MalwareAnalysisReports/AmateraStealer/Amatera shark.exe.md at main · VenzoV/MalwareAnalysisReports

By VenzoV

Archived: 2026-04-05 14:46:49 UTC

Sample

This sample is pulled from the Amatera config from the C2. It is fetched from the following path:

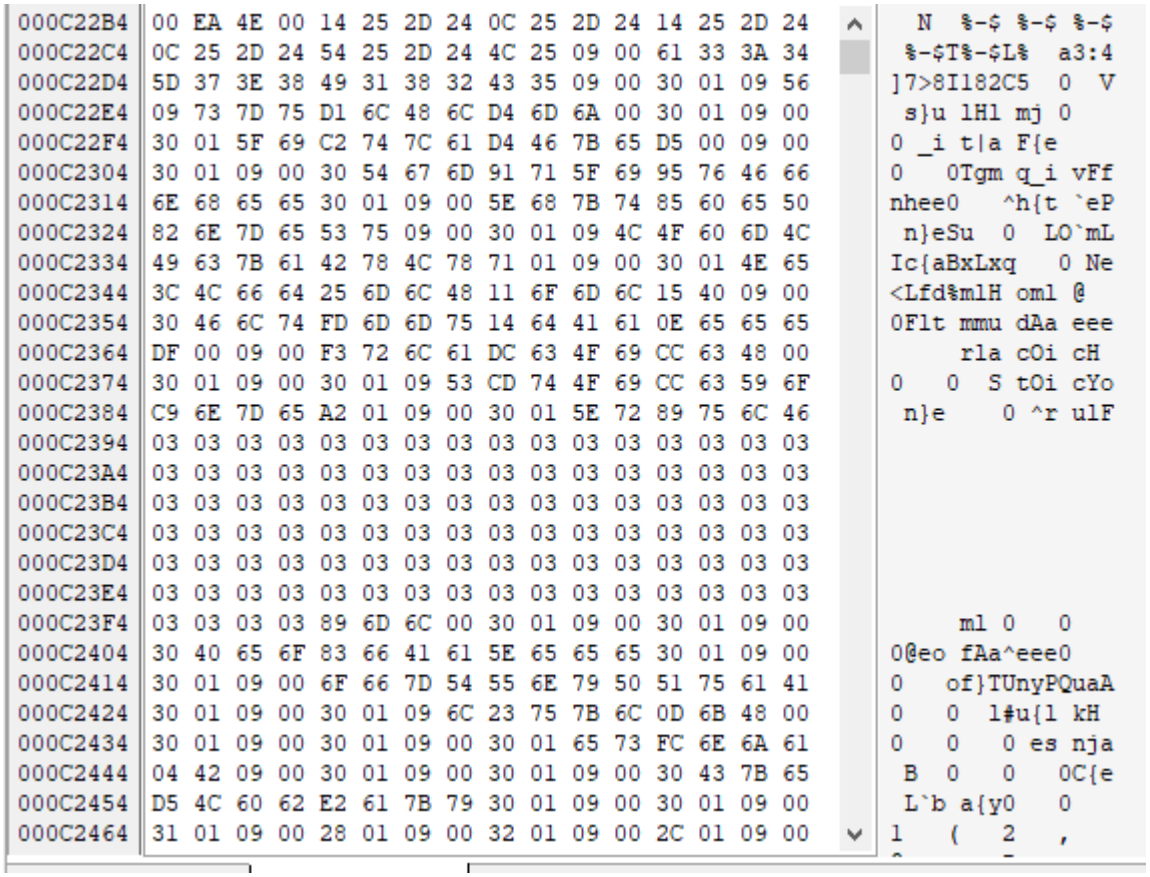
- h4.possumdefense.digital/shark.bin

SHA256
9FC9558C681F0370B1BA1F7B79551B2B253647EAC3C47F10EE4FE96F1FAA8B24
From PeStudio we can immedtialy see this is a 32 bit binary compiled with delphi.
Checking for further leads, the resource section contains an unknown data blob which will actually be the next stage and important to track.

