Inside Windows' Default Browser Protection

binary.ninja/2025/03/25/default-browser-upcd.html

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice]
"ProgID"="FirefoxURL-308046B0AF4A39CB"
"Hash"="itSkw9xbEI8="
```

Binary Ninja Blog

- Xusheng Li
- 2025-03-25
- reversing

The battle for the default browser on Windows has always been heated. You might have heard of how Microsoft leveraged its UCPD (User Choice Protection Driver) to prevent third-party browsers from setting themselves as the default one. However, in this post, I will show my journey into uncovering how various browsers try to bypass the restriction, and how UCPD gets updated to defeat their attempts.

Note: this is an extended version of my lightning talk at RE//verse. Please also check out the video and slides.

Background

The default browser is saved in the following registry keys:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice
HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\https\UserChoice
```

Once upon a time, setting the default browser was as easy as setting the value of these keys – which is relatively straightforward. However, this was often abused by certain vendors to hijack the default browser to their own, without user consent or even interactions.

To address this issue, Windows introduced a `Hash` sub-key which contains a hash value of the selected default browser. The default browser settings are only respected if the hash is correct. The hash algorithm is proprietary, and can only be calculated correctly if you use the Windows's

Settings dialog to set it. It also includes entropy from the current machine so it cannot be pre-computed.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice]
"ProgID"="FirefoxURL-308046B0AF4A39CB"
"Hash"="itSkw9xbEI8="
```

And of course, the secret for this hash would not last very long. In 2017, Christoph Kolbicz reverse-engineered the hash algorithm and deployed it in his SetUserFTA tool, a command line utility that can set file type associations or default browser. In 2021, Mozilla similarly enabled Firefox to set itself as the default browser directly.

Microsoft responded to the “cracking” of its hash algorithm with the introduction of the UCPD driver in March 2024. UCPD stands for User Choice Protection Driver, and it is a filter driver that protects the registry keys that store the default browser settings (along with similar things, e.g., the default PDF reader).

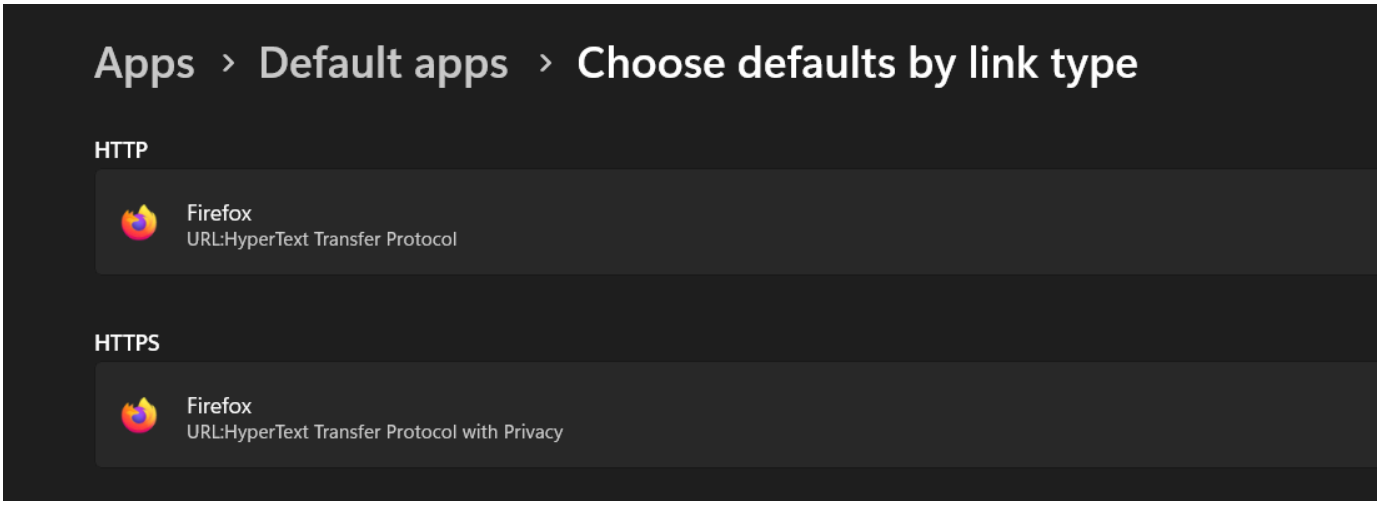
Gunnar Haslinger first reported the discovery of the sneaky UCPD driver and analyzed its functionality. In short, it uses the standard Window registry filtering mechanism – CmRegisterCallbackEx – to register a callback that monitors the registry operations. If a protected key is being operated on (e.g., edited, created, etc), it only allows it if the requesting process is trusted. The criteria for a trusted process are:

1. The executable is signed by Microsoft
2. The executable is NOT in a list of utility programs, including:
    - `reg.exe`
    - `rundll32.exe`
    - `powershell.exe`
    - `regedit.exe`
    - `wscript.exe`
    - `cscript.exe`
    - …etc

