

Custom keyboard layout persistence, by satokon and Jonas Lykk



Hello everybody, in this paper we will be presenting how it is possible to achieve persistence through a malicious custom keyboard layout driver.

Before proceeding any further, I must specially thank [Jonas Lykk](#) and [Matti](#) for their help. This would have not been possible without them, since they have helped me with the code fixing, and the idea itself came from Jonas after discussing new persistence techniques. Both are really smart people and I encourage you to follow them if you aren't already, you won't regret it.

This has been built with the sample US keyboard layout in <https://github.com/microsoft/Windows-driver-samples/tree/master/input/layout>, and additionally tested out in Windows 11 version 21h2 and Windows 10 version 21h1.

Warning: *I recommend running the binary once compiled in a VM, since I haven't implemented a way to uninstall the custom layout in the PoC. This can be easily done by deleting all the related registry subkeys which are **highlighted** in the explanation below. (See last section of this document for more, before Appendix).*

The main problem: how can we successfully get execution using our custom keyboard layout, after login, without any corruption?

To answer this question, we need to explain how things work at some basic level. In essence, keyboard layout drivers should only contain conversion tables in charge of transforming “scancode to virtual key code” and “virtual key code to character”, using arrays and structs for the implementation. Additionally, it should only have exports that will be called as necessary, to work out with these conversion tables.

Exports can vary depending on the requirements of the language, for example, the default KBDUS.DLL only has one export: KbdLayerDescriptor, meanwhile KBDJPN.DLL, has KbdNlsLayerDescriptor, KbdLayerRealDllFileNT4, KbdLayerRealDllFile, and KbdLayerMultiDescriptor.

I recommend looking at:

<https://github.com/microsoft/Windows-driver-samples/tree/master/input/layout> and <http://kbdlayout.info/> which gives an in-depth explanation of the different parts of how keyboard layouts are managed in Windows.

For our intentions, we only care where the main information is stored to avoid any type of possible corruption while loading our custom layout, which is mainly done in kernel mode. In this sense, the conversion tables are stored in the .data section of the PE in all cases, which can be seen in the sample's code and in the default layouts in the System32 directory.

One example that confirms the fact on how layouts are manipulated in kernel is NtUserLoadKeyboardLayoutEx (win32kbase.sys)

Here, there is a clear check for the .data section inside the function LoadKeyboardLayoutFile, specifically in ReadLayoutFile, where our handle to the layout DLL file will be used to load it and obtain a pointer to IMAGE_SECTION_HEADER for later processing.

```
else
    ImageBase = HIWORD(NtHeader->OptionalHeader.ImageBase);
NumberOfSections = NtHeader->FileHeader.NumberOfSections;
pCurrentSection = (&NtHeader->OptionalHeader + NtHeader->FileHeader.SizeOfOptionalHeader);
pSectionCurrent = pCurrentSection;
if ( !NumberOfSections )
    goto nullNumberSections;
pNextSection = &pCurrentSection[1];
while ( 1 )
{
    if ( pCurrentSection < pPEBuffer
        || !bEqualProcess && pNextSection - 1 < pCurrentSection
        || pNextSection >= pEndPe )
    {
        goto nullNumberSections;
    }
    if ( !strcmp_0(pCurrentSection, ".data") )
        break;
    pPEBuffer = pPe;
    ++pCurrentSection;
    pNextSection = pNextSection + 40;
    pSectionCurrent = pCurrentSection;
    if ( !--NumberOfSections )
        goto nullNumberSections;
}
```

Having these ideas in mind, creating our malicious keyboard driver will probably require the same conversion tables as your target keyboard layout.

However, since our payload will be probably storing hardcoded strings to create a process as a proof of execution, we need to store all of them in a different way than using .data section, which is the default section used for compile-time string storage.