property	value
md5	FD18F6EB3BE1C34B672F75D786BCBE9E
sha1	0FAE2C0782423FBAAB9467E8236F87FDF2298A9E
sha256	9FC9558C681F0370B1BA1F7B79551B2B253647EAC3C47F10EE4FE96F1FAA8B24
first-bytes-hex	4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00 00
first-bytes-text	M Z P .. .. @ .. ..
file-size	8359712 bytes
entropy	6.018
imphash	184E974F749059E83D5E287EB01D156A
signature	BobSoft Mini Delphi -> BoB / BobSoft

name	instance	signature	location	size (7558719 byt...	file-ratio (90.42%)	hash	entropy	language	first-bytes-hex
HREGT	F3J486J239	unknown	.rsrc:0x000C22B4	7522416	89.98 %	B006A7571BA7DCDE35EA768F7DA76694	5.832	English-US	00 EA 4E 00 14
string-table	4073	string-table	.rsrc:0x007F1E2C	1260	0.02 %	ADAEFE4501F8179F863C9FBADA50810D	3.290	neutral	1D 00 4D 00 69
string-table	4078	string-table	.rsrc:0x007F33B4	1308	0.02 %	D9D0BEF92F7DADB542DA3BD16C92D71E	3.164	neutral	36 00 53 00 69
string-table	4079	string-table	.rsrc:0x007F38D0	1340	0.02 %	B69D7DA41D7F07AE048AD9C100C90E41	3.186	neutral	2F 00 56 00 61
icon	1	icon	.rsrc:0x007F1574	744	0.01 %	8738B36430A86192C6C538A84908B4DC	3.852	English-US	28 00 00 00 20
version	1	version	.rsrc:0x007F7684	628	0.01 %	8F4EB56480F6045FA59821F67C8407B6	3.057	English-US	74 02 34 00 00
string-table	4072	string-table	.rsrc:0x007F1B30	764	0.01 %	F65C986167189C2F865ECF50C971A776	3.323	neutral	25 00 41 00 78
string-table	4074	string-table	.rsrc:0x007F2318	1172	0.01 %	5F1F2B7A844D8569BC9292EC3DD42A06	3.216	neutral	1C 00 44 00 75
string-table	4075	string-table	.rsrc:0x007F27AC	932	0.01 %	A33B95AA89718A0A9EFB6ECDA56B9C5F	3.194	neutral	24 00 43 00 6F
string-table	4076	string-table	.rsrc:0x007F2B50	1008	0.01 %	778128C4E4C7AD25CA662D828E53A3BE	3.245	neutral	2C 00 41 00 67
string-table	4077	string-table	.rsrc:0x007F2F40	1140	0.01 %	7469538B0F61768FBF59348D07947E36	3.233	neutral	1C 00 55 00 6E
string-table	4080	string-table	.rsrc:0x007F3E0C	560	0.01 %	75EA859759714B016B6C8D0B4F40DDA7	3.380	neutral	10 00 49 00 6E

Initial contents of the resource section:



```
000C22B4 00 EA 4E 00 14 25 2D 24 0C 25 2D 24 14 25 2D 24   N  %-$ %-$ %-$
000C22C4 0C 25 2D 24 54 25 2D 24 4C 25 09 00 61 33 3A 34   %-$T%-$L%  a3:4
000C22D4 5D 37 3E 38 49 31 38 32 43 35 09 00 30 01 09 56   ]7>8I182C5  0  V
000C22E4 09 73 7D 75 D1 6C 48 6C D4 6D 6A 00 30 01 09 00    s}u lHl mj 0
000C22F4 30 01 5F 69 C2 74 7C 61 D4 46 7B 65 D5 00 09 00   0 _i t|a F{e
000C2304 30 01 09 00 30 54 67 6D 91 71 5F 69 95 76 46 66   0   0Tgm q_i vFf
000C2314 6E 68 65 65 30 01 09 00 5E 68 7B 74 85 60 65 50   nhee0   ^h{t `eP
000C2324 82 6E 7D 65 53 75 09 00 30 01 09 4C 4F 60 6D 4C    n}eSu  0  LO`mL
000C2334 49 63 7B 61 42 78 4C 78 71 01 09 00 30 01 4E 65   Ic{aBxLxq   0 Ne
000C2344 3C 4C 66 64 25 6D 6C 48 11 6F 6D 6C 15 40 09 00   <Lfd%mlH oml @
000C2354 30 46 6C 74 FD 6D 6D 75 14 64 41 61 0E 65 65 65   0Flt mmu dAa eee
000C2364 DF 00 09 00 F3 72 6C 61 DC 63 4F 69 CC 63 48 00        rla cOi cH
000C2374 30 01 09 00 30 01 09 53 CD 74 4F 69 CC 63 59 6F   0   0  S tOi cYo
000C2384 C9 6E 7D 65 A2 01 09 00 30 01 5E 72 89 75 6C 46    n}e    0 ^r ulF
000C2394 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23A4 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23B4 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23C4 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23D4 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23E4 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
000C23F4 03 03 03 03 89 6D 6C 00 30 01 09 00 30 01 09 00        ml 0   0
000C2404 30 40 65 6F 83 66 41 61 5E 65 65 65 30 01 09 00   0@eo fAa^eee0
000C2414 30 01 09 00 6F 66 7D 54 55 6E 79 50 51 75 61 41   0   of}TUnyPQuaA
000C2424 30 01 09 00 30 01 09 6C 23 75 7B 6C 0D 6B 48 00   0   0  l#u{l kH
000C2434 30 01 09 00 30 01 09 00 30 01 65 73 FC 6E 6A 61   0   0   0 es nja
000C2444 04 42 09 00 30 01 09 00 30 01 09 00 30 43 7B 65    B  0   0   0C{e
000C2454 D5 4C 60 62 E2 61 7B 79 30 01 09 00 30 01 09 00    L`b a{y0   0
000C2464 31 01 09 00 28 01 09 00 32 01 09 00 2C 01 09 00   1   (   2   ,
```

