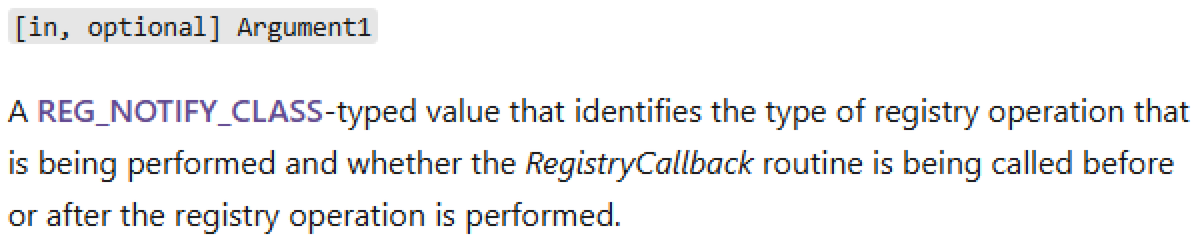
Rehabilitating Registry Tradecraft with RegRestoreKey

originhq.com/blog/rehabilitating-registry-tradecraft-with-regrestorekey

Michael Barclay

November 13, 2025


`[in, optional] Argument1`

A REG_NOTIFY_CLASS-typed value that identifies the type of registry operation that is being performed and whether the *RegistryCallback* routine is being called before or after the registry operation is performed.

Next Generation
Endpoint Security

If you have ever attempted to test the limits of out-of-the-box EDR detection strategies, you have likely found ample evasion opportunities that revolve around simply re-implementing or tweaking some existing (and well-covered) tradecraft. While these are a win in isolation, they often only evade naive detection strategies that are focused on specific execution mechanisms or superficial characteristics of observable artifacts like files and processes. These minute modifications of existing tradecraft require a similarly minor change in defensive capability to account for them.

So what does it look like to move beyond these superficial evasion attempts and force defenders to make big changes in how they collect telemetry and write detection rules?

This post will take registry primitives as an example to explore how we can (a) think critically about the ways that EDR products detect specific procedures, (b) design tooling that takes away their ability to observe those procedures, and (c) force vendors to make meaningful changes to telemetry collection and detection strategies.

Creating Services in the Status Quo

I have chosen to focus on service creation in this post because it is straightforward, but still requires the caller to both create a new registry key AND several specific values within that key. EDR products have robust coverage for standard approaches to registry primitives, so this example forces us to think creatively if we hope to create a service in a way that removes crucial telemetry. This post is really about registry primitives, not services.

Win32 API methods like `CreateServiceW` (and its lower level RPC equivalent `RCreateServiceW`) result in the creation of a new registry key and values that represent the configuration details of a new service. While the latter is responsible for formally *registering* the service with the Service

Control Manager (SCM) so that it can be started without a reboot, this is not a strictly necessary step. Calling any of the CreateService methods (whether at the RPC layer or above it) will also result in the generation of two well-known events in the Windows Security Event Log:

- 4697(S): A service was installed in the system.
- 7045: A new service was installed in the system.

On reboot, the SCM will parse all of the subkeys under `HKLM\SYSTEM\CurrentControlSet\Services\` and register each service with the SCM, so we really only need to create the registry representation of a service to be able to use it for something like persistence. With that in mind, I am going to focus my efforts on registry interaction, rather than finding less obvious ways to make Win32 or RPC calls that will inevitably generate well-known signal in the Event Log regardless of how I call them.

In the vast majority of cases, application authors will create new keys and values in the registry through Win32 API calls like `RegCreateKeyEx` and `RegSetValueEx` (or their Native API and system call equivalents). If you were to consider this the only means to create new keys and values, the landscape for evasion is relatively limited. But some additional consideration of *how* EDR products observe these operations yields some additional ground for research.

Observing Registry Interaction in the Status Quo

If we move past the approach where we call `CreateService/RCreateService` and assume that we won't have 4697 or 7045 events to observe, we have to identify a means to observe the creation of arbitrary registry keys and values to observe service creation. I will discuss the two most common ways to collect this telemetry—Event Tracing for Windows (ETW) and registry callback routines in the kernel. There are plenty of readily available resources online that discuss the structure of both of these telemetry generation mechanisms, so I will only describe the ways that EDR agents frequently use them.

Microsoft-Windows-Kernel-Registry ETW

The primary ETW provider for registry events is a kernel provider named `Microsoft-Windows-Kernel-Registry` (`{70eb4f03-c1de-4f73-a051-33d13d5413bd}`), which generates the following common events (among others):

CreateKey	EnumerateValueKey
SetValueKey	QueryMultipleValueKey
OpenKey	SetInformationKey
DeleteKey	FlushKey
QueryKey	CloseKey

CreateKey	EnumerateValueKey
DeleteValueKey	QuerySecurityKey
QueryValueKey	SetSecurityKey
EnumerateKey	

`Microsoft-Windows-Kernel-Registry` also provides a collection of performance tracking events with dizzyingly long template names and a limited number of fields documented in the manifest. I will discuss some of these events later in the post, but the above list represents the majority of ETW events that your average EDR sensor would ever think to collect.

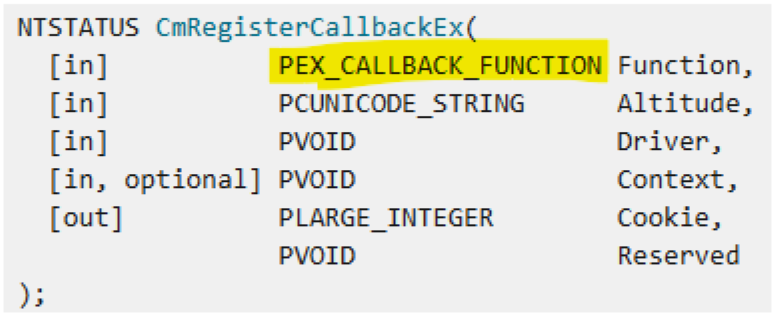
When a sensor creates an Event Tracing Session to ingest ETW events, it must specify what subset of events it wishes to receive and, under certain circumstances, the verbosity of those events. Sensor developers must weigh the performance hit of collecting a given event with the value that it provides to the entire product's observation and detection capabilities.

Luckily, the act of modifying which events a trace session ingests is well-documented for common providers. However, you are stuck with whatever Microsoft has decided to include in those events. Any additional context needed to make sense of that event data must be collected elsewhere and correlated in some way.