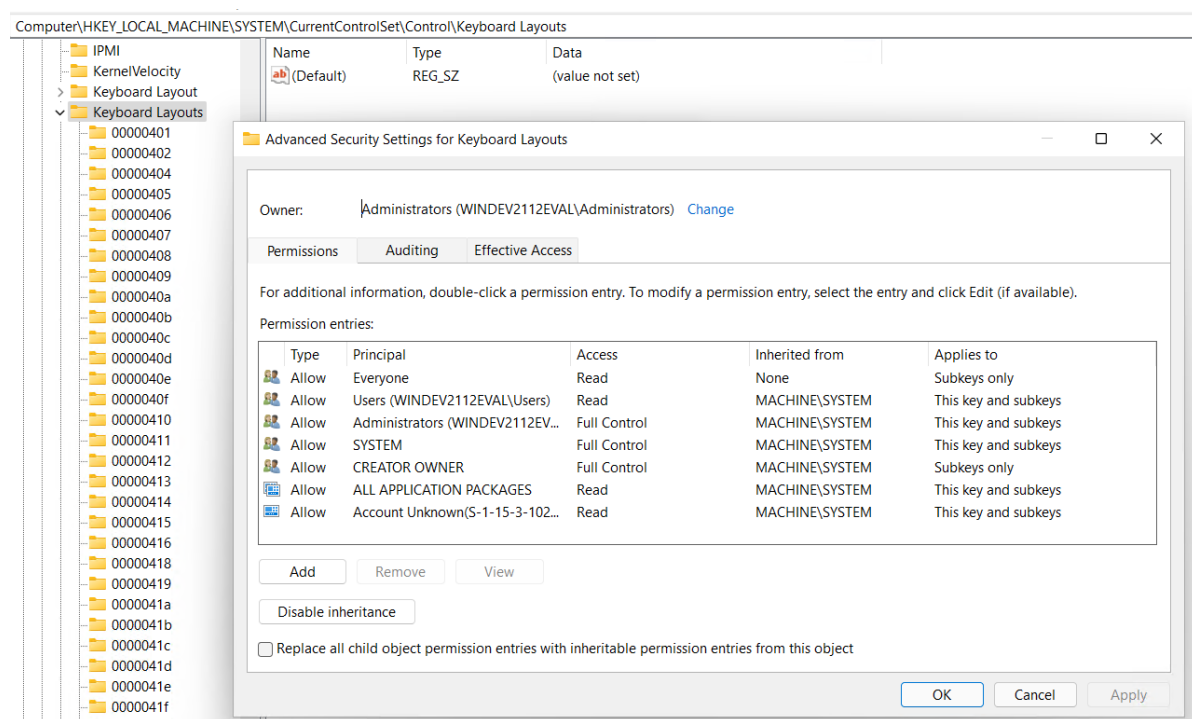
Two simple solutions I came across were either stack strings or generating an entire new section inside the keyboard driver, where I decided for this demonstration to use an entire new section.

After having an idea on how to approach things, it was necessary to consider how a custom keyboard layout can be installed as fast as possible for our purposes.

In this regard, one of the concepts involved in this process is keyboard layout identifiers or KLID for short, which is an 8-digit number (DWORD), where the LOWORD is the language ID (or LANGID) and the HIWORD is the sublanguage ID for a specific keyboard layout.

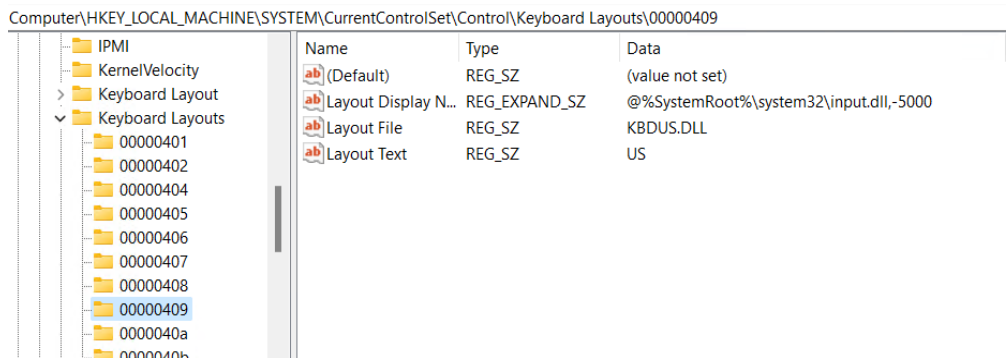
For this reason, we will have to write our own KLID as a subkey in

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Keyboard Layouts, where the DACL specifies only READ access for unprivileged users.