Analysis First Stage

To identify the routine responsible for loading and decrypting the resource file, a debugger was initially used to extract the next stage directly. However, by analyzing the binary in IDR (Interactive Delphi Reconstructor), we can gain deeper insight into the internal structure of the Delphi executable.

For a quick solution, I searched for the resource ID within the strings and identified the function referencing it. This function immediately reveals the following sequence:

- The resource is loaded or created.
- VirtualAllocis called to allocate memory.
- ReadBuffer is used to read the resource contents.
- A push ecx followed by a ret instruction effectively results in a jmp ecx.
-

This pattern strongly suggests that the resource is decrypted in memory and execution is then transferred to the unpacked second stage.

```
004AEB1F        push        4AECE8;'f3j486j239'
004AEB24        push        4AECF4
004AEB29        mov         ecx,dword ptr ds:[4B3668];gvar_004B3668
004AEB2F        mov         dl,1
004AEB31        mov         eax,[41A14C];TResourceStream
004AEB36        call        TResourceStream.Create;TResourceStream.Create
004AEB3B        mov         ebx,eax
004AEB3D        mov         eax,4069B4;kernel32.VirtualAlloc:Pointer

004AEB61        call        dword ptr [ebp-48]
004AEB64        mov         dword ptr [ebp-1C],eax
004AEB67        mov         eax,dword ptr [ebp-3C]
004AEB6A        sub         eax,0A5F
004AEB6F        push        eax
004AEB70        sub         edi,4E3
004AEB76        push        edi
004AEB77        mov         eax,ebx
004AEB79        mov         edx,dword ptr [eax]
004AEB7B        call        dword ptr [edx];TStream.GetSize
004AEB7D        add         eax,esi
004AEB7F        push        eax
004AEB80        push        0
004AEB82        call        dword ptr [ebp-48]
004AEB85        mov         dword ptr [ebp-24],eax
004AEB88        mov         eax,ebx
004AEB8A        mov         edx,dword ptr [eax]
004AEB8C        call        dword ptr [edx];TStream.GetSize
004AEB8E        mov         ecx,eax
004AEB90        mov         edx,dword ptr [ebp-1C]
004AEB93        mov         eax,ebx
004AEB95        call        TStream.ReadBuffer

004AEC80        mov         eax,dword ptr [ebp-24]
004AEC83        xor         dword ptr [eax],edi
004AEC85        add         dword ptr [ebp-14],4
004AEC89        add         dword ptr [ebp-24],4
004AEC8D        mov         eax,dword ptr [ebp-14]
004AEC90        cmp         eax,dword ptr [ebp-28]
004AEC93>       jb          004AEC18
004AEC95        mov         eax,1000
004AEC9A        mov         edx,dword ptr [ebp-20]
004AEC9D        add         edx,dword ptr [ebp-2C]
004AECA0        sub         edx,eax
004AECA2        add         edx,4
004AECA5        mov         dword ptr [ebp-18],edx
004AECA8        mov         eax,0A6A
004AECAD        push        dword ptr [ebp-4]
004AECB0        mov         eax,0A6A
004AECB5        push        dword ptr [ebp-8]
004AECB8        mov         eax,0A6A
004AECBD        push        dword ptr [ebp-10]
004AECC0        mov         eax,0A6A
004AECC5        push        dword ptr [ebp-0C]
004AECC8        push        dword ptr ds:[4B3668];gvar_004B3668
004AECCE        push        dword ptr [ebp-14]
004AECD1        xor         ecx,ecx
004AECD3        add         ecx,dword ptr [ebp-18]
004AECD6        push        ecx
004AECD7        ret
```