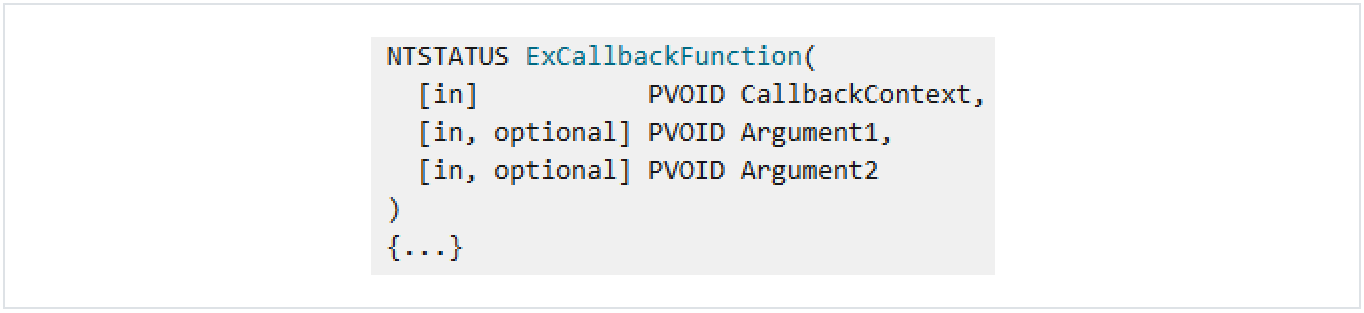
Registry Callback Routines

If we don't want to (or can't) collect registry telemetry from ETW in user mode, we will have to install a kernel driver and register a registry callback routine. These routines specify a type of interaction with the registry (creating a key, writing a value, etc.) and invoke a callback function when that type of interaction is handled by the relevant kernel driver.

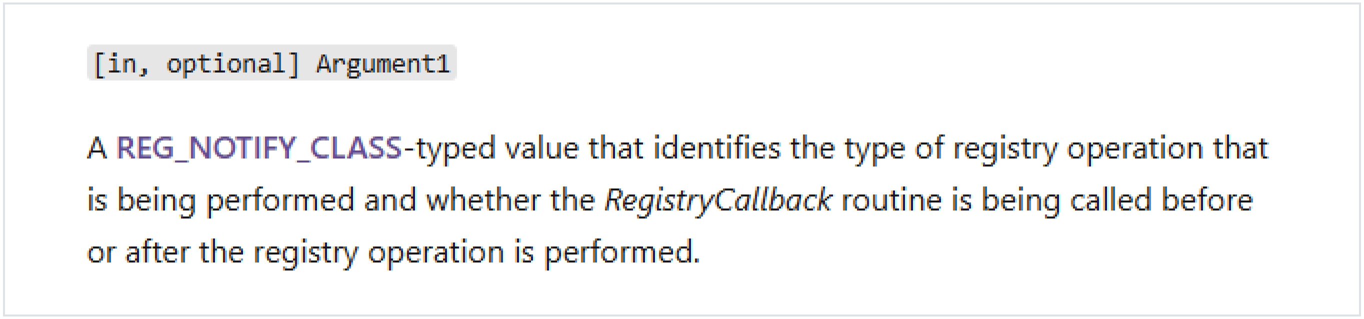
When you want to register a kernel callback routine for a registry operation, you use the `CmRegisterCallback` NT API function. When you call this function, you must supply a pointer to the RegistryCallback routine you wish to register, which is of the type `EX_CALLBACK_FUNCTION`.


```
NTSTATUS CmRegisterCallbackEx(
  [in]           PEX_CALLBACK_FUNCTION Function,
  [in]           PCUNICODE_STRING      Altitude,
  [in]           PVOID                 Driver,
  [in, optional] PVOID                 Context,
  [out]          PLARGE_INTEGER        Cookie,
                 PVOID                 Reserved
);
```

If we look at the parameters for these callback functions, we see that the second parameter (`Argument1`) is a value that specifies the type of registry operation we wish to observe and intercept in the kernel. This value must be sourced from the `REG_NOTIFY_CLASS` enumeration, which is defined by Microsoft.



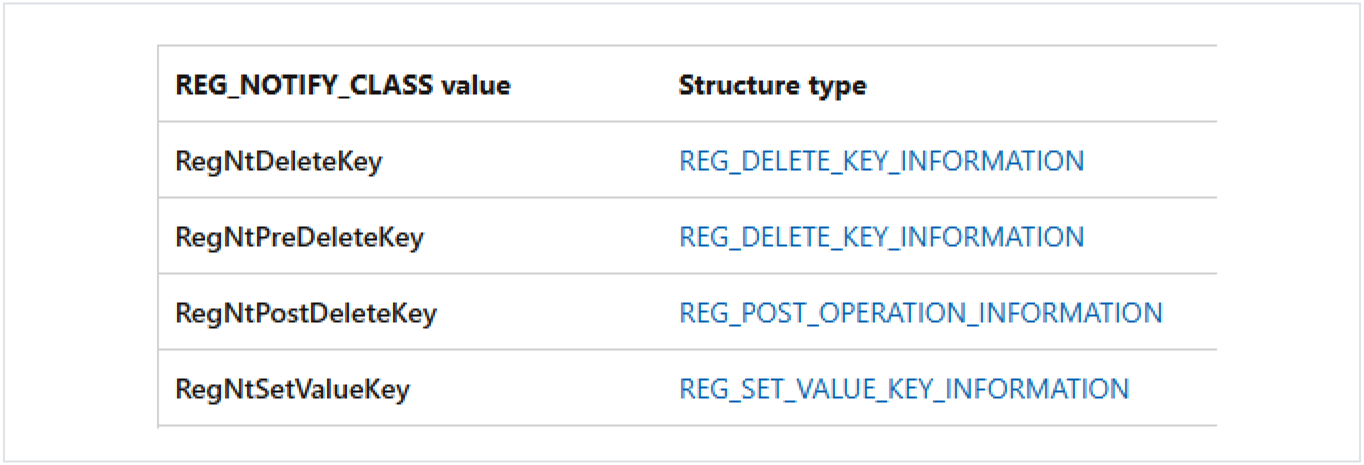
```
NTSTATUS ExCallbackFunction(
  [in]           PVOID CallbackContext,
  [in, optional] PVOID Argument1,
  [in, optional] PVOID Argument2
)
{...}
```



