80 50 FE FF FF  mov ecx,dword ptr [ebp-100]
002A0954  0F B7 51 02      movzx edx,word ptr [ecx+2]
002A0958  39 95 40 FE FF FF  cmp dword ptr [ebp-1C0],edx
002A095E  70 39          jnb ZAD95E
002A0960  8B 85 48 FE FF FF  mov eax,dword ptr [ebp-188]
002A0966  03 85 40 FE FF FF  add eax,dword ptr [ebp-1C0]
002A096C  0F B6 08      movzx ecx,byte ptr [eax]
002A096E  8B 95 40 FE FF FF  mov edx,dword ptr [ebp-1C0]
002A0975  83 F2 02      xor edx,2
002A0978  83 C2 1F      add edx,1F
002A097B  2B CA      sub ecx,edx
002A097D  8B 85 44 FE FF FF  mov eax,dword ptr [ebp-18C]
002A0983  03 85 40 FE FF FF  add eax,dword ptr [ebp-1C0]
002A0989  8B 08      mov byte ptr [eax],cl
002A098B  8B 80 44 FE FF FF  mov ecx,dword ptr [ebp-18C]
002A0991  03 80 40 FE FF FF  add ecx,dword ptr [ebp-1C0]
002A0997  C6 41 01 00     mov byte ptr [ecx-1],0
002A099D  EB A2      jmp ZAD99E
002A099E  8B 95 44 FE FF FF  mov edx,dword ptr [ebp-18C]
002A09A4  0B CC 7A      mov dword ptr [ebp-18C],edx
```

Fig. 19b

The third stage (see above) is a C++ executable compiled with Visual Studio. Global object initializers allow custom classes to run during the C runtime initialization, before the apparent *WinMain* entry point. Organizing code in this way allows the malware to prepare the system survey in a way that is hidden from analysts who commence their analysis from *WinMain*. Later, when the associated code is used, the execution is masked by memory redirection and virtual function calls.

Below you can see the callback function addresses stored in the *.rdata* segment of the PE file (Fig.20) and its initialization function *InitCallbacks* (Fig.21 and Fig. 23).

```
_rdata      segment para public 'DATA' use32
            assume cs:_rdata
            ;org 4CB5ACh
CinitFunctions dd 0 ; DATA XREF: __cinit+4B70
            dd offset sub_4CAA5D
            dd offset sub_4CAA73
            dd offset sub_4CAA89
            dd offset sub_4CAA89
            dd offset sub_4CAA51
            dd offset Callbacks crt_init
            dd offset sub_4CA724
            dd offset sub_4CA73A
            dd offset sub_4CA750
            dd offset sub_4CA766
            dd offset sub_4CA77C
```

Fig. 20

```
004CA70C
004CA70C
004CA70C
004CA70C
004CA70C 000 E8 3D 7E FF FF  Callbacks crt_init proc near
004CA711 000 8B C8          call CreateCallbacks
004CA713 000 E8 7D 75 F3 FF  mov ecx, eax
004CA718 000 68 95 AA 4C 00  call InitCallbacks
004CA71D 004 E8 98 E8 FB FF  push offset CallbacksDestroy ; void (__cdecl *)()
004CA722 004 59          call _atexit
004CA723 000 C3          pop ecx
004CA723 000 C3          retn
004CA723  Callbacks crt_init endp
004CA723
```

Fig. 21

From the pre-*WinMain* C Run Time library (CRT) initialization, the Callback function list gets created and populated with an association of named strings (e.g. "OS"), later observed in the CnC traffic and several system information collection callback functions. For example a "systemFS" string in the CnC traffic, leads to a call to the *Graftor_CollectSystemVolumeInformation* function or "OS" triggers the call of *Graftor_CollectWindowsInformation*.

Fig. 22 shows an example of such function calls and pseudo code which would lead to a similar assembler code as discussed.