In this PoC, the code will get the last KLID that exists in the subkeys and add a random number which summed up with the LOWORD part of this KLID is less than 0xFFFF, since the language ID is 2 bytes, and I didn't wanted to overflow. I decided to use 0xC as a test, and it works fine, and changing it having in mind this constraint shouldn't cause any problem.

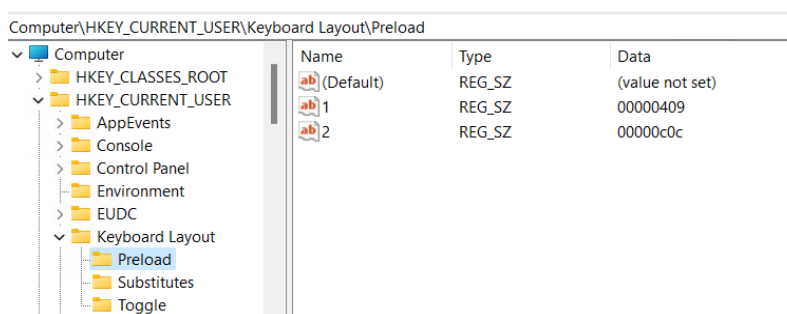
Once our custom KLID subkey has been created, it is necessary to create REG_SZ values. The bare minimum specific values that need to be created are “**Layout Text**”, “**Layout Display Name**” and most importantly, “**Layout File**”.



For our purposes, the most important value inside our language identifier subkey is “**Layout file**”, because it will store our keyboard driver dll. It is important to remember that this DLL must be in the System32 directory to be manipulated in kernel mode.

You probably also noticed I additionally added Layout ID as a subkey in the PoC. This is to make it blend with the last KLID subkey values in this “**Keyboard Layouts**” key in my Windows 11 VM, although it really doesn’t have any practical purpose for this specific usage, since it’s mostly related to device-specific variations to the same keyboard layout.

Once finished writing to the newly created language identifier subkey, its necessary to make the current user load it. There are two ways to do it that I’ve found so far: adding an entry with the value name + 1 to **HKEY_CURRENT_USER\Keyboard Layout\Preload** with our language identifier or calling InstallLayoutOrTip with the format that’s shown in MSDN: <https://docs.microsoft.com/en-us/windows/win32/tsf/installlayoutortip>



For example, for the screenshot shown above, if you wanted to load your layout by writing to the Preload subkey, you will have to create a REG_SZ value with 3 as the name, and your KLID as data, like “00000cbc”.

On the other hand, if you choose InstallLayoutOrTip method, you will have to use your KLID for the format string specified in the documentation.

What I’ve found to work has been simply using “%04x: %08x” as format string with the KLID of the custom layout as first argument, and NULL as second parameter for installation.

So far, we have a brief idea of how keyboard layouts are structured and how to install a custom one for our malicious purposes, but how can we get execution if keyboard layouts are only intended to have data and exports defined?

The solution: achieving execution with Winlogon!

After looking at Procmon in Windows 11 and finding interesting results, this is what I found after staring at IDA for some time.

At some point in the boot process, Winlogon will call UpdatePerUserSystemParameters, as it can be seen in the screenshot below.

```

f
g_WinlogonStage = 13;
if ( IsLoadLocalFontsPresent() )
{
    WLEventWrite(&WLEvt_UpdatePerUserSystemParameters_Start, 0);
    UpdatePerUserSystemParameters(0i64);
    WLEventWrite(&WLEvt_UpdatePerUserSystemParameters_Stop, 0);
}
g_WinlogonStage = 14;
```

Inside this function, LoadPreloadKeyboardLayouts will be called, which will basically get all KLIDs from **HKCU\Keyboard Layout\Preload**, using initialization file mapping.