`[in, optional] Argument1`

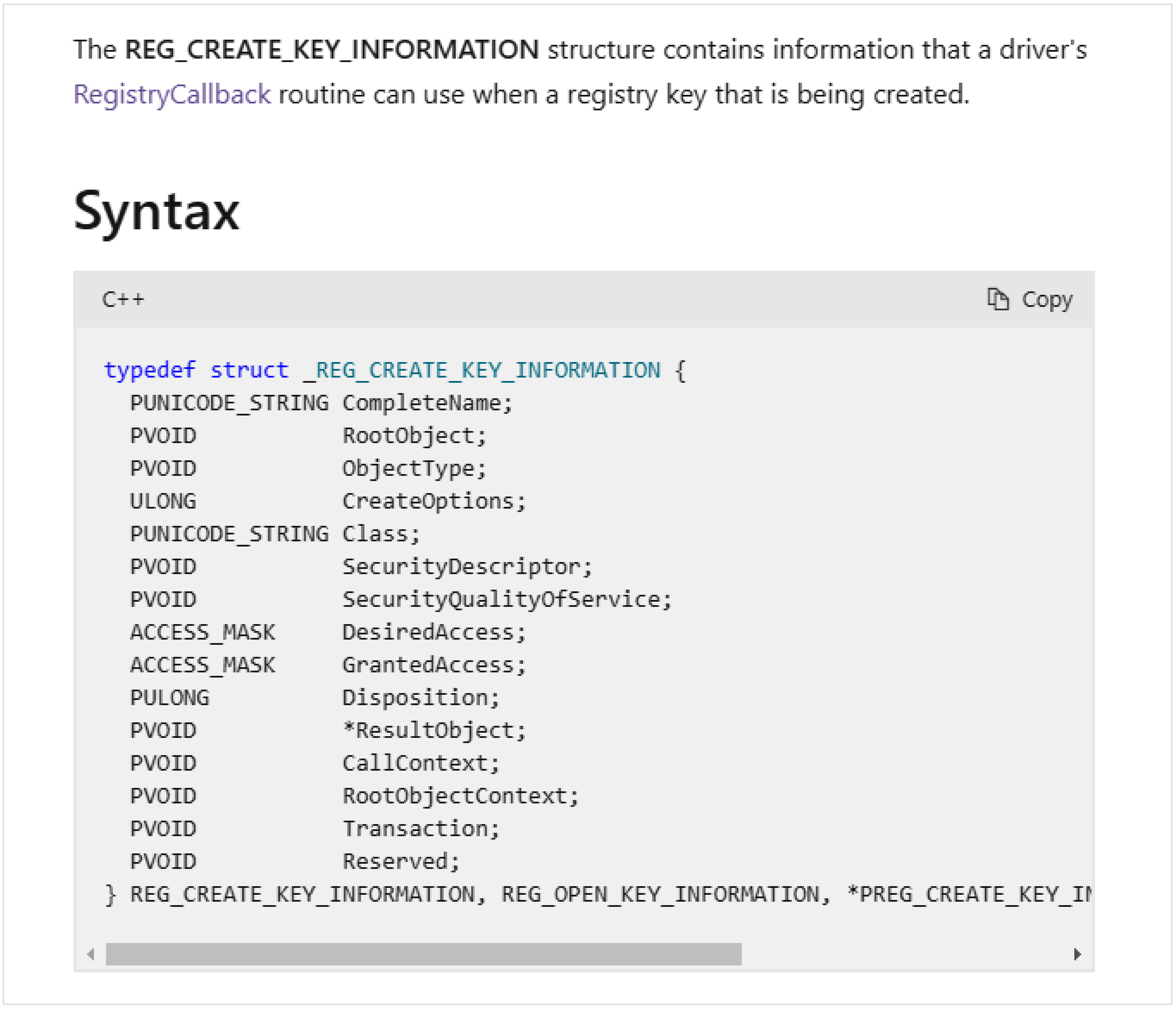
A REG_NOTIFY_CLASS-typed value that identifies the type of registry operation that is being performed and whether the *RegistryCallback* routine is being called before or after the registry operation is performed.

Taking a look at the `REG_NOTIFY_CLASS` constants defined by Microsoft, we can see that they correspond to many of the types of registry actions that we are familiar with – creating a key (`RegNtPreCreateKey`/`RegNtPostCreateKey`), setting the value of a key (`RegNtSetValueKey`/`RegNtPreSetValueKey`), and others.



REG_NOTIFY_CLASS value	Structure type
RegNtDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtSetValueKey	REG_SET_VALUE_KEY_INFORMATION

Each `REG_NOTIFY_CLASS` constant is associated with a specific structure that will be returned to the driver that registered a callback using that constant.


The **REG_CREATE_KEY_INFORMATION** structure contains information that a driver's RegistryCallback routine can use when a registry key that is being created.

Syntax

C++ Copy

```
typedef struct _REG_CREATE_KEY_INFORMATION {
  PUNICODE_STRING CompleteName;
  PVOID           RootObject;
  PVOID           ObjectType;
  ULONG           CreateOptions;
  PUNICODE_STRING Class;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
  ACCESS_MASK     DesiredAccess;
  ACCESS_MASK     GrantedAccess;
  PULONG          Disposition;
  PVOID           *ResultObject;
  PVOID           CallContext;
  PVOID           RootObjectContext;
  PVOID           Transaction;
  PVOID           Reserved;
} REG_CREATE_KEY_INFORMATION, REG_OPEN_KEY_INFORMATION, *PREG_CREATE_KEY_IN
```

`REG_CREATE_KEY_INFORMATION` structures are returned to drivers when their registered callbacks use the `RegNtPreCreateKey` or `RegNtPreCreateKeyEx` constants

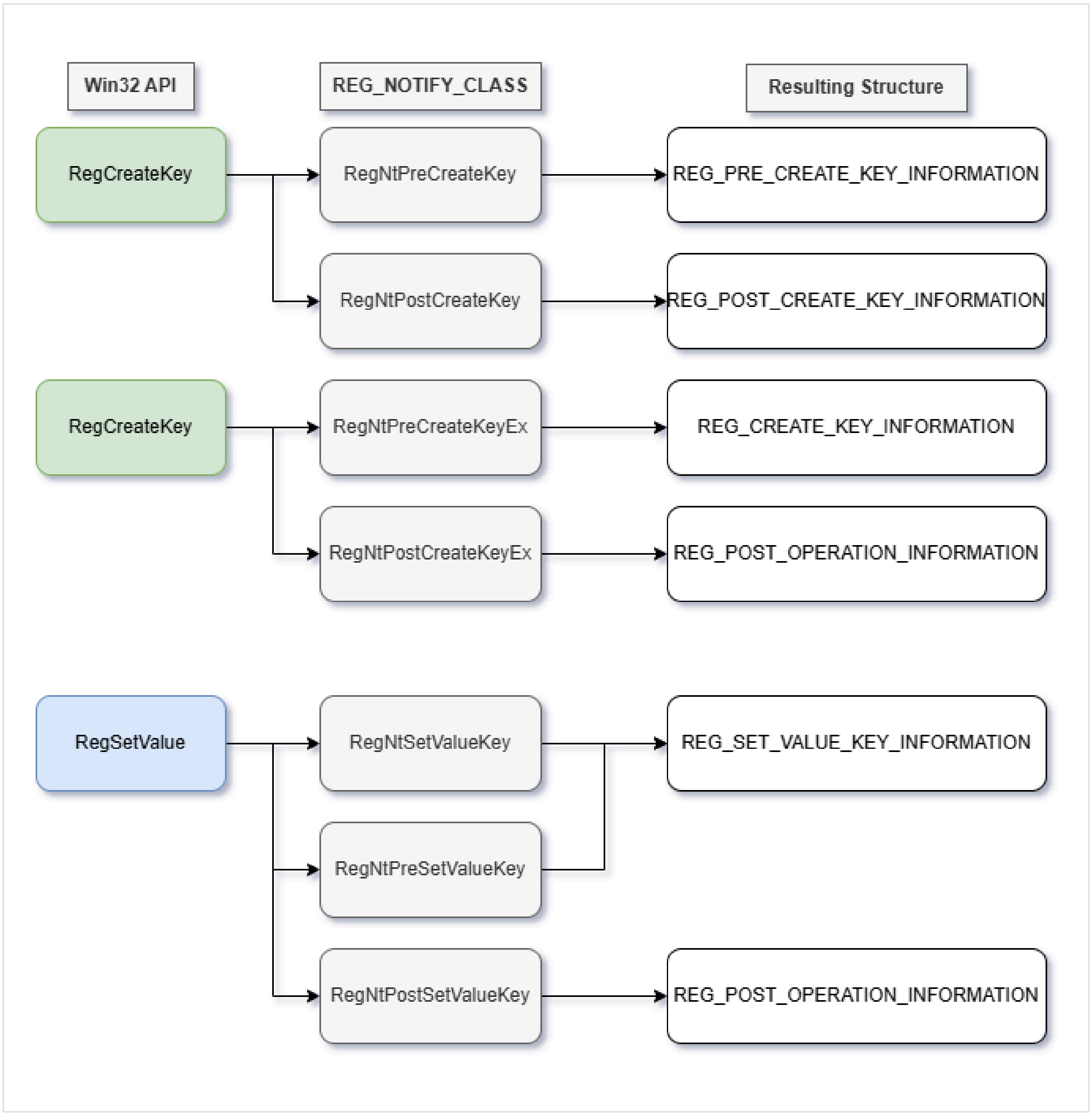
We have reached the first set of decision points that sensor developers must contend with when collecting registry telemetry via kernel callbacks—*Which operations do I care about enough to pay for in resources, bandwidth, and storage?*