```

loc_4C25B2:
or     ebx, 0FFFFFFFh
push  offset aSystemFs ; "systemFS"
lea   esi, [esp+60h+WideString] ; std::wstring *
mov   [esp+60h+Status], ebx
call  std_widestring_set_wstring
mov   eax, esi
push  eax ; std::wstring *
mov   ecx, edi ; CallbackList *
mov   [esp+60h+Status], 2
call  FindCallback ; operator[]<std::wstring>
push  0 ; int
push  1 ; free
mov   dword ptr [eax], offset Graftor_CollectSystemVolumeInformation
mov   [esp+64h+Status], ebx
call  std_widestring_reset
push  offset aOs ; "OS"
lea   esi, [esp+60h+WideString] ; std::wstring *
call  std_widestring_set_wstring
mov   eax, esi
push  eax ; std::wstring *
mov   ecx, edi ; CallbackList *
mov   [esp+60h+Status], 3
call  FindCallback ; operator[]<std::wstring>
push  0 ; int
push  1 ; free
mov   dword ptr [eax], offset Graftor_CollectWindowsInformation

```

```

std::map<std::basic_string<wchar_t>, void*> CallbackList;
CallbackList[L"systemFS"] = &Graftor_CollectSystemVolumeInformation;
CallbackList[L"OS"] = &Graftor_CollectWindowsInformation;

```

Fig. 22

The created list is linked to a global address location, which is later linked back again to local variables.

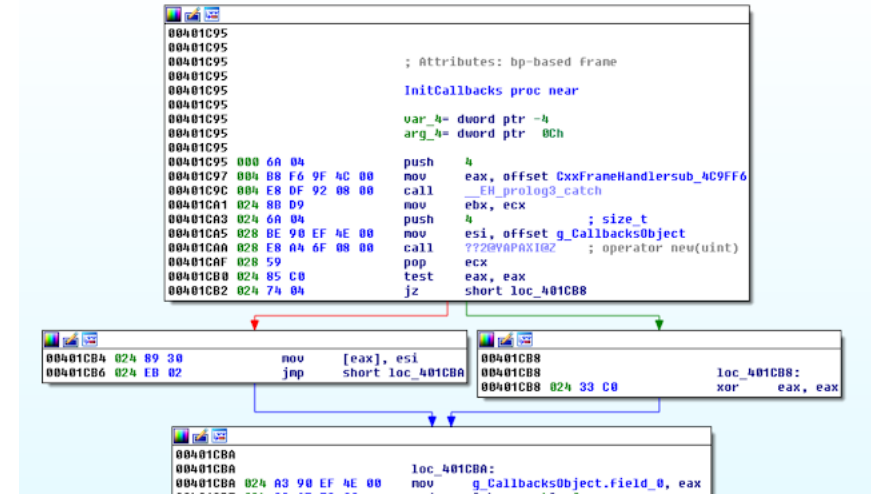


Fig. 23

Such redirection is subtle in source code, but the resulting execution means that chains of memory accesses are seen instead of just nice clean references to the object.

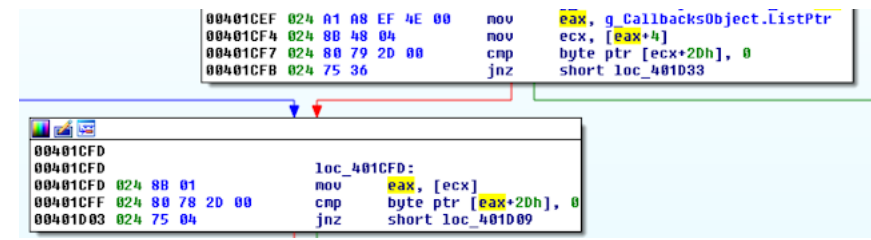


Fig. 24

Later on, a string is passed along to look up the callback and call it indirectly (Fig.25).