Here, our custom keyboard layout identifier is also in, whether we use one of the two installation methods mentioned before.

```

loc_18001E7A1:          ; FLAGS
078 mov     r9d, ebp
078 mov     [rsp+78h+zero], 1 ; zero
078 movzx   r8d, si      ; cero
078 lea     rdx, [rsp+78h+pszKLID] ; pszKLID
078 xor     ecx, ecx      ; Zero
078 call    ?LoadKeyboardLayoutWorker@@YAPEAUHKL_@@@PEAU1@PEBGGIH@Z
078 call    LoadPreloadKeyboardLayouts ; Call Procedure
078 cmp     cs:word_1800B83B0, r15d ; Compare Two Operands
078 jnz     loc_18001E715 ; Jump if Not Zero (ZF=0)
```

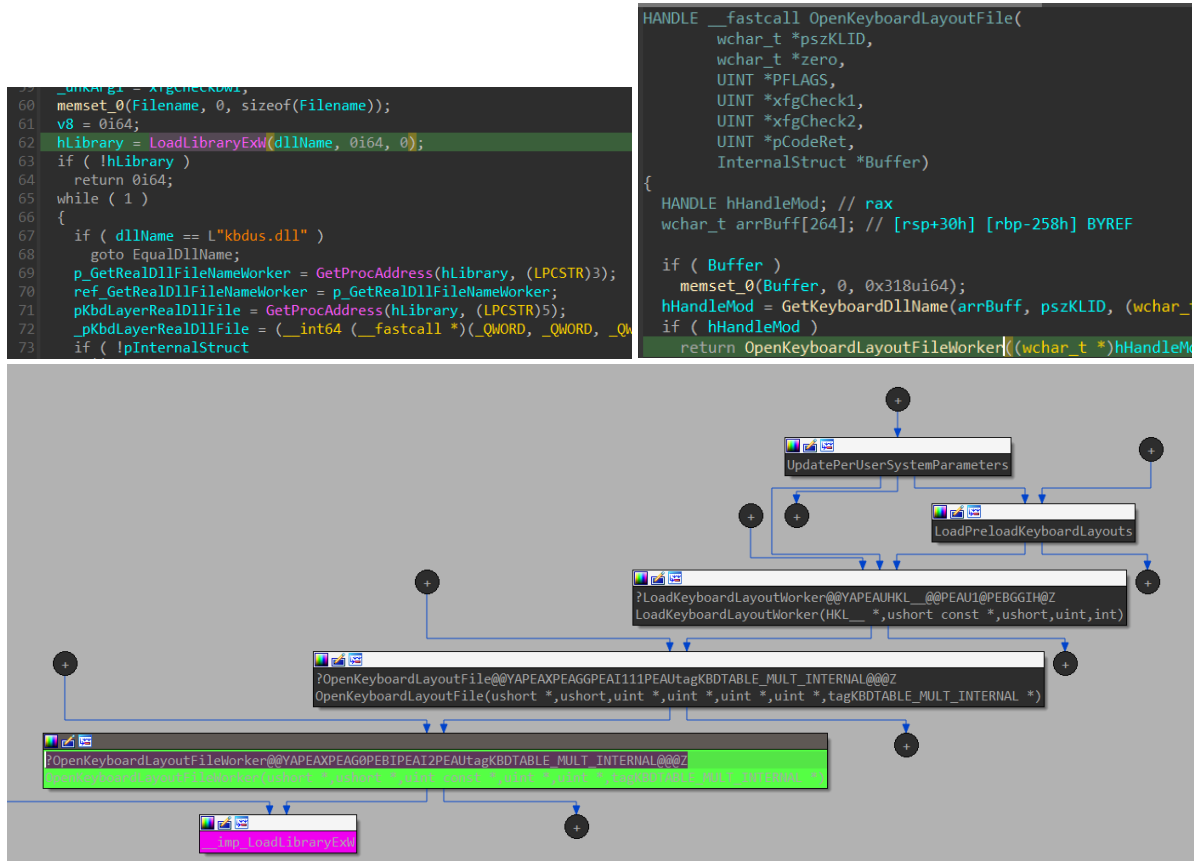
```

WORD __fastcall LoadPreloadKeyboardLayouts(__int64 a1, __int64 a2)

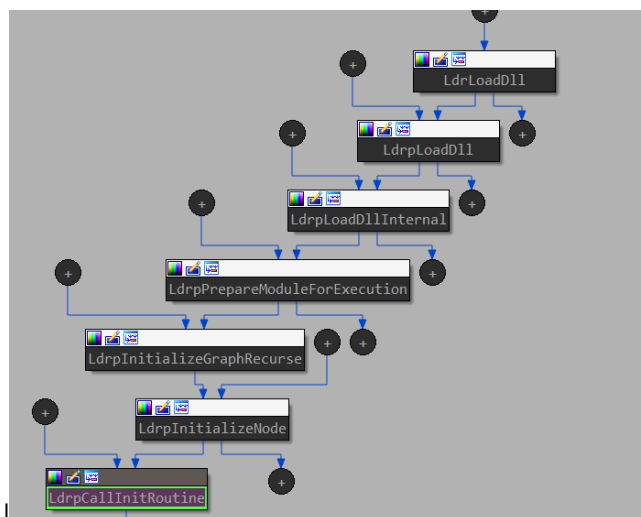
unsigned int v2; // ebx
DWORD result; // eax
WCHAR KeyName; // [rsp+30h] [rbp-38h] BYREF
WCHAR klidPreload[12]; // [rsp+38h] [rbp-30h] BYREF

if ( NtCurrentPeb()->SessionId == (unsigned int)WTSGetServiceSessionId(a1, a2)
    || (v2 = 1, NtCurrentPeb()->SessionId == (unsigned int)RtlGetActiveConsoleId()) )
{
    v2 = 2;
}
do
{
    StringCchPrintfW(&KeyName, 4ui64, L"%d", v2);
    result = GetPrivateProfileStringW(L"Preload", &KeyName, &WindowName, klidPreload, 9u, L"keyboardlayout.ini");
    if ( !klidPreload[0] )
        break;
    if ( result == -1 )
        break;
    result = (unsigned int)LoadKeyboardLayoutWorker(0i64, klidPreload, 0, 0x92u, 0);
    ++v2;
}
while ( v2 < 1000 );
return result;
```