Beyond pre-selecting which registry operations they want to observe, EDR sensor developers must implement the logic that decides what to do with an intercepted operation. Imagine attempting to identify a straightforward persistence technique like run key persistence via registry callback routines. As part of your `RegistryCallback` routine, you have to:

- Identify which type of operation an event represents by inspecting the `REG_NOTIFY_CLASS` constant value.

- If it is a `RegNtSetValueKey`, `RegPreNtSetValueKey`, or `RegPostNtSetValueKey` notification, identify which key's value is being written by inspecting the `REG_SET_VALUE_KEY_INFORMATION` or `REG_POST_OPERATION_INFORMATION` structure provided to the `RegistryCallback` routine.
- If that key is one of several "run keys", take some action.

If we wanted to observe registry key creation and registry value writes via kernel callbacks, we might conceptualize it using something like the below diagram. On the left are example Win32 API calls that will send events to any driver that has registered a registry callback routine using the `REG_NOTIFY_CLASS` constant from the middle column. The receiving driver would then be presented with the details defined by the struct in the third column.



Specific ways of interacting with the registry (represented by specific Win32 API calls) can be observed by registering a registry callback routine using the relevant `REG_NOTIFY_CLASS` constant

There may be additional callback logic implemented by the intercepting driver at this point, but the danger of adding overheard to an already hot code path requires as much brevity as possible. Remember, you are collecting registry telemetry as *overhead* to other collection priorities as well.

If we have to make additional function calls to query the registry object in question and inspect its characteristics, it's not all that difficult to introduce significant demands on the system if this logic gets too complex. On the other side of the distributed systems that we call EDR, logging and shipping everything without doing as much as possible on the sensor/agent side can balloon server and storage costs very quickly.

Reacting to the Status Quo

There are a lot of decisions to make when answering seemingly simple questions like "how do I know when a registry key has been created?".

There are multiple "sources" to identify and instrument collection from, each with their own set of configurable parameters far beyond what I have presented above. Your answer also has to fit into a much larger serialization, processing, shipping, and cloud pipeline with its own set of requirements and prior content expectations. Each decision you make creates a specific path to evasion based on how that change interacts with the system as a whole. Like Virilio said, "when you invent the ship, you also invent the shipwreck".

While there isn't necessarily a wrong answer to this question, there are more or less risky decisions from a visibility perspective. I could register a registry callback with the `RegNtPreCreateKey` or `RegNtPostCreateKey` `REG_NOTIFY_CLASS` constant and call it a day, but a discerning attacker would be right to question whether that is the only operation that could result in the creation of a new registry key.

Investigating Alternative Registry Operations

Let's assume that our target EDR sensor collects every single event generated by a registry callback registered with the `RegNtPreCreateKeyEx` or `RegNtPreSetValueKey` `REG_NOTIFY` constant. This should generate an event any time that a thread calls the `ZwCreateKey` or `ZwSetValueKey` native functions. Some form of this collection strategy seems to be the norm for current EDR products.

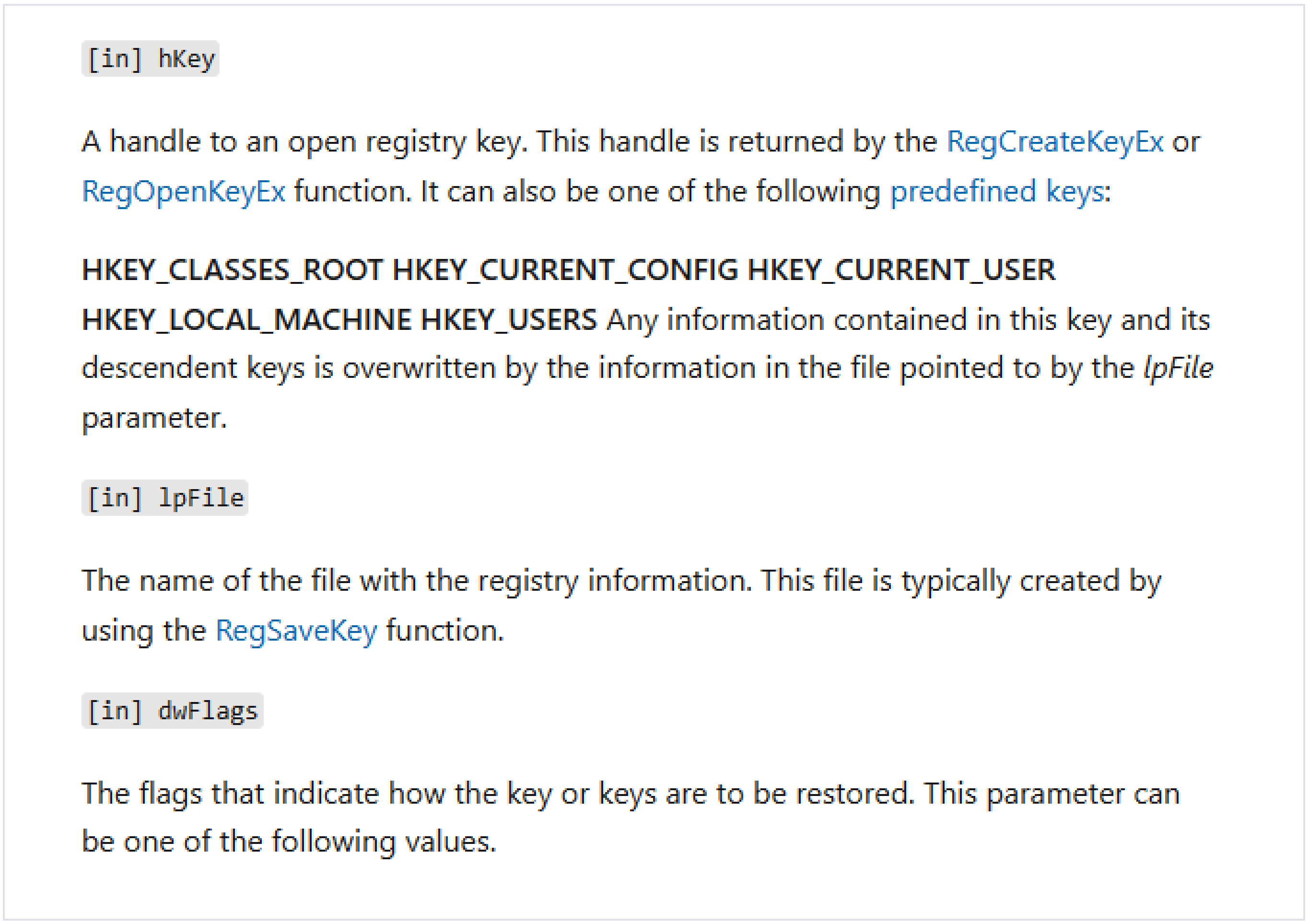
With this in mind, we can start looking at Microsoft docs for any additional functions that seem like they might change the contents of the registry or any `REG_NOTIFY` constants that suggest some other kind of poorly documented operation is possible. Our goal is to shift the collection requirements of defenders and force them to re-evaluate the tradeoffs they've already made peace with.