```
140004a38    BOOLEAN IsTrustedProcess(HANDLE pid, int16_t* arg2)

140004a49        void var_48
140004a49        int64_t rax_1 = __security_cookie ^ &var_48
140004a58        UNICODE_STRING process_name
140004a58        BOOLEAN ret
140004a58
140004a58        if (pid != 4)
140004a62            process_name.Length = 0
140004a62            process_name.MaximumLength = 0
140004a62            process_name.Buffer = 0
140004a67            ret = 0
140004a69            GetProcessImageNameEx(pid, &process_name)
140004a69
140004a74        BOOLEAN result
140004a74
140004a74        if (pid == 4 || process_name.Buffer == 0)
140004aba            result = 1
140004a74        else
140004a82            if (IsInDenyList(&process_name) == 0)
140004a93                ret = 0
140004a93
140004a9b                if (IsMicrosoftSignedFile(&process_name, arg2) != 0)
140004a9b                    ret = 1
140004a9b
140004a9e            WCHAR* Buffer = process_name.Buffer
140004a9e
140004aa6            if (Buffer != 0)
140004aaa                ExFreePoolWithTag(P: Buffer, Tag: 0)
140004aaa
140004ab6            result = ret
140004ab6
140004ac7        __security_check_cookie(rax_1 ^ &var_48)
140004ad6        return result
```

Interestingly, I was lucky enough to catch a glimpse of the driver before Microsoft removed its symbol file from the PDB server. Based on the function names and the implementation, it is easy to see its intention to stop any third-party from modifying the default browser registry keys and enforcing that the only way to set the default browser is through the Settings app:



If you think about this carefully, you will realize that UCPD still allows the Edge browser to set itself as the default browser (it is signed by Microsoft and it is not in the utility list). However, per my tests, Edge is NOT utilizing this to force itself to be the default browser – when you instruct Edge to set itself as the default browser, it launches the Windows Settings app in which you will need to select it as the default by yourself – which is the officially recommended way.

Are you already surprised that Microsoft even bothers to create a driver to protect the default browser? Well, bear with me since this is only the start of the story!

Injection is Not Allowed

As a hobbyist security researcher, I am curious to see if there is a way to bypass the UCPD. I did not have the time to dig into it until late October. At that time, my Windows PC was already updated to the build 24H2. And I was quickly welcomed by a surprise – the new UCPD driver contained the names of several well-known vendors!