Following the execution flow, inside LoadPreloadKeyboardLayouts the function LoadKeyboardLayoutWorker is called, which will eventually pass our malicious keyboard layout ID as an argument. From this function, OpenKeyboardLayoutFileWorker is called, where an apparently innocent LoadLibraryExW is executed, used for a simple handle check.



This is the start of the end, since LoadLibraryExW will internally call LdrLoadDll, which will eventually call LdrpCallInitRoutine, redirecting execution to the entrypoint established in the PE's Optional Header.



At this point, leveraging this LoadLibraryExW is easy, we just needed to define an entrypoint function and set it in the linker options for the keyboard layout sample driver used.

What I did inside my entrypoint function is call a simple CreateProcess for cmd.exe, which was more than enough to make winlogon show a shell as System every single time the machine is rebooted after the user which has the keyboard custom layout installed, has logged in.

But there was one main issue: When I changed between keyboard layout to my custom layout, it spawned a shell everytime as unprivileged user, which was a little bit problematic.

The reason is because the internal function LoadKeyboardLayoutWorker is called by APIs such as LoadKeyboardLayoutA, which can also trigger the execution of our malicious entrypoint function inside the custom keyboard driver.

What I decided to do was to check for "winlogon.exe" module before executing our payload. This way we can only ensure the system shell after user login for now, or when winlogon loads the keyboard layout again, which can also become a problem.

For this reason, I created a global mutex with NtCreateMutant that will prevent multiple instances of cmd.exe once it's loaded at least once. This solved every issue related to multiple execution.

Limitations so far:

Once reached this point, I think it is worth the time to clearly mention the limitations of this persistence method:

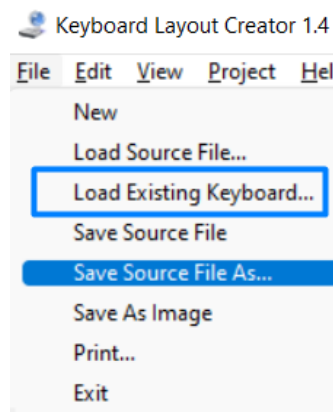
- 1.-It requires administrator privileges to be done.
- 2.-The range of keyboard layout samples in the Microsoft repository is limited, see the next page for a possible solution.
- 3.-Probably easily detected by AV minifilters.
4. The methods for installation explained above mainly target a specific user, but there is probably a way to do it for all users.