There are several functions in the winreg header file that might allow us to create and/or replace keys and values in the registry judging by name alone: `RegRestoreKey`, `RegCopyTree`, `RegLoadKey`, `RegReplaceKey`. This post is primarily about using `RegRestoreKey`, but it's important to think through the distinctions between these functions.

Win32 API	Description
RegRestoreKey	Takes a hive file on disk and overwrites the child sub-keys and values of a targeted key with the data in the supplied hive file.
RegCopyTree	Replaces one key (and its sub-keys/values) with another key and its sub-keys/values.
RegLoadKey	Similar to RegRestoreKey, but the targeted key must be a root key.
RegReplaceKey	Similar to RegRestoreKey, but instead of copying over the existing sub-key and its children, it replaces the file backing that sub-key on disk. This change does not take effect until the next reboot.

`RegRestoreKey` is an obvious choice for a first attempt because it allows us to target an arbitrary key and replace its child elements with content of our choosing (defined in a hive file). If we want to call this function, we will need to prepare a hive file that contains the key and value data that we want to use to overwrite some target key and its children. We can specify the sub-key we want to overwrite by supplying a handle `RegRestoreKey` via the `hKey` parameter.

```
LSTATUS RegRestoreKeyW(
        [in] HKEY    hKey,
        [in] LPCWSTR lpFile,
        [in] DWORD   dwFlags
);
```



`[in] hKey`

A handle to an open registry key. This handle is returned by the RegCreateKeyEx or RegOpenKeyEx function. It can also be one of the following predefined keys:

HKEY_CLASSES_ROOT HKEY_CURRENT_CONFIG HKEY_CURRENT_USER HKEY_LOCAL_MACHINE HKEY_USERS Any information contained in this key and its descendent keys is overwritten by the information in the file pointed to by the *lpFile* parameter.

`[in] lpFile`

The name of the file with the registry information. This file is typically created by using the RegSaveKey function.

`[in] dwFlags`

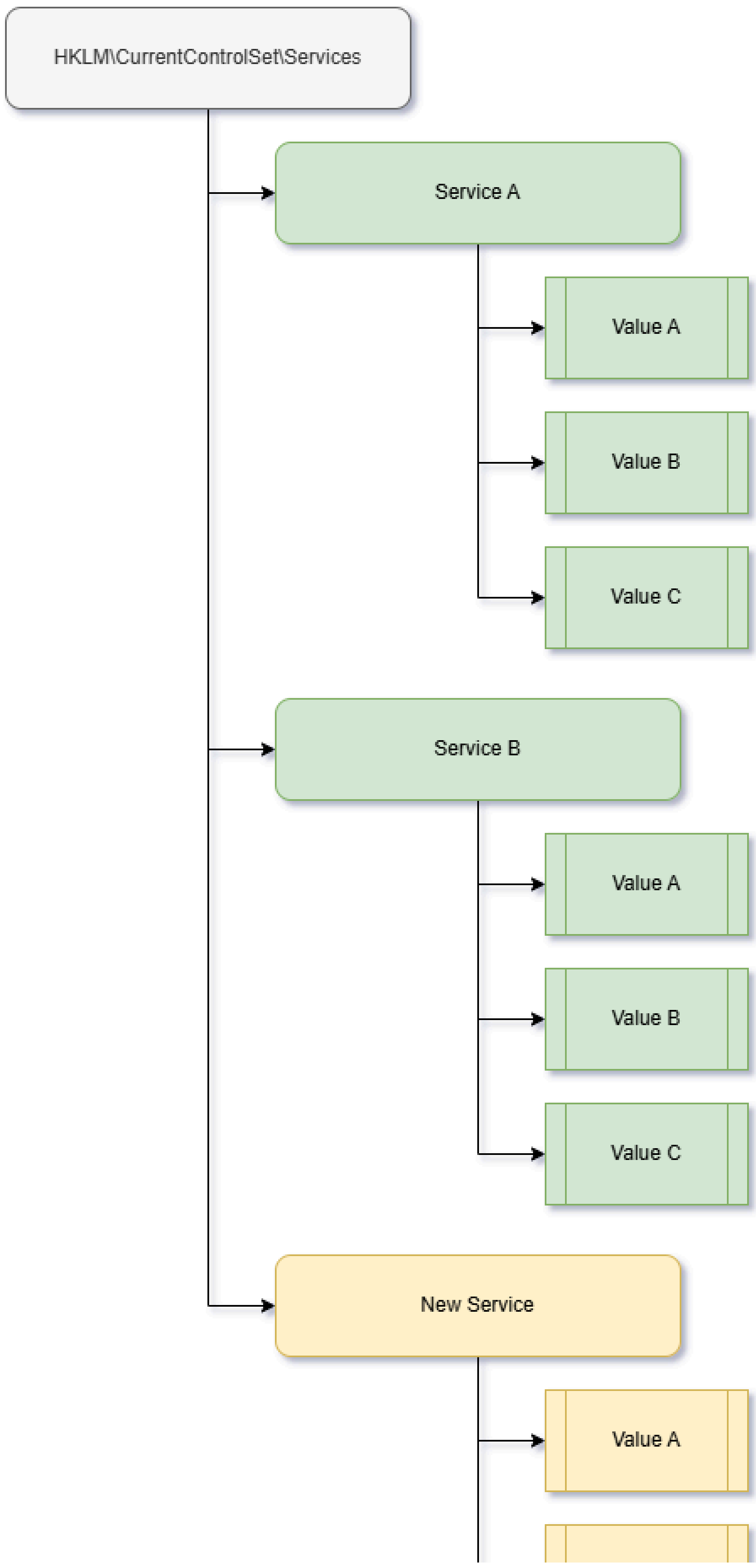
The flags that indicate how the key or keys are to be restored. This parameter can be one of the following values.