```
1c000d1b8  wchar16 const (* data_1c000d1b8)[0xa] = data_1c000a2c8 {u"\safemon\"}
1c000d1c0  wchar16 const (* data_1c000d1c0)[0xb] = data_1c000a2e0 {u"\kingsoft\"}
1c000d1c8  wchar16 const (* data_1c000d1c8)[0x8] = data_1c000a2f8 {u"\opera\"}
1c000d1d0  wchar16 const (* data_1c000d1d0)[0xc] = data_1c000a308 {u"\msedge.exe"}
1c000d1d8  wchar16 const (* data_1c000d1d8)[0xe] = data_1c000a320 {u"\explorer.exe"}
1c000d1e0  wchar16 const (* data_1c000d1e0)[0xd] = data_1c000a340 {u"explorer.exe"}
1c000d1e8  wchar16 const (* data_1c000d1e8)[0xb] = data_1c000a360 {u"msedge.exe"}
1c000d1f0  wchar16 const (* data_1c000d1f0)[0x11] = data_1c000a378 {u"kwsprotect64.exe"}
1c000d1f8  wchar16 const (* data_1c000d1f8)[0xd] = data_1c000a3a0 {u"kxescore.exe"}
1c000d200  wchar16 const (* data_1c000d200)[0xc] = data_1c000a3c0 {u"360Tray.exe"}
1c000d208  void* data_1c000d208 = 0x1c000a3d8
1c000d210  wchar16 const (* data_1c000d210)[0x2b] = data_1c000a400 {u"Beijing Kingsoft Security softwa…"}
1c000d218  wchar16 const (* data_1c000d218)[0x22] = data_1c000a460 {u"Beijing Qihu Technology Co., Ltd…"}
```

What is going on?

To begin with, I noticed that the new driver uses PsSetCreateProcessNotifyRoutineEx to monitor all the new process creation. For each created process, it first checks if its image file name contains any of the following sub-strings:

- `\safemon\`
- `\kingsoft\`
- `\opera\`
- `\msedge.exe`
- `\explorer.exe`

If so, for each such process, it classifies the processes into several different categories and assigns an enum value to it. The categories are:

- Type 0x4: the process is `explorer.exe` or `msedge.exe`
- Type 0x8:
  - The process is `kwsprotect64.exe`, `kxescore.exe`, or `360Tray.exe`, and
  - The executable is signed by one of the following:
    - `Beijing Kingsoft Security software Co.,Ltd`
    - `Beijing Qihu Technology Co., Ltd.`
    - `Zhuhai Juntian Electronic Technology Co., Ltd.`
- Type `0x10`: the executable is signed by `Opera Norway AS`, likely the Opera browser

The process -> category mapping information is saved into an AVL table. If you are not familiar with it, you can think of it as a `std::map` equivalent. You can find the main processing logic of the process notify routine in the below screenshot. Within it, the `element.type` is the process category mentioned above.

```
1c00029c4       if (check_process_path_folder(ImageFileName) != 0)
1c00029ce           uint128_t* rax_4 = get_exe_from_path(arg2->ImageFileName)
1c00029ce
1c00029d9           if (rax_4 == 0)
1c00029db               rbx = -0x3fffff45
1c00029e0               goto label_1c0002aa2
1c00029e0
1c00029ec           RtlInitUnicodeString(DestinationString: &element.field_10,
1c00029ec               SourceString: rax_4)
1c00029f8           int16_t* ImageFileName_1 = arg2->ImageFileName
1c0002a04           bool valid_sig = false
1c0002a07           GetVendorName(ImageFileName_1, &element.vendor_info, &valid_sig)
1c0002a27           element.type =
1c0002a27               check_vendor(&element.field_10, &element.vendor_info, valid_sig, arg2)
1c0002a2d           element.field_0 = 1
1c0002a35           element.pid = ProcessId
1c0002a39           ExAcquireFastMutex(FastMutex: &mutex)
1c0002a39
1c0002a69           if (RtlInsertElementGenericTableAvl(Table: &table, Buffer: &element,
1c0002a69                   BufferSize: 0x38, NewElement: nullptr) == 0)
1c0002a6b               ExReleaseFastMutex(FastMutex: &mutex)
1c0002a7c               rbx = -0x3fffff45
1c0002a81               ExFreePoolWithTag(P: rax_4, Tag: 0)
1c0002a8d               goto label_1c0002aa2
```

And where is the saved mapping information used? Well, it is used in the object callbacks registered with ObRegisterCallbacks.

There are three object callbacks, one for each of `PsProcessType`, `PsThreadType`, and `ExDesktopObjectType`. The callbacks are all `PreOperation` so that they can examine the requests and act accordingly:

```
1c0001da8    NTSTATUS setup_object_callbacks()

