Reversing Xilisoft

Introduction:

In this tutorial I will discuss the encryption routine used by Xilisoft, this tutorial will not in any way show you how to crack/keygen Xilisoft products. But will show you how to retrieve the serial number you have already registered your program with.

When you register your program, the app stores this serial number in the registry, but first it encrypts it with the name you registered with. So let's get started.

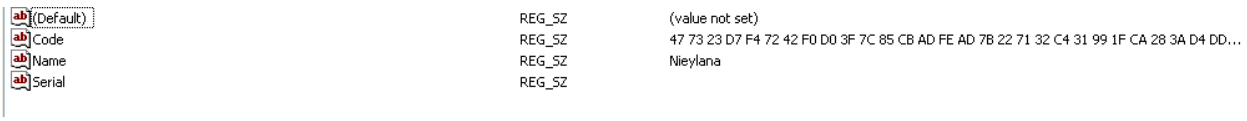
Target:

- Xilisoft Products
- Tools Used:
- RegEdit
- OllyDbg

Key in the Registry:

Open up the Registry Editor by clicking Start->Run and then typing 'regedit' without the quotes.

Next navigate to HKCU\Software\Xilisoft\<Product Name>\RegInfo, you should see keys like this:



- The Code value seems to contain encrypted data (the serial number).
- The Name value contains the Name you registered with (Decryption Key)
- The Serial Value is ALWAYS empty

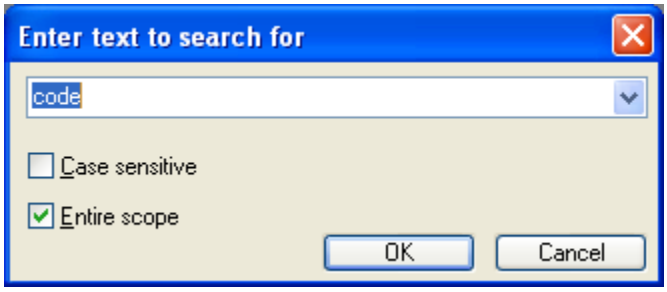
Find the Loading of Encrypted Data:

Open up <Product's exe>.exe (Xilisoft <Product Name> main EXE) in OllyDbg.

Now, if you have followed my Keygenning MD5 tutorial, you will know that all registration stuff is handled in the UILib DLL. So for Sound Recorder they use UILib8_MFCDLL.dll. Open up the Executable Modules window and select UILib8_MFCDLL and press [ENTER].

Once you have the UILib's code in the CPU window search for all referenced text strings by right clicking and selecting Search For->All referenced text strings.

Next, search for the word 'code' to find where it reads the encrypted data from the registry.



You will find the first one at 0038C3B8 set a BP here, press Ctrl+L to search for others, place a BP on every reference to 'code'. (Should be a total of 3 references). Now run the application.

OllyDbg should pause at 0038D1B6 on the push statement we BPd earlier. Go ahead and step up to the CALL ESI statement:

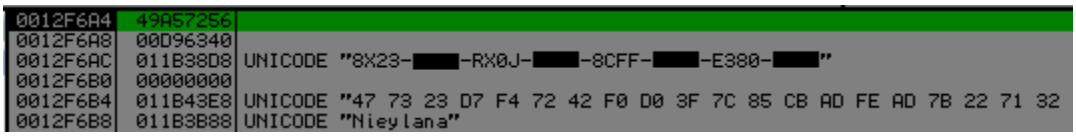
```
0038D1B4  .  6A 00          PUSH 0
0038D1B6  .  68 FC083D00    PUSH 003D08FC                UNICODE "Code"
0038D1BB  .  50             PUSH EAX
0038D1BC  .  FFD6           CALL ESI                     ADVAPI32.RegQueryValueExW
```

You can see here that it's going to get the encrypted data from the registry. So we have found where the app loads the encrypted data. Next is to find a point at which it's been decrypted. Then we will search in-between to find the Encrypt/Decrypt routine.

Find Decrypted Data

From the CALL ESI Statement, step with F8 until you see the decrypted data on the stack (Decrypted data will be the key you registered with). You should see this at 0038D238:

Now look at your stack:



(Note: I blacked out parts of mine, as to not give a serial away, due to legality issues)

So now that we have found a point that the data has been decrypted, let's make a note of all CALL statements we stepped over that are NOT system APIs.

- CALL 0038C000
- CALL 003BC290
- CALL 003BC6D0

Next, we need to dig into these routines and find out what role each one plays in the decryption of the data.

The Fist CALL (0038C000):

By taking a quick look at this routine, we see that they call wcslen:

According to the MSDN

> Each of these functions returns the number of characters in *string*, not including the terminating null character. **wcslen** is a wide-character version of **strlen**; the argument of **wcslen** is a wide-character string. **wcslen** and **strlen** behave identically otherwise.

So, we need to find what string it's passing, go ahead and set a BP on the call to wcslen. You will see that the encrypted data is what's being passed.