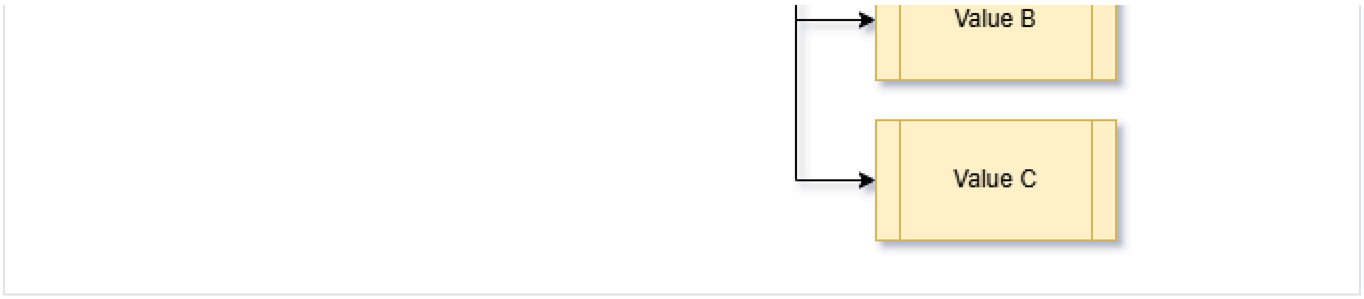
Creating Services with RegRestoreKey

While restoring a hive file from disk may be straightforward in isolation, there are some logistical hurdles involved with choosing a target sub-key and ensuring that your hive will not interrupt the normal behavior of the system when it is “restored”.

Picking a Target Key

Since we want to create a new service, we will need to create a new sub-key under `HKLM\CurrentControlSet\Services` and several registry values within that sub-key to define its configuration details. The resulting services sub-key should look something like this when we are done:


HKLM\CurrentControlSet\Services
Service A
Value A
Value B
Value C
Service B
Value A
Value B
Value C
New Service
Value A



Since `RegRestoreKey` takes a handle to the sub-key we want to overwrite, we actually have to pass a handle to the *parent* key of the sub-key we want to create. The sub-key has to already exist to get a handle to it. In our example, this means we must overwrite all of the service sub-keys starting from their common parent sub-key, `HKLM\CurrentControlSet\Services`.

In theory, you could create a dummy key just for the new service and target that for restore, but you would still generate a kernel callback event if there is a `RegNtPreCreateKeyEx` callback registered. For our purposes, we want to avoid creating a new sub-key in this registry path through `RegCreateKey` calls as we know that most EDR products are able to observe these operations.

Constructing a Hive File to Restore

The trick with creating our hive file is that its content will overwrite everything beneath that key — we are stuck restoring at the Services sub-key. So if we do not include all of its current child objects (keys and values) in our hive file, there is a potential to interrupt the normal behavior of the system. If our hive file only contains the key data for our new services key it will overwrite the main Services sub-key in place, deleting all previously installed services and leaving our new service's configuration values as its only child objects.

Instead, we have to preserve the current state of all of the keys and values under `HKLM\CurrentControlSet\Services` before adding a new key and values at the appropriate place in the hierarchy. This now modified snapshot of the services installed in the registry must be saved as a hive file so a handle to it can be passed as a parameter to `RegRestoreKey`.

You can create a hive file in the regedit GUI, using the native reg utility, or by calling one of the `RegSaveKeyEx`* Win32 API methods. However, we will need to figure out how to parse and modify the resulting hive file in some way.

We have several options here:

1. **Do everything manually (yuck)**
    1. Export a hive file on the target system from the Services key (`RegSaveKey`)
    2. Transfer to attacker-controlled system
    3. Load hive file on attacker-controlled system
    4. Add new service key and export
    5. Transfer modified hive file to target system
    6. Restore data from modified hive file on target system (`RegRestoreKey`)
2. **Export locally and modify exported file (no transfer)**
    1. Export a hive file from the Services key (`RegSaveKey`)
    2. Parse resulting hive file and add new key and values in place in the hive file
    3. Restore data from modified hive file (`RegRestoreKey`)
3. **Parse the Services key, create hive file via Offline Registry Library**
    1. Create an in-memory representation of a hive file using `ORCreateHive`
    2. Open a handle to `HKLM\CurrentControlSet\Services`, enumerate its sub-keys, and all of their respective values
    3. Copy data from each sub-key and value, creating their equivalent in the in-memory hive file via `ORCreateKey` and `ORSetValue`
    4. Create an additional key for our new service via `ORCreateKey`
    5. Create the values for our new service within that key via `ORSetValue`
    6. Write the offline registry hive to disk via `ORSaveHive`

The manual option is logistically difficult and annoying to carry out, so we will opt for some way of crafting the hive file programmatically on the target system. One of my teammates sent me a publicly available means of carrying out the second approach, which can be found here.

This POC follows the set of operations I laid out above for option #2 and saves a hive file for the `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` sub-key before adding a new value that references `C:\Users\Public\<RANDOM_FILENAME>.exe`. It then uses this modified hive file to call `RegRestoreKey` and overwrite the targeted key (HKCU run key) with the old data AND your new value.

We could extend this POC to implement more detailed hive file parsing and modification logic that accounts for services, but there are some small downsides to this approach. First, a call to `RegSaveKey` could potentially generate kernel callback notifications for a discerning EDR driver (assuming they have registered a callback with either of the `RegNtPostSaveKey` or `RegNtPreSaveKey` REG_NOTIFY constants). This wasn't a problem for the original author because they were specifically targeting Sysmon, which does not register this registry callback routine. However, many commercial EDR sensors will provide serialized events related to `RegSaveKey` calls.

Option #3 uses the Offline Registry Library to copy the already existing registry objects residing within the target key (the Services key) to an in-memory representation of a hive file. After building this hive file key by key and value by value, we can then insert our own sub-key and values before exporting it to a file. Because we are not calling `RegSaveKey`, we don't have to worry about that set of callback notifications. If implemented correctly, we should only have to worry about observation when we call `RegRestoreKey`.

Using the Offline Registry Library for Stealthier Hive File Creation

While the additional step of rebuilding our own representation of the Services sub-key and its children in memory may sound daunting, it's essentially an iterative copy/paste exercise using Win32 APIs from the [Offline Registry Library and the standard registry library.