1c0001dc1        void var_e8
1c0001dc1        int64_t rax_1 = __security_cookie ^ &var_e8
1c0001dd1        NTSTATUS result
1c0001dd1
1c0001dd1        if (RegistrationHandle != 0)
1c0001dd3            result = 0xc0000510
1c0001dd1        else if ((enabled_feature & 0x180) != 0)
1c0001dfc            OB_OPERATION_REGISTRATION registration[0x3]
1c0001dfc            registration[0].ObjectType = PsProcessType
1c0001e05            registration[0].Operations.q = 0
1c0001e09            registration[1].Operations.q = 0
1c0001e14            registration[0].PreOperation = PreOperation
1c0001e25            registration[1].ObjectType = PsThreadType
1c0001e2f            registration[2].Operations.q = 0
1c0001e3a            registration[1].PreOperation = PreOperation
1c0001e49            registration[2].ObjectType = ExDesktopObjectType
1c0001e54            registration[2].PreOperation = DesktopObjectPreOp
1c0001e5c            registration[0].Operations = 1
1c0001e63            registration[1].Operations = 1
1c0001e6d            registration[2].Operations = 1
1c0001e77            void var_28
1c0001e77            void* var_c0_1 = &var_28
1c0001e7b            registration[0].PostOperation = 0
1c0001e7f            registration[1].PostOperation = 0
1c0001e83            registration[2].PostOperation = 0
1c0001e87            int64_t var_c8 = 0x140000
1c0001e8f            result = printf-ish(&var_c8, u"%lu.%lu", 0x5e0e2)
1c0001e8f
1c0001e96            if (result s>= STATUS_SUCCESS)
1c0001e98                int128_t zmm0_1 = var_c8.o
1c0001ea0                struct _OB_CALLBACK_REGISTRATION CallbackRegistration
1c0001ea0                CallbackRegistration.Version = 0x100
1c0001ea0                CallbackRegistration.OperationRegistrationCount = 3
1c0001eaf                CallbackRegistration.RegistrationContext = 0
1c0001eb7                CallbackRegistration.OperationRegistration = &registration
1c0001ebb                CallbackRegistration.Altitude.Length = zmm0_1.w
1c0001ebb                CallbackRegistration.Altitude.MaximumLength = zmm0_1:2.w
1c0001ebb                CallbackRegistration.Altitude.Buffer = zmm0_1:8.q
1c0001ec0                result = ObRegisterCallbacks(&CallbackRegistration, &RegistrationHandle)
1c0001de7        else
1c0001de9            result = STATUS_NOT_SUPPORTED
1c0001de9
1c0001ed3        stack_check(rax_1 ^ &var_e8)
1c0001ee8        return result
```

The two PreOperation callbacks for the `PsProcessType` and `PsThreadType` are the same. Within it, it first gets the requesting process that is trying to acquire a handle, and the target process (or the process the target thread belongs to) whose handle is being acquired. It then uses the PID to look up the process category information from the AVL table it maintains. After that, it checks for the following conditions:

- The target process has a category type `0x4` (`explorer.exe` or `msedge.exe`)

- The requesting process has a category type `0x8` (`360Tray.exe/kxescore.exe/kwsprotect64.exe`)

If the condition is met, then it removes the following access rights from the requested rights:

- For `PsProcessType`, the process access rights `0x28` (`PROCESS_VM_OPERATION` | `PROCESS_VM_WRITE`) are removed
- For `PsThreadType`, the thread access right `0x10` (`THREAD_SET_CONTEXT`) is removed

```
if (target_process != 0)
    if (((target_process->type).b & ExplorerOrMsedge) != 0)
        PEPROCESS Process = IoGetCurrentProcess()

        if (Process != 0 && OperationInformation->Object != Process)
            HANDLE current_process_pid = PsGetProcessId(Process)

            if (current_process_pid != 0)
                struct TableBuffer* current_process =
                    table_lookup(current_process_pid)

                if (current_process != 0)
                    if (((current_process->type).b & QihuOrKinsoft)
                            != 0)
                        union _OB_PRE_OPERATION_PARAMETERS*
                            Parameters = OperationInformation->Parameters
                        int32_t rbx_1 = *Parameters
                        enum ThreadAccessRights thread_rights =
                            THREAD_SET_CONTEXT
                        enum ProcessAccessRights process_rights =
                            thread_rights