Another important question: Since there are limited official keyboard layout driver's sources from Microsoft samples, can we even generate our own keyboard layout sources?

The answer to this question is simple, and it is Microsoft Keyboard Layout Creator (MSKLC). This is a must to anyone that will be messing around this topic, for example, I reverse engineered the exports of KbdMsi.dll to understand how I can install my custom layout driver.

Inside the binaries of MSKLC, we are only interested in kbdutool.exe. This tool will generate the source code for our keyboard driver, from a .klc extension file. But how can we get the .klc file in the first place?

Two ways, you can either: use kbdlayout.info for the specific language that you are messing around or inside MSKLC you can use the following options:
FILE-> Load Existing Keyboard and pick your target language -> Save Source File as:



This method will query some registry values in your computer for the virtual keys, so it's more reliable than the first method, but the final decision it's up to you.

After storing the klc file, use the following command line:

`"kbdutool.exe -u -i -s <your klc file path>"`

if -u parameter doesn't work, try with -a.

This will generate a .c file, a .h file, a .def file and a .rc file, very similar to the ones found in the Microsoft samples, so it's just a matter of making a proper Visual Studio solution and compiling it.

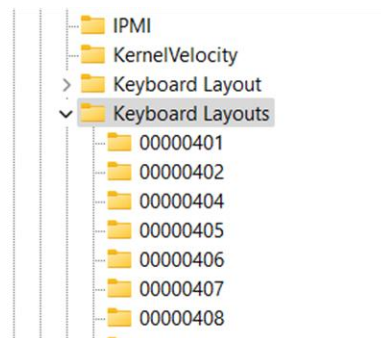
Maybe you could even reverse which part of the binary deals with "Load Existing Keyboard" option and find a way to automate the process for all the languages.

How to uninstall the custom keyboard layout:

Maybe you are messing around the layouts already, but you are not sure how to uninstall it after using any of the methods I showed above.

This is easy:

1. Inside HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Keyboard Layouts, delete your KLID subkey for your custom layout.



2. Go to HKEY_CURRENT_USER\Keyboard Layout\Preload, delete the entry value that contains your custom layout KLID

 1	REG_SZ	00000409
 2	REG_SZ	00000419
 3	REG_SZ	0000080a

If used InstallLayoutOrTip as method, then you should also check for HKEY_CURRENT_USER\Keyboard Layout\Substitutes.

Here look for the value that contains your custom layout KLID and delete it.

3. Delete the custom layout driver in system32 folder.

4. Reboot.

Final thoughts and future ideas:

This method is obviously not the best for persistence, but it was mostly one of the challenges Jonas asked if it was possible, and it totally is.

Maybe you, the reader, could also be interested in replacing one specific default keyboard layout, for which I could totally encourage you to do it.

If you do mess around this, be aware: default layouts are signed with Authenticode.

A really bad C/C++ PoC for both the payload and the installer has been uploaded to the main repository.

Appendix:

1. LoadKeyboardLayoutA function:
<https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-loadkeyboardlayouta>
2. InstallLayoutOrTip function:
<https://docs.microsoft.com/en-us/windows/win32/tsf/installlayoutortip>
3. Language Identifiers: <https://docs.microsoft.com/en-us/windows/win32/intl/language-identifiers>
4. Literally everything you need to know in terms of terminology:
<http://kbdlayout.info/terminology>
5. Everything related to different layouts for different languages:
<http://kbdlayout.info/features/languages>
6. Source code for Microsoft keyboard layout samples:
<https://github.com/microsoft/Windows-driver-samples/tree/master/input/layout>
7. https://levicki.net/articles/2006/09/29/HOWTO_Build_keyboard_layouts_for_Windows_x64.php
8. Microsoft tool link: <https://www.microsoft.com/en-us/download/details.aspx?id=102134>