1. **Enumerate all currently installed services on the system, saving their names and configuration details as you go:**
    1. Open a handle to the `HKLM\SYSTEM\CurrentControlSet\Services` key (`RegOpenKeyExW`).
    2. Enumerate all sub-keys of the `\Services` key (`RegEnumKeyEx`).
    3. Open a handle to each sub-key (`RegOpenKeyExW`).
    4. Enumerate each sub-key's values (`RegEnumValueW`).
    5. Query each value to get its name and current value (`RegQueryValueEx`).
2. **Prepare an in-memory representation of the `HKLM\SYSTEM\CurrentControlSet\Services` key and its descendant keys and values:**
    1. Create an empty offline registry hive (`ORCreateHive`).
    2. Write each saved service sub-key to the offline registry hive (`ORCreateKey`).
    3. Create each service key's relevant values in its offline equivalent (`ORSetValue`).
    4. Make desired changes to the offline hive. In our example, this involves adding a new key and set of values that represent our new service (`ORCreateKey` and `ORSetValue`).
3. **Write the offline registry hive to disk (`ORSaveHive`).**
4. **Reuse the original handle to the `HKLM\SYSTEM\CurrentControlSet\Services` key, along with the path to the newly written hive file created in memory, to overwrite the current content of that key and its descendants (`RegRestoreKeyW`).**
    1. In my testing, it was necessary to supply the `REG_FORCE_RESTORE` flag as part of the `dwFlags` parameter to `RegRestoreKeyW`.
    2. In order to make this call, you must be able to enable the `SE_RESTORE_NAME` and `SE_BACKUP_NAME` privileges.
    3. The `lpFile` parameter must be a valid path to a file on disk, but there is lots of room for creativity here.

There are obviously a few peculiarities surrounding where and how to target your registry restoration operations so that you don't overwrite registry data unnecessarily. Aside from that, we are just copying, modifying, and overwriting the content of the registry starting from a specific key. The way this approach interacts with common detection strategies for registry-based tradecraft is much more interesting than the approach I outlined above.

Extremely important: Please read this part

Please note that overwriting every single service key and its values based on iterative parsing and recreation is almost CERTAIN to lead to stability issues. This is likely due to the criticality of services to the driver loading and startup process during boot. DO NOT run the linked proof of concept on a system that you are not prepared to re-image or restore from a stable snapshot. I am releasing this tool in a "broken", but still demonstrative, state on purpose. If you want to use this in a non-POC context, you will need to make the modifications yourself. This WILL happen to your system if you overwrite the entirety of the `HKLM\SYSTEM\CurrentControlSet\Services\` registry key.

> This is almost certainly the result of supplying the REG_FORCE_RESTORE flag to RegRestoreKey. It is likely possible to make this work, but I will leave that work up to you. In general, I would suggest simply choosing a different target registry key to overwrite.

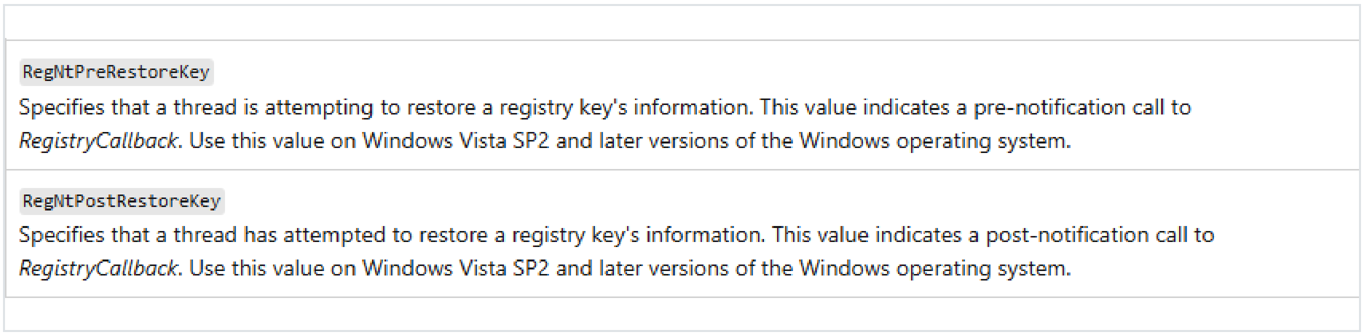
Quantifying Evasion

Let's step back and think about what we have accomplished:

When in possession of a token with the SE_RESTORE_NAME and SE_BACKUP_NAME privileges enabled, we can create or modify an arbitrary registry key without making calls to RegCreateKey, RegSetValue, or any of their lower level equivalents that generate commonly collected registry telemetry.

Registry Callback Routines

From the earlier discussion of registry callback routines, we know that EDR agents must make specific choices about which events are *valuable* enough to collect at scale. We also know that the "typical" collection strategy outlined for the sake of this example would not observe calls to RegRestoreKey because it generates an entirely different type of callback notification that isn't covered by callbacks registered with any of the RegNtPre/PostCreateKey* or RegNtPre/PostSetValueKey constants. Drivers would need to also register a callback with either RegNtPreRestoreKey or RegNtPostRestoreKey to observe calls to RegRestoreKey.


RegNtPreRestoreKey
Specifies that a thread is attempting to restore a registry key's information. This value indicates a pre-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.

RegNtPostRestoreKey
Specifies that a thread has attempted to restore a registry key's information. This value indicates a post-notification call to *RegistryCallback*. Use this value on Windows Vista SP2 and later versions of the Windows operating system.

Link

Even if a driver had registered callbacks related to saving registry hives, which are commonly used to combat credential theft tradecraft, my proof of concept never saves a copy of an existing key as a hive file. Instead, the offline registry library allows us to skip calls like [RegSaveKeyEx] by recreating the keys we wish to overwrite in an in-memory representation.

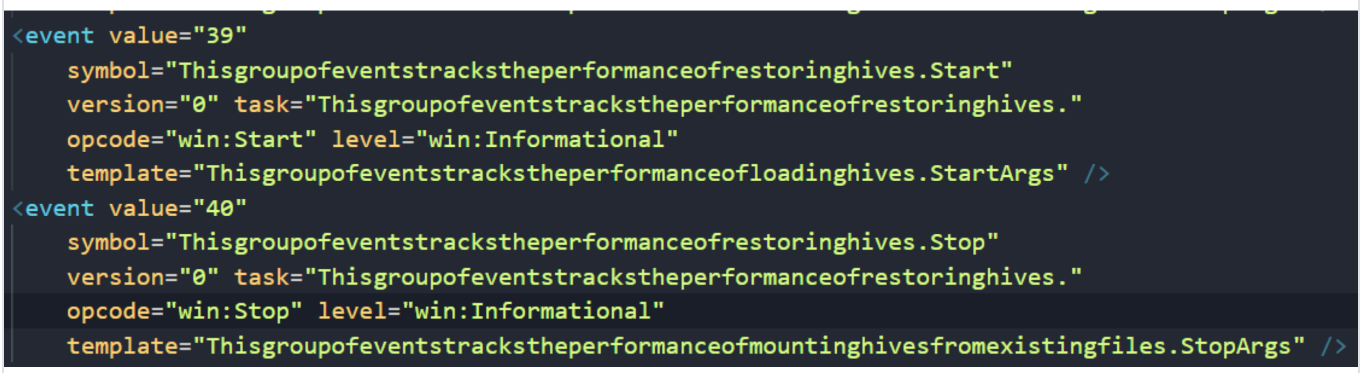
Callbacks registered with the above two REG_NOTIFY_CLASS constants are provided with a REG_RESTORE_KEY_INFORMATION struct that provides information about the restore operation itself. Of particular note is the Flags field that can communicate when the REG_FORCE_RESTORE flag was supplied, which was required for me to get my proof of concept to work reliably.

```
typedef struct _REG_RESTORE_KEY_INFORMATION {
  PVOID  Object;
  HANDLE FileHandle;
  ULONG  Flags;
  PVOID  CallContext;
  PVOID  ObjectContext;
  PVOID  Reserved;
} REG_RESTORE_KEY_INFORMATION, *PREG_RESTORE_KEY_INFORMATION;
```

ETW

There may well be additional ETW providers that generate telemetry related to registry restore operations, but I will focus on `Microsoft-Windows-Kernel-Registry` (`{70eb4f03-c1de-4f73-a051-33d13d5413bd}`) as it is the most common. The list of events I provided earlier in this post intentionally left out a few interesting opportunities for detection. In addition to the succinctly named events like `CreateKey` or `DeleteKey`, there is a list of keywords with extremely long names that suggest they are intended primarily for performance tracking (ex. `Thisgroupofeventstracksheperformanceof`...).