                        if (ObjectType == PsProcessType)
                            process_rights =
                                PROCESS_VM_OPERATION | PROCESS_VM_WRITE

                        int32_t r9_3 = not.d(process_rights) & rbx_1
                        *Parameters = r9_3
                        int32_t r10_1 = process_rights & rbx_1
```

Now it should be quite easy to see the intention – it is preventing 360Tray.exe/kxescore.exe/kwsprotect64.exe from injecting code into the explorer.exe/msedge.exe! Why would UCPD bother doing that? The only explanation is they are trying to bypass UCPD by injecting code into explorer.exe/msedge.exe since the two can modify the registry key for the default browser. And Microsoft did not like the idea, so it tightened its protection by directly banning the offenders!

The remaining callback for `ExDesktopObjectType` checks if the current process has a category 0x10, i.e., the Opera browser. The code is simple – it just removes the access right 0x20 (by `&= 0xffffffdf`):

```
struct TableBuffer* process = table_lookup_current_process()

if (process != 0)
    if (((process->type).b & Opera) != 0)
        union _OB_PRE_OPERATION_PARAMETERS* Parameters =
            OperationInformation->Parameters
        *Parameters &= 0xffffffdf
```

But it took me some time to figure out what it meant. To start with, we can see 0x20 corresponds to `DESKTOP_JOURNALPLAYBACK` in the official docs on "Desktop Security and Access Rights". The docs say the access right is "required to perform journal playback on a desktop", which I had no idea about. I found little information about it after Googling, so I asked ChatGPT got some clue – it is part of the Windows UI Automation and meant to be used to playback previously recorded user keyboard and mouse inputs. ChatGPT even helpful pointed out the security considerations related to it – that it can be abused for bogus UI interactions.

Chances are the Opera browser is playing some UI tricks to set the default browser, e.g., by interacting with the Settings app directly. UCPD has no mercy for it.

UCPD Manager

You might be wondering why I am so sure about my conclusion. As I dig deeper into this, I found the handle protection is only enabled when a global flag has the bit 0x100 set – otherwise, it will do nothing:

```
1c0001a60    int64_t PreOperation2(void* RegistrationContext, struct _OB_PRE_OPERATION_I

1c0001a8a        void var_168
1c0001a8a        int64_t rax_1 = __security_cookie ^ &var_168
1c0001a8a
1c0001aaa        if ((enabled_feature & 0x100) != 0 && OperationInformation != 0)
1c0001ab0            void* Object = OperationInformation->Object
1c0001ab0
```

This flag value is read from the following registry key:

`HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Srevices\UCPD\FeatureV2`

Meanwhile, there is a `UCPDMgr.exe` (UCPD Manager) in the system directory, whose `SetUCPDStatus` can set the value of this registry key:

```
14001d570    void SetUCPDStatus()