```

; int __cdecl CallCallbacks(std::wstring *, wchar_t *CallbackName, int callback_arg)
CallCallbacks proc near
var_40= std::wstring ptr -40h
keyval= std::wstring ptr -2Ch
var_10= dword ptr -10h
var_4= dword ptr -4
Dest= dword ptr 8
CallbackName= dword ptr 0Ch
callback_arg= dword ptr 10h

push 40h
mov eax, offset __ehandler$?_initialize@SchedulerPolicy@Concurrency@G3ARXIPAPADGZ
call __EH_prolog3
push [ebp+CallbackName] ; src
xor ebx, ebx
lea esi, [ebp+keyval] ; std::wstring *
mov [ebp+var_10], ebx
call std_wstring_set_wstring
mov eax, esi
push eax ; std::wstring *
mov ecx, offset g_CallbacksObject ; CallbackList *
mov [ebp+var_4], ebx
call FindCallback ; operator[]<std::wstring>
mov edi, [eax]
or [ebp+var_4], 0FFFFFFFh ; int
push ebx
inc ebx
push ebx ; free
call std_wstring_reset
test edi, edi
jnz short loc_4BEDCB

p+Dest] ; std::wstring *
ullString ; src
estring_set_wstring
c_4BEDF0
loc_4BEDCB:
004BEDCB 060 FF 75 10 push [ebp+callback_arg]
004BEDCE 064 8D A5 B8 lea eax, [ebp+var_48]
004BEDD1 064 50 push eax
004BEDD2 068 FF D7 call edi ; CallCallback
004BEDD4 0A8 50 nnn ecx

void CallCallbacks(wchar_t* CallbackName, void* callback_arg)
{
CALLBACKPROC CB = ((CALLBACKPROC)CallbackList[CallbackName]);
if (CB)
{
CB(callback_arg);
}
}

```

Fig. 25

By using `std::basic_string<wchar_t>` instead of just plain `wchar_t` arrays, every string interaction adds two function calls and indirection. Instead of the analyst seeing a wide string being pushed to one function, it is instead a series of three. Before significant markup is performed (or when viewed in a debugger) this is just a mess of function calls and memory manipulation. Complicating the matter is that the std library is included rather than dynamically linked, so the analyst doesn't get dll calls as hints. Further on, this 3rd stage is protected by another anti-debugging technique: the sample registers a `VectoredExceptionHandler` for `FirstChanceExceptions` (C0000005) as you can see in Fig. 26 and 27:

0048BDA0	89 48 01	mov dword ptr ds:[eax-1],ecx	
0048BDB0	FF 75 EC	push dword ptr ss:[ebp-14]	
0048BDB3	6A 01	push 1	
0048BDB5	FF 15 3C B3 4C 00	call dword ptr ds:[Set14VectoredExceptionHandler]	LPVOID VectoredHandler: 003F0024 ULONG FirstHandler = 1 int AddVectoredExceptionHandler
0048BDB8	A3 CC EC 4E 00	mov dword ptr ds:[4ECCCC],eax	

Fig. 26

003F0075	53	push ebx	
003F0076	51	push ecx	
003F0078	83 EC 04	sub esp,4	
003F0079	8B 4C 04 10	mov eax, dword ptr ds:[esp+10]	eax:VirtualProtect
003F007B	88 00	mov eax, dword ptr ds:[eax]	
003F007D	8B 18	mov ebx, dword ptr ds:[eax]	
003F007F	81 EB 05 00 00 C0	cmp ebx, C0000005	
003F0081	75 3C	jnz 3F0075	
003F0083	8B 58 10	mov ebx, dword ptr ds:[eax+10]	
003F0085	83 F8 02	cmp ebx, 2	
003F0087	7C 34	jl 3F0075	
003F0089	8B 58 18	mov ebx, dword ptr ds:[eax+18]	
003F008B	8B 00 00 3F 00	mov eax, 3F0000	
003F008D	39 18	cmp dword ptr ds:[eax],ebx	
003F008F	77 28	jnb 3F0075	
003F0091	8B 08	mov ecx, dword ptr ds:[eax]	
003F0093	83 F9 00	cmp ecx, 0	
003F0095	74 21	je 3F0075	
003F0097	01 48 04	add ecx, dword ptr ds:[eax+4]	
003F0099	3B 09	cmp ebx, ecx	
003F009B	72 05	jbe 3F0080	
003F009D	83 C0 0C	add ecx, C	
003F009F	E3 00	jmp 3F0080	
003F00A1	54	push esp	
003F00A3	FF 70 08	push dword ptr ds:[eax+8]	flNewProtect=PAGE_EXECUTE_READWRITE
003F00A5	6A 01	push 1	decSize
003F00A7	53	push ebx	lpAddress=0048B8E0
003F00A9	8B 47 43 BA 75	mov eax, <kernel32.VirtualProtect>	
003F00AB	FF 00	call ecx	eax:VirtualProtect
003F00AD	8B FF FF FF	mov eax, 0FFFFFFF	
003F00AF	E8 05	jmp 3F007A	
003F00B1	8B 00 00 00 00	mov eax, 0	
003F00B3	83 C4 04	add esp,4	
003F00B5	59	pop ecx	
003F00B7	5B	pop ebx	
003F00B9	C2 04 00	ret 4	

Fig. 27

Then it marks the code section as PAGE_NOACCESS.