There are two events that appear to be directly related to registry restoration operations based on their names. The below shows the definition of these events in XML manifest for the Microsoft-Windows-Kernel-Registry ETW provider:



```
<event value="39"
    symbol="Thisgroupofeventstrackstheperformanceofrestoringhives.Start"
    version="0" task="Thisgroupofeventstrackstheperformanceofrestoringhives."
    opcode="win:Start" level="win:Informational"
    template="Thisgroupofeventstrackstheperformanceofloadinghives.StartArgs" />
<event value="40"
    symbol="Thisgroupofeventstrackstheperformanceofrestoringhives.Stop"
    version="0" task="Thisgroupofeventstrackstheperformanceofrestoringhives."
    opcode="win:Stop" level="win:Informational"
    template="Thisgroupofeventstrackstheperformanceofmountinghivesfromexistingfiles.StopArgs" />
```

`Thisgroupofeventstrackstheperformanceofrestoringhives.Start` and `Thisgroupofeventstrackstheperformanceofrestoringhives.Stop` declared

```
<template tid="Thisgroupofeventstrackstheperformanceofloadinghives.StartArgs">
    <data name="SourceFile" inType="win:UnicodeString" />
    <data name="Flags" inType="win:HexInt32" />
</template>
<template tid="Thisgroupofeventstrackstheperformanceofexportinghives.StartArgs">
    <data name="SourceKeyPath" inType="win:UnicodeString" />
</template>
```

Fields for the `Thisgroupofeventstrackstheperformanceofloadinghives.Start` event, which uses the same template as `…performanceofmountinghivesfromexistingfiles…` events

```
<template
    tid="Thisgroupofeventstrackstheperformanceofmountinghivesfromexistingfiles.StopArgs">
    <data name="StatusCode" inType="win:HexInt32" />
</template>
```

Fields for the `Thisgroupofeventstrackstheperformanceofloadinghives.Stop` event

While testing, I was able to confirm the lack of `CreateKey` or `SetValue` events. However, I was able to see several ETW events related to the hive file itself.

```
**Microsoft-Windows-Kernel-Registry/RegPerfTaskHiveRestore/Start**
ThreadID="16,816"
ProcessorNumber="1"
SourceFile="\Device\HarddiskVolume3\Users\mbarc\AppData\Local\Temp\restore_hive_runkey.h
Flags="8" // Flags == 0x00000008L == REG_FORCE_RESTORE

**Microsoft-Windows-Kernel-Registry/RegPerfTaskHiveRestore/Stop**
ThreadID="16,816"
ProcessorNumber="1"
DURATION_MSEC="6.154"
StatusCode="0"

**Microsoft-Windows-Kernel-
Registry/RegPerfTaskHiveMount/RegPerfOpHiveMountBaseFileMounted**
ThreadID="16,816"
ProcessorNumber="1"
HiveFilePath="\Device\HarddiskVolume3\Users\mbarc\AppData\Local\Temp\restore_hive_runkey.
FileSize="4,096"
```

How useful is this?

The extent to which any tool or approach is *evasive* is always going to rely entirely on the nature of the detection strategies it seeks to evade. The approach I've outlined above and demonstrated in the linked proof of concept code is only evasive insofar as EDR products do not collect telemetry related to restore operations. At time of writing, I have not encountered an EDR product that presents any restore-related events to an end user, nor have I seen an alert related to this behavior in my limited testing.

However, the raw telemetry needed to at least observe attempts to restore data from a registry hive does exist in both user mode and kernel mode. Whether or not those events are sufficient to develop a working detection strategy is an additional question, but visibility is entirely possible already. Its absence is a design decision, not a technical limitation. While I focused on a specific use case for registry interaction (service creation) in this post, the approach reawakens any tradecraft that involves creating, modifying, or deleting a registry key or value. We often consider registry-centric persistence techniques like run keys and services to be unworthy of specific attention because they are “solved problems”, but I’m hoping that this post is a reminder that what is old is always new again.

Reacting to Innovation

Realistically, the usefulness of this approach for evasion is a matter of how long it takes for EDR vendors to address it and the extent to which their solution is thorough.

An ETW-centric strategy would likely require additional work from Microsoft, since the performance tracking events mentioned above represent the absolute bare minimum to identify a restoration operation. While I was able to see performance events from Microsoft-Windows-Kernel-Registry when I ran my PoC code, these events exclusively describe the hive file that is being used for the restore operation. They do not include information about which key is being targeted.

On the other hand, the registry callback routine telemetry is a bit more verbose and likely lends itself more to a thorough detection strategy, as defined by the `REG_RESTORE_KEY_INFORMATION` struct. As long as you can follow the pointer to the registry key object that was targeted and query additional information about it, you can write a detection strategy that considers the *purpose* of the restore operation.

Regardless of what Windows provides, end users of EDR products have very few options to address this gap if their chosen agent doesn’t already observe and report information about registry restore operations. While there are tons of publicly available tools that help you configure and start trace sessions to collect events from ETW providers on your own, attempting to instrument this at an environment-wide level is the equivalent of building your own EDR. This isn’t so much a data volume problem—it’s a data selection problem.

Vids & Dids

PoC code is available here.

A very important disclaimer about my example code!

The linked PoC is INTENTIONALLY BROKEN! While it can successfully overwrite all of the services keys and their values without generating traditional key creation and value write telemetry, this is a wildly unstable and unpredictable action to take and CONSISTENTLY leads to a blue screen. If you would like to test it for yourself I would suggest either:

- restoring to a dummy key you create for testing so that you don't overwrite the real services key (no arguments)
- running the tool unmodified on a system that you KNOW you will be able to re-image or restore from snapshot (supply the path to the real services key)

The tool will attempt to restore the services data in the hive file to `HKLM\\Software\\Test` if no arguments are supplied. If you would like to attempt to overwrite the real services key, you can supply that path as a string.

Restoring Run Keys

While I have chosen to release a PoC for this approach specific to a use case that is NOT useful in an operational context due to instability, the below video shows the addition of a registry run key value via a registry restore operation. This use case has not led to any observable instability on the system because system critical drivers do not rely on the integrity of services and their configuration values. It is likely possible to do the same with the services use case, but you will have to figure out which values are causing instability problems and how to account for them.