14001d584        uint32_t lpData_3
14001d584
14001d584    >   if (wil::details::FeatureImp...ts_Feature_UCPD>::__private_IsEnabled(this:
14001d584            &wil::Feature<struct __Wi...s_Feature_UCPD>::GetImpl::impl::~impl) == 0) ...
14001d584        else
14001d58f            SetV1Status(1)
14001d5aa            lpData_3 = zx.d(
14001d5aa                wil::details::FeatureImp..._Feature_UCPDV2>::__private_IsEnabled(this:
14001d5aa                &wil::Feature<struct __Wi...Feature_UCPDV2>::GetImpl::impl::~impl)) * 2
14001d5aa
14001d5b3            if (wil::details::FeatureImp...ure_UCPD_PRONG1>::__private_IsEnabled(this:
14001d5b3                    &wil::Feature<struct __Wi...re_UCPD_PRONG1>::GetImpl::impl::~impl) != 0)
14001d5b5                lpData_3 |= 4
14001d5b5
14001d5c6            if (wil::details::FeatureImp...ure_UCPD_PRONG2>::__private_IsEnabled(this:
14001d5c6                    &wil::Feature<struct __Wi...re_UCPD_PRONG2>::GetImpl::impl::~impl) != 0)
14001d5c8                lpData_3 |= 0x20
14001d5c8
14001d5d9            if (wil::details::FeatureImp...re_UCPD_TASKBAR>::__private_IsEnabled(this:
14001d5d9                    &wil::Feature<struct __Wi...e_UCPD_TASKBAR>::GetImpl::impl::~impl) != 0)
14001d5db                lpData_3 |= 0x10
14001d5db
14001d5ec            if (wil::details::FeatureImp...eature_UCPD_WSB>::__private_IsEnabled(this:
14001d5ec                    &wil::Feature<struct __Wi...ature_UCPD_WSB>::GetImpl::impl::~impl) != 0)
14001d5ee                lpData_3 |= 8
14001d5ee
14001d5ff            if (wil::details::FeatureImp...D_ANTIINJECTION>::__private_IsEnabled(this:
14001d5ff                    &wil::Feature<struct __Wi..._ANTIINJECTION>::GetImpl::impl::~impl) != 0)
14001d601                lpData_3 |= 0x100
14001d601
14001d613            if (wil::details::FeatureImp...re_UCPD_ANTIUIA>::__private_IsEnabled(this:
14001d613                    &wil::Feature<struct __Wi...e_UCPD_ANTIUIA>::GetImpl::impl::~impl) != 0)
14001d615                lpData_3 |= 0x80
```

Apparently the bit `0x100` is set when `wil::Feature<struct __WilFeatureTraits_Feature_UCPD_ANTIINJECTION>::GetImpl::impl::~impl` is enabled. And the `ANTIINJECTION` (anti-injection) in the name is a solid confirmation!

Another value that caught my eye is `0x80`, which goes by the name `UCPC_ANTIUIA`. I figured the `ANTIUIA` means `anti-UI attack`, and it reminds me of the desktop object callback above. I checked the start of the desktop callback function and all dots are connected:

```
int64_t DesktopObjectPreOpInternal(void* RegistrationContext,
  struct _OB_PRE_OPERATION_INFORMATION* OperationInformation)