```

004BB8D8 8D 45 FC      lea eax,dword ptr ss:[ebp-4]
004BB8DB      push eax
004BB8DF      push 1
004BB8E1      push dword ptr ss:[ebp+4]
004BB8E4      push dword ptr ss:[ebp+8]
004BB8E7      call dword ptr ss:[<VirtualProtect>]

```

Fig. 28a

```

00401000 00001000     ".text"           Executable code
00402000 00003000     ".rdata"         Read-only initialized data
00405000 0001A000     ".data"          Initialized data

```

Fig. 28b

This means an exception is triggered for every single instruction in this section. The exception handler function (see Fig. 27 above) overwrites the PAGE_NOACCESS access right for the memory location which caused the exception, with a PAGE_EXECUTE_READWRITE, so it can be executed. Then the exception handler function returns to the initial instruction, it can now be executed, but the next instruction is still protected by PAGE_NOACCESS and will cause the next exception. With a debugger attached, this interrupts the debugging session for every instruction. Even if the exceptions are directly passed back to the executable, it massively slows down the execution speed. At 004BB3FA the software starts preparing the internet request to the CnC server and encrypts the collected information to perform a GET request (Fig. 29a-c):

```

004BB3FA 53          push ebx
004BB3FB 53          push ebx
004BB3FC 6A 03      push 3
004BB3FE 52          push edx
004BB3FF 51          push ecx
004BB400 0F B7 8F C8 00 00 00  movzx ecx,word ptr ds:[edi+C8]
004BB407 51          push ecx
004BB408 50          push eax
004BB409 FF 74 24 28  push dword ptr ss:[esp+24]
004BB40C      call dword ptr ss:[<InternetConnectW>]

```

Fig. 29a

```

004BA122 33 06      xor ebx,ebx
004BA124 53          push ebx
004BA125 68 00 F0 01 84  push 8401F000
004BA12A 53          push ebx
004BA12B 53          push ebx
004BA12C 53          push ebx
004BA12D 50          push eax
004BA12E FF B7 24 01 00 00  push dword ptr ds:[edi+24]
004BA131 FF 75 08      push dword ptr ss:[ebp+8]
004BA134      call dword ptr ss:[<HttpOpenRequestW>]
004BA137      mov dword ptr ax,[ebx+4],eax

```

Fig. 29b