Analysis Second Stage

The second stage is pretty short is just used to load another binary which is embedded inside. Once again push + ret is used again to jmp to the code to run.

```
004ee7a0    int32_t _start(int32_t arg1, int32_t arg2, int32_t arg3, int32_t arg4, void* arg5)

004ee7a0    {
004ee7a0        int32_t var_10 = 0;  /* "$$$$$$$$$$$$$$$$$$$$$" */
004ee7c0        void* pPe;
004ee7c0        sub_4edf80(&pPe, 0, 0x384);  /* "$$$$$$$$$$$$$$$$$$$$$" */
004ee7da        int32_t var_390 = arg2;
004ee7e3        int32_t var_394 = arg3;
004ee7ec        int32_t var_398 = arg4;
004ee7f5        pPe = arg5;
004ee802        sub_4edc80(&pPe);
004ee80f        void* const var_3b0 = 0xffbff000;
004ee825        int32_t var_378;
004ee825        int32_t var_40;
004ee825
004ee825        if (__return_addr)
004ee83a            var_378 = arg1;
004ee825        else
004ee82c            var_378 = var_40(0);  /* "$$$$$$$$$$$$$$$$$$$$$" */
004ee852        void* eax_5 = var_378 + *(uint32_t*)(var_378 + 0x3c);
004ee86b        uint32_t var_384 = (uint32_t)*(uint16_t*)((char*)eax_5 + 0x16);
004ee880        int32_t var_37c = var_378 + *(uint32_t*)((char*)eax_5 + 0x28);
004ee88f        int32_t var_374 = *(uint32_t*)((char*)eax_5 + 0x50);
004ee8ca        void* allocatedMemory = mw_VirtualAlloc(0x4eda00);
004ee8ed        mw_memcpy(allocatedMemory, &global_mz, 0x4eda00);
004ee903        int32_t result = mw_rLoading(&pPe, allocatedMemory);
004ee903
```

```
004ee928  8b9584fcffff        mov     edx, dword [ebp-0x37c {var_380}]
004ee92e  8bb570fcffff        mov     esi, dword [ebp-0x390 {var_394}]
004ee934  8bbd74fcffff        mov     edi, dword [ebp-0x38c {var_390}]  {".reloc"}
004ee93a  8ba568fcffff        mov     esp, dword [ebp-0x398]
004ee940  8bad6cfcffff        mov     ebp, dword [ebp-0x394 {var_398}]
004ee946  52                  push    edx
004ee947  c3                  retn
```

Third stage

The third stage of the malware is more intriguing. It can be carved directly from the second stage using any hex editor. The resulting binary is fairly large, and upon decompilation, it quickly becomes evident that heavy obfuscation is in play.

Initial string analysis reveals telltale signs of a Go binary, specifically due to the presence of obfuscated .go package names and runtime artifacts. Further analysis indicates that the binary has been obfuscated using garble, an open-source tool for Go code obfuscation. Notably, it appears to have been built using a **newer version** of Garble that introduces breaking changes to tools like GoReSym and GoStringUngarbler—both of which failed to function correctly in this context (unless there was a usage error).

To verify this, I compiled a simple "Hello, World" Go program and obfuscated it using the latest version of Garble. As expected, both GoReSym and GoStringUngarbler were unable to parse or recover symbol and string information, confirming the observed breakage.

EDIT Checking further, I noticed I made a mistake. The literals are NOT encrypted, garble was run without -literals tab. The strings can be found inside the binary and are built at runtime.