    void var_e8
    int64_t rax_1 = __security_cookie ^ &var_e8

    if ((enabled_feature).b s< 0
            && ((*OperationInformation->Parameters).b & 0x20) != 0)
        struct TableBuffer* process = table_lookup_current_process()
```

The check `if ((enabled_feature).b s< 0` is indeed checking whether the bit 0x80 is set – since it takes the low byte from `enabled_feature` and checks if it is negative, which is equivalent to checking if the highest bit is set.

There are also several other interesting strings in it:

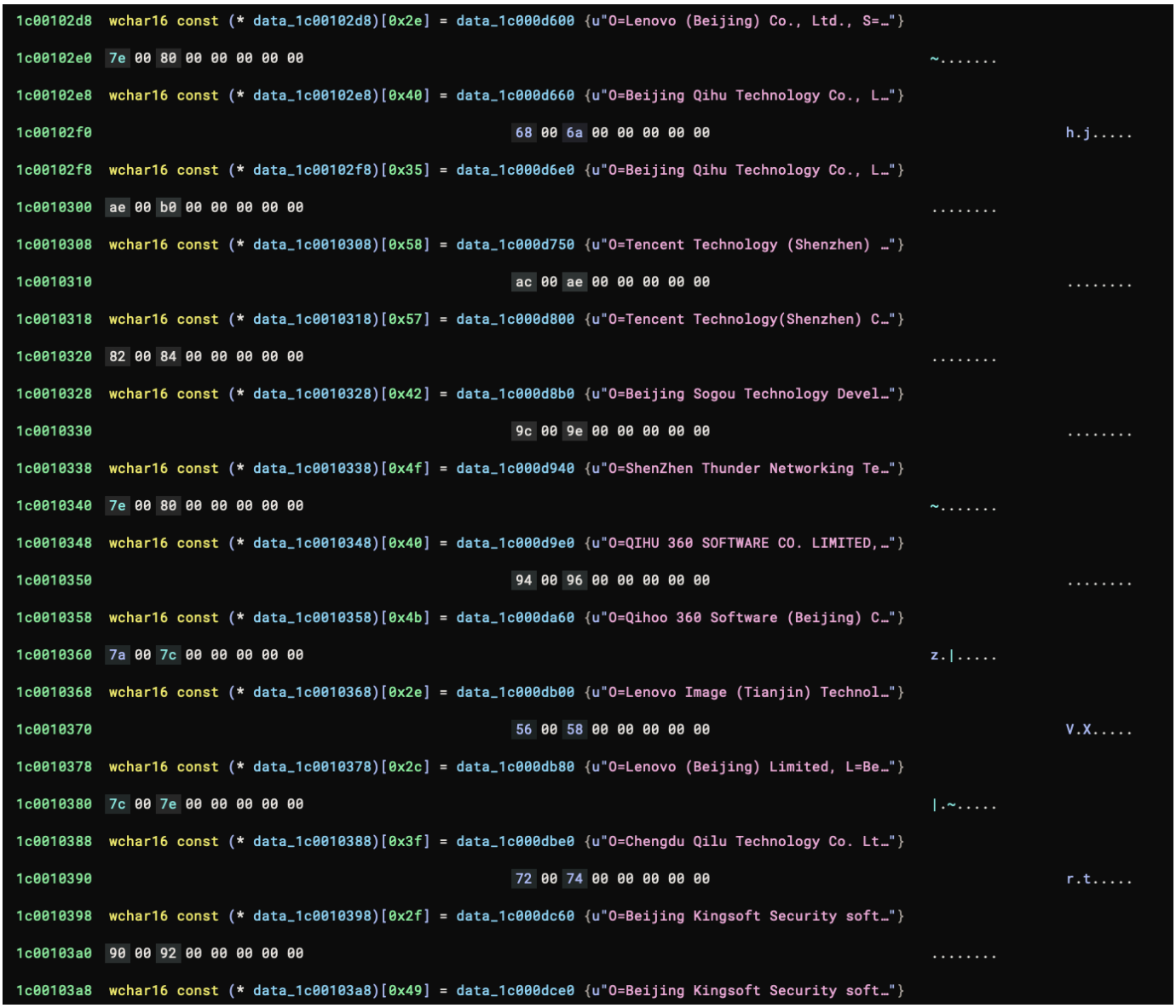
- `0x200` – `UCPD_NEWDENYLIST`, which enables a new deny list for registry keys
- `0x800` – `UCPD_RENAME_ATTACK`, which protects against an attack that renames certain keys
- `0x8` – `UCPD_WSB`, which protects the Windows search bar
- `0x10` – `UCPD_TASKBAR`, which protects the taskbar

For the sake of time, I did not dig into each of these. If you are interested, please feel free to check it out by yourself!

Another interesting bit is that when the handle protection takes action to limit the access rights, an event is generated in event tracing. From the content of the event, we can see that the UCPD driver I looked at is version `3.1.0.0`. Coincidentally, the earliest UCPC driver I have, i.e., the one still has the symbol, is version `2.1.0.1`. So I must have missed the very first version of it!

Stacktrace Must be Checked

While preparing this blog, the Windows March 2025 update dropped and it comes with UCPD 4.0. This time, some 16 vendors are added to a new block list:



It turns out that UCPD is getting tougher again. To start with, the list is used when the bit `0x400` is set in the feature DWORD. And in the new UCPDMgr.exe, it is associated with `UCPD_BACKTRACE`. And yes, UCPD is now examining the stack trace to determine if a registry operation should be allowed!

In short, when its registry callback sees SetValueKey/DeleteKey/RenameKey is called on a protected key, in addition to the existing `IsTrustedProcess` check, now checks whether the requesting process is `SystemSettings.exe` (the official app to set the default browser). And if so, it examines the stack trace with `RtlWalkFrameChain`. If any of the frames contains a module signed by a blocked vendor, then the operation is rejected.