```

004BA4AC 83 C0 04      add eax,4
004BA4AF 52          push edx
004BA4B0 56          push esi
004BA4B1 53          push ecx
004BA4B2 50          push eax
004BA4B3 89 30 18      mov dword ptr ss:[ebp+18],ebx
004BA4B6 FF 15 00 85 4C 00  call dword ptr ds:[<HttpSendRequestW>]
004BA4BC 85 C0      test eax,eax

```

Fig. 29c

Talos has decrypted the GET request that is sent to the CnC server. The decoded content consists of a JSON file, which you can download [here](#).

The executable is capable of sending the following informations to the C2 server:

MAC, SID, HD serial number, username, GUID, hostname, HD size, HD devicename, Filesystem, OS version, browser version, DotNET version, Video Driver, Language Settings, Memory, system bios version, domainname, computername, several processor related parameters, number of processors, other installed adware and unwanted programs, running processes, keyboard settings, Antispyware, Firewall, Antivirus and more.

The server responds to this with an encrypted configuration file which is processed here:

```

004BA7F3 8D 45 18      lea eax,dword ptr ss:[ebp+18]
004BA7F6 50          push eax
004BA7F7 56          push esi
004BA7F8 FF 75 0C      push dword ptr ss:[ebp+C]
004BA7FB FF 75 08      push dword ptr ss:[ebp+8]
004BA7FE 89 30 18      mov dword ptr ss:[ebp+18],ebx
004BA801      call dword ptr ds:[<InternetReadFile>]
004BA809 85 C0      test eax,eax
004BA80B      jmp 2263387E61.4BA727

```

Fig. 30

The same decryption algorithm which is used for the GET request, is also used to decrypt the CnC servers response. It generates a fairly simple stream seeded by the first byte of the packet and XORs it with the data. Underneath the encryption is a simple gzip stream. The full decrypted file can be downloaded [here](#). It contains the adware and other unwanted programs the Graftor downloader is supposed to install for it's partners/customers. You can see an example in Fig. 31.