Following the start of a Garbled 32 bit binary.

```
void __fastcall sub_46ff90(int32_t arg1) __noreturn
0046ff90    {
0046ff90        void arg_4;
0046ff9e        void* var_4 = &arg_4;
0046ffbd        bool p = /*   bool p = unimplemented  {and esp, 0xfffffff0} */;
0046ffbd        bool a = /* undefined */;
0046ffbd        void* const __return_addr_1;
0046ffbd        bool z = !&__return_addr_1;
0046ffbd        bool s = &__return_addr_1 < 0;
0046ffd4        void var_10028;
0046ffd4        data_8d5aa8 = &var_10028;
0046ffd7        data_8d5aac = &var_10028;
0046ffda        data_8d5aa0 = &var_10028;
0046ffdd        data_8d5aa4 = &__return_addr_1;
0046ffe2        bool d;
0046ffe2        int32_t var_98_1 = ((d ? 1 : 0) << 0xa | (s ? 1 : 0) << 7 | (z ? 1 : 0) << 6
0046ffe2            | (a ? 1 : 0) << 4 | (p ? 1 : 0) << 2) ^ 0x200000;
0046ffea        int32_t var_98_2 = (TEST_BITD(var_98_1, 0xb) ? 1 : 0) << 0xb
0046ffea            | (TEST_BITD(var_98_1, 0xa) ? 1 : 0) << 0xa
0046ffea            | (TEST_BITD(var_98_1, 7) ? 1 : 0) << 7 | (TEST_BITD(var_98_1, 6) ? 1 : 0) << 6
0046ffea            | (TEST_BITD(var_98_1, 4) ? 1 : 0) << 4 | (TEST_BITD(var_98_1, 2) ? 1 : 0) << 2
0046ffea            | (TEST_BITD(var_98_1, 0) ? 1 : 0);
0046ffea
0046fff5        if ((var_98_2 ^ ((d ? 1 : 0) << 0xa | (s ? 1 : 0) << 7 | (z ? 1 : 0) << 6
0046fff5            | (a ? 1 : 0) << 4 | (p ? 1 : 0) << 2)) & 0x200000)
0046fff5        {
00470029            int32_t eax_4;
00470029            int32_t ecx;
00470029            int32_t edx_1;
```

For now the only functionlity recovered is that the shark.exe is added as scheduled task for persistence.

```
String > C:\\Users\\Dynamic\\Desktop\\shark.exe
String > cldbd296
DLL Loaded: 77180000 C:\Windows\SysWOW64\ws2_32.dll
String > cmd
String > cmd /c \"schtasks /create /f /sc MINUTE /mo 1 /tn shark /tr C:\\ProgramData\\shark.exe\"
Thread 6548 exit
Thread 716 exit
Thread 4568 exit
Thread 7932 exit
Thread 5312 exit
Thread 2644 exit
Thread 3812 exit
Thread 6740 exit
Thread 3900 exit
EXCEPTION_DEBUG_INFO:
           dwFirstChance: 1
           ExceptionCode: C0000005 (EXCEPTION_ACCESS_VIOLATION)
         ExceptionFlags: 00000000
       ExceptionAddress: 24BC8B00
       NumberParameters: 2
ExceptionInformation[00]: 00000008 DEP Violation
ExceptionInformation[01]: 24BC8B00 Inaccessible Address
First chance exception on 24BC8B00 (C0000005, EXCEPTION_ACCESS_VIOLATION)!
```

Attaching to the running binary the strings in memory contain 1 malicious IP address. Also makes DNS query to binance domain

- data-seed-prebsc-2-s1.binance.org
- 109.172.87.40

0xa4e8d0	26	109.172.87.40
0xa4eb00	26	109.172.87.40
0xa4eb28	26	109.172.87.40
0xa4eb50	26	109.172.87.40
0xa4eb78	26	109.172.87.40
0xa4eba0	26	109.172.87.40
0x140a090	13	109.172.87.40
0x140a190	13	109.172.87.40

```
025-07-07 10:47:38  HTTP connection, method: GET, URL: http://ctldl.windowsupdate.com/msdownload/update/v3
html
025-07-07 10:50:35  DNS connection, type: A, class: IN, requested name: data-seed-prebsc-2-s1.binance.org
025-07-07 10:50:35  Last simulated date in log file
```

My analysis for now ends here until I am able to fully recover more from the binary.

References

- https://cloud.google.com/blog/topics/threat-intelligence/gostringungarbler-deobfuscating-strings-in-garbled-binaries
- https://research.openanalysis.net/garble/go/obfuscation/strings/2023/08/03/garble.html

---