```
1c0007a44    void** check_blacklist_2(int32_t operation_id, int32_t arg2, int16_t* arg3)

1c0007a71        void var_298
1c0007a71        int64_t rax_1 = __security_cookie ^ &var_298
1c0007a94        void var_1f0
1c0007a94        sub_1c0009b40(&var_1f0, 0, 0xd8)
1c0007a9f        char var_268 = 1
1c0007ab3        int32_t rdi = 0
1c0007ab5        void** result =
1c0007ab5            ExAllocatePoolWithTag(PoolType: 0x600, NumberOfBytes: 0x320, Tag: 'StPT')
1c0007ab5
1c0007ac7        if (result == 0)
1c0007fa6            result.b = 1
1c0007ac7        else
1c0007ad7            uint32_t rax_2 = RtlWalkFrameChain(Callers: result, Count: 0x64, Flags: 1)
1c0007ae3            uint32_t r12_1 = rax_2
1c0007aef            void* rax_5
1c0007aef
1c0007aef            if (rax_2 - 1 u<= 0x62)
1c0007b08                HANDLE Buffer_1 = PsGetCurrentProcessId()
1c0007b0c                ExAcquireFastMutex(FastMutex: &data_1c0010b40)
1c0007b23                rax_5 = RtlLookupElementGenericTableAvl(Table: &table_stracktrace_related,
1c0007b23                    Buffer: &Buffer_1)
1c0007b23
1c0007b35                if (rax_5 != 0)
1c0007b3f                    rdi = sub_1c0007798(rax_5)
1c0007b3f
1c0007b48                ExReleaseFastMutex(FastMutex: &data_1c0010b40)
1c0007b48
1c0007b5f            if (rax_2 - 1 u> 0x62 || rax_5 == 0 || rdi == 0)
1c0007f9a                ExFreePoolWithTag(P: result, Tag: 'StPT')
1c0007fa6                result.b = 1
1c0007b5f            else
1c0007b65                int32_t var_264_1 = 0
1c0007b69                int32_t r14_1 = 0
```

To be able to do so in an efficient way, UCPD now leverages both `PsSetCreateProcessNotifyRoutineEx` and `PsSetLoadImageNotifyRoutine` to record the list of active processes and their loaded modules. The results are stored in an AVL table, along with some metadata.

```
1c0007268    NTSTATUS sub_1c0007268()

1c0007278        if ((FeatureV2 & 0x400) == 0)
1c000727a            return STATUS_SUCCESS
1c000727a
1c00072a3        RtlInitializeGenericTableAvl(Table: &table_stracktrace_related,
1c00072a3            CompareRoutine: sub_1c0007540, AllocateRoutine: sub_1c0006dc0,
1c00072a3            FreeRoutine: sub_1c0006f30, TableContext: nullptr)
1c00072af        data_1c0010b48 = 0
1c00072be        data_1c0010b50 = 0
1c00072c8        data_1c0010b40 = 1
1c00072d6        KeInitializeEvent(Event: &data_1c0010b58, Type: SynchronizationEvent, State: 0)
1c00072eb        NTSTATUS rax =
1c00072eb            PsSetCreateProcessNotifyRoutineEx(NotifyRoutine: sub_1c00075b0, Remove: 0)
1c00072eb
1c00072f9        if (rax s< STATUS_SUCCESS)
1c0007339            return rax
1c0007339
1c0007302        NTSTATUS rax_1 = PsSetLoadImageNotifyRoutine(NotifyRoutine: sub_1c0006f90)
1c0007302
1c0007312        if (rax_1 s>= STATUS_SUCCESS)
1c000732b            data_1c0010868 = 1
1c0007312        else
1c000731d            PsSetCreateProcessNotifyRoutineEx(NotifyRoutine: sub_1c00075b0, Remove: 1)
1c000731d
1c0007332        return rax_1
```

Conclusion

In this post, I tracked how the UCPD driver is evolving across different versions. Looking back, it is quite surprising to me that both sides went so far in the fight for the default browser, and I believe the trend will go on. I hope you enjoyed reading it! If you wish to check things out by yourself, you can find my analysis database here.

References

- https://kolbi.cz/blog/2024/04/03/userchoice-protection-driver-ucpd-sys/
- https://hitco.at/blog/windows-userchoice-protection-driver-ucpd/
- https://github.com/xusheng6/ucpd_analysis