```

* chkdsk
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100

```

Fig. 31

The first URL from the 'l' key is used to download the partner executable and install it. The 'a' key is used as its command line parameters. We have yet to identify the exact meaning of all the keys; they are passed as parameters to a quite large JSON library. This library is also statically compiled into the binary. Besides the JSON library we also found a statically compiled SQLite library, we haven't fully investigated how it is used by the executable. However at this point we have enough information to detect and stop this adware downloader. The information presented so far clearly shows the sophistication of this piece of software. With the data presented in the two decoded files, you have a good idea of the capabilities of the software and the impact it has on infected systems. Graftor, and the applications that it downloads also heavily check for AV products and use various techniques to detect if it is running in a sandbox environment. These are very similar to techniques commonly observed in malware.

Attempts to identify installed AV products by installation directory (6 events)	
file	C:\Program Files\avast software\avast\setup\setup.ini
file	C:\Avira
file	C:\Program Files (x86)\McAfee\Common Framework\McTray.exe
file	C:\Program Files (x86)\McAfee\MSC\McAPExe.exe
file	C:\Program Files (x86)\AVG\Framework\Common\avgui.exe
file	C:\Program Files (x86)\Avg\AV\avgui.exe
Attempts to identify installed AV products by registry key (11 events)	
registry	HKEY_LOCAL_MACHINE\SOFTWARE\AVG\AV\
registry	HKEY_CURRENT_USER\Software\Avg\AV\
registry	HKEY_LOCAL_MACHINE\SOFTWARE\AVAST Software\Avast\
registry	HKEY_CURRENT_USER\Software\AVAST Software\Avast\
registry	HKEY_LOCAL_MACHINE\SOFTWARE\ESET\ESET Security\
registry	HKEY_LOCAL_MACHINE\SOFTWARE\ESET\NOD\
registry	HKEY_CURRENT_USER\Software\G Data\AntiVirenKit\
registry	HKEY_CURRENT_USER\SOFTWARE\KasperskyLab\
registry	HKEY_LOCAL_MACHINE\SOFTWARE\McAfee\McTray\Plugins\VSEPlugin\
registry	HKEY_CURRENT_USER\Software\McAfee\DesktopProtection\
registry	HKEY_LOCAL_MACHINE\SOFTWARE\McAfee.com\
Checks the version of Bios, possibly for anti-virtualization (1 event)	
registry	HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\SystemBiosVersion

Fig. 32a

ⓘ Detects VirtualBox through the presence of a device (1 event)	
file	\\?\VBoxMiniRdrDN
ⓘ Detects VirtualBox through the presence of a file (1 event)	
dll	VBoxHook.dll
ⓘ Detects VMWare through the in instruction feature (1 event)	
Time & API	Arguments
Aug. 10, 2017, 11:46 p.m.	<pre> stacktrace: __Get__JsonConfigImpl__Instance__-0x3d68 agloader+0xc5e8 @ 0x6fcac5e8 RunLoader+0x168 0x13cbd8a __Get__JsonConfigImpl__Instance__+0xbc97 anonymizerlauncher+0x1b597 @ 0x13cb597 __Get__ __Get__JsonConfigImpl__Instance__+0xb93d anonymizerlauncher+0x1b23d @ 0x13cb23d __Get__JsonConf VerifyConsoleIoHandle-0xb3 kernel32+0x133ca @ 0x74e833ca RtlInitializeExceptionChain+0x63 RtlAl RtlAllocateActivationContextStack-0xce ntdll+0x39ea5 @ 0x772e9ea5 exception.instruction_r 81 fb 68 58 4d 56 0f 94 45 e7 5b 59 5a c7 45 exception.instruction: in eax, dx exception.exception_code: 0xc0000096 exception.symbol: __Get__JsonConfigImpl__Instance__-0x3c0b agloader+0xc745 exception.address: 0x6fcac745 registers.esp: 3601532 registers.edi: 5528040 registers.eax: 1447909480 registers.ebp: 3601588 registers.edx: 22104 registers.ebx: 0 registers.esi: 5598536 registers.ecx: 10 </pre>

Fig. 32b

ⓘ Checks amount of memory in system, this can be used to detect virtual machines that have a low amount of memory available (1 event)		
Time & API	Arguments	Status
Aug. 10, 2017, 11:43 p.m.	GlobalMemoryStatusEx	success

Fig. 32c

ⓘ A process attempted to delay the analysis task. (2 events)	
description	2263387661.exe tried to sleep 263 seconds, actually delayed analysis time by 263 seconds
description	explorer.exe tried to sleep 240 seconds, actually delayed analysis time by 240 seconds

Fig. 32d

ⓘ Executes one or more WMI queries which could be used to identify virtual machines (19 events)	
--	--

Fig.32e

The software makes many excessive API calls such as the following (Fig. 33) which has the effect of polluting sandbox analysis.

Aug. 10, 2017, 11:42 p.m.	module_handle: 0x76cd0000	failed
FindResourceExW	type: #0 name: #14 language_identifier: 0	

Fig. 33

Conclusion

Graftor continues to be one of the most notorious potentially-unwanted-software downloaders we see in the wild. Users may be unaware that it is being bundled and executed as part of the freeware installation, since these installation files silently execute Graftor alongside the freeware. Once Graftor is running, it exfiltrates a huge amount of user and machine identifiable information and installs additional potentially-unwanted-applications from its partners. The downloader requests administrative rights on the local machine, with this access, it can do anything it wants to do on the user's machine.

Solutions such as AMP for endpoints and AMP on network devices give administrators visibility of when software such as Graftor, and the

further packages it downloads, are installed on devices. Similarly, network based detection can identify and block the CnC activity (Short SID 44214). Thought should be given to blocking access to freeware websites to prevent the download of the Graftor installer. However, much freeware does not come bundled with Graftor and may be of great use to some users.

At the end of the day, keep in mind that if the software is free, you might be the product. Anyone using freeware should closely review the EULA before installing it. We know it is painful, but trying to remove this kind of software is likely more painful.

Coverage

Additional ways our customers can detect and block this threat are listed below.

PRODUCT	PROTECTION
AMP	✓
CWS	✓
Email Security	✓
Network Security	✓
Threat Grid	✓
Umbrella	✓
WSA	✓

Advanced Malware Protection ([AMP](#)) is ideally suited to prevent the execution of the malware used by these threat actors.

[CWS](#) or [WSA](#) web scanning prevents access to malicious websites and detects malware used in these attacks.

[Email Security](#) can block malicious emails sent by threat actors as part of their campaign.

The Network Security protection of [IPS](#) and [NGFW](#) have up-to-date signatures to detect malicious network activity by threat actors.

[AMP Threat Grid](#) helps identify malicious binaries and build protection into all Cisco Security products.

[Umbrella](#), our secure internet gateway (SIG), blocks users from connecting to malicious domains, IPs, and URLs, whether users are on or off the corporate network

IOC

Alternate Data Streams(ADS):

C:\Users\dex\AppData\Local\Temp\2263387661.exe:Zone.Identifier

C:\Users\dex\AppData\Local\Temp\QBPO5ppcuHJG.exe:tmp

C:\Users\dex\AppData\Local\Temp\2263387661.exe:tmp

C:\Users\dex\AppData\Local\Temp\AyWdp7tHPleU.exe:tmp

C:\Windows\System32\regsvr32.exe:Zone.Identifier **Hashes:**

2263387661.exe (Graftor Dropper)

9b9ce661a764d84a4636812e1dfcb03b (MD5)

Fd3ccf65eab21a77d2e440bd23c59d52e96a03a4 (SHA1)

41474cd23ff0a861625ec1304f882891826829ed26ed1662aae2e7ebbe3605f2 (SHA256)

Dumped 2nd stage:

40bde09fc059f205f67b181c34de666b (MD5)

99c7627708c4ab1fca3222738c573e7376ab4070 (SHA1)

Eefdbe891e35390b84181eabe0ace6e202f5b2a050e800fb8e82327d5e57336d (SHA256)

Dumped 3rd stage:

1e9f40e70ed3ab0ca9a52c216f807eff (MD5)

7c4cd0ff0e004a62c9ab7f8bd991094226eca842 (SHA1)

5eb2333956bebb81da365a26e56fea874797fa003107f95cda21273045d98385 (SHA256)

URLs:

Command and Control Server GET Request:

hxxp://kskmasdqsjuzom[.]regularfood[.]jgdn/JJZGF0YV9maWxlcz0yMyZ0eXBIPXN0YXRpYyZuYW1IPVRIbXAINUMyMjYzZmzg3NjYxLmV4ZSzybml

Set-Cookie: GSID=3746aecf3b94384b9de7201584e7d88; expires=Sat, 12-Aug-2017 15

Command and Control Server POST Request

hxxp://kskmasdqsjuzom[.]regularfood[.]gdn/J/ZGF0YV9maWxlcz0yMyZ0eXBIPXN0YXRpYyZuYW1IPVRIbXAINUMyMjYzMzg3NjYxLmV4ZSzybm

Set-Cookie: GSID=3746aecf3b94384b9de720158c4e7d88; expires=Sat, 12-Aug-2017 15

Domains from sandbox run:

arolina[.]torchpound[.]gdn
binupdate[.]mail[.]ru
crl[.]microsoft[.]com
dreple[.]com
gambling577[.]xyz
jvusdtufhlreari[.]twiceprint[.]gdn
kskmasdqsjuzom[.]regularfood[.]gdn
mentalaware[.]gdn
mrds[.]mail[.]ru
nottotrack[.]com
plugpackdownload[.]net
s2[.]symcb[.]com
sputnikmailru[.]cdnmail[.]ru
ss[.]symcd[.]com
xml[.]binupdate[.]mail[.]ru

Snort Rules:

SID 44214