Later on down the routine we see a loop with a call to swscanf with the format string being "%2X" which means to convert a hex string to it's numeric value. Set a BP after the loop at the MOV ESI, [ESP+8] statement.

```
0038C075  >  6A 03          PUSH 3
0038C077  .  33D2           XOR EDX,EDX
0038C079  .  8D4424 18      LEA EAX,[ESP+18]
0038C07D  .  895424 18      MOV [ESP+18],EDX
0038C081  .  56             PUSH ESI
0038C082  .  50             PUSH EAX
0038C083  .  895424 24      MOV [ESP+24],EDX
0038C087  .  FFD3           CALL EBX
0038C089  .  8D4C24 1C      LEA ECX,[ESP+1C]
0038C08D  .  51             PUSH ECX
0038C08E  .  8D5424 24      LEA EDX,[ESP+24]
0038C092  .  68 E8083D00    PUSH 003D08E8                    format = "%2X"
0038C097  .  52             PUSH EDX                         wstr
0038C098  .  FF15 14F93C00  CALL [<&MSVCR71.swscanf>]        swscanf
0038C09E  .  8B4424 28      MOV EAX,[ESP+28]
0038C0A2  .  83C4 18        ADD ESP,18
0038C0A5  .  50             PUSH EAX
0038C0A6  .  8D4C24 10      LEA ECX,[ESP+10]
0038C0AA  .  FF15 34F33C00  CALL [<&MFC71U.#894>]            MFC71U.7C29B51B
0038C0B0  .  83C6 06        ADD ESI,6
0038C0B3  .  4F             DEC EDI
0038C0B4  .^ 75 BF          JNZ SHORT 0038C075
0038C0B6  >  8B75 08        MOV ESI,[EBP+8]
```

Continue running the routine, until you get to the BP set on the MOV ESI statement, step once with F8. You should now be on a LEA ECX, [ESP+C] statement, go ahead and step this statement and then follow the address loaded into ECX in the dump.

```
011D38E0  47 00 73 00  23 00 D7 00  ■■ 00 72 00  42 00 F0 00  G.s.#.╫.■.r.B.≡.
011D38F0  ■■ 00 3F 00  ■■ 00 85 00  CB 00 AD 00  ■■ 00 AD 00  ■■.¦.à.╦.↓.■.↓.
011D3900  7B 00 22 00  71 00 32 00  C4 00 31 00  99 00 1F 00  {.".q.2.─.1.Ö.▼.
011D3910  CA 00 28 00  3A 00 D4 00  ■■ 00 27 00  83 00 D5 00  ╩.(.:.╘.■.'.â.ƒ.
011D3920  30 00 F7 00  BF 00 78 00  41 00 C8 00  ■■ 00 00 00  0.≈.┐.x.A.╚.■...
```

(Note: Some bytes have been blacked out because of the possibility to obtain a valid serial number from it)

So we can see that this routine takes the encrypted data loaded from the registry and converts the unicode string into the hexadecimal equivalent.

The Second CALL (003BC290)

This routine is not of much value to us, although it would seems so, this routine appears to be setting some constants prior to the encryption, but I assure you we don't need these constants right now:

```
003BC294  .  894D FC        MOV [EBP-4],ECX
003BC297  .  8B45 FC        MOV EAX,[EBP-4]
003BC29A  .  C700 C0443D00  MOV DWORD PTR [EAX],003D44C0
003BC2A0  .  8B4D FC        MOV ECX,[EBP-4]
003BC2A3  .  C741 08 DF9B5713  MOV DWORD PTR [ECX+8],13579BDF
003BC2AA  .  8B55 FC        MOV EDX,[EBP-4]
003BC2AD  .  C742 0C E0AC6824  MOV DWORD PTR [EDX+C],2468ACE0
003BC2B4  .  8B45 FC        MOV EAX,[EBP-4]
003BC2B7  .  C740 10 3175B9FD  MOV DWORD PTR [EAX+10],FDB97531
003BC2BE  .  8B4D FC        MOV ECX,[EBP-4]
003BC2C1  .  C741 14 62000080  MOV DWORD PTR [ECX+14],80000062
003BC2C8  .  8B55 FC        MOV EDX,[EBP-4]
003BC2CB  .  C742 18 20000040  MOV DWORD PTR [EDX+18],40000020
003BC2D2  .  8B45 FC        MOV EAX,[EBP-4]
003BC2D5  .  C740 1C 02000010  MOV DWORD PTR [EAX+1C],10000002
003BC2DC  .  8B4D FC        MOV ECX,[EBP-4]
003BC2DF  .  C741 20 FFFFFF7F  MOV DWORD PTR [ECX+20],7FFFFFFF
003BC2E6  .  8B55 FC        MOV EDX,[EBP-4]
003BC2E9  .  C742 24 FFFFFF3F  MOV DWORD PTR [EDX+24],3FFFFFFF
003BC2F0  .  8B45 FC        MOV EAX,[EBP-4]
003BC2F3  .  C740 28 FFFFFF0F  MOV DWORD PTR [EAX+28],0FFFFFFF
003BC2FA  .  8B4D FC        MOV ECX,[EBP-4]
003BC2FD  .  C741 2C 00000080  MOV DWORD PTR [ECX+2C],80000000
003BC304  .  8B55 FC        MOV EDX,[EBP-4]
003BC307  .  C742 30 000000C0  MOV DWORD PTR [EDX+30],C0000000
003BC30E  .  8B45 FC        MOV EAX,[EBP-4]
003BC311  .  C740 34 000000F0  MOV DWORD PTR [EAX+34],F0000000
003BC318  .  8B4D FC        MOV ECX,[EBP-4]
003BC31B  .  C741 04 00000000  MOV DWORD PTR [ECX+4],0
003BC322  .  8B45 FC        MOV EAX,[EBP-4]
```

The Third CALL (003BC6D0):

For this final call before everything is decrypted, we should probably note what parameters are pushed to it. Set a BP on this call statement and then run until the BP.

Once you hit the BP look at the stack, there are 2 values passed to it, follow each in the dump and you will notice that one of them contains the Encrypted data that was converted from String to Hex by first call, and the other contains the decryption key (in the case the Name we registered with)

```
003BC6D0  r$  55              PUSH EBP
003BC6D1  .   8BEC            MOV EBP,ESP                    mov stack pointer to EBP
003BC6D3  .   83EC 10         SUB ESP,10                     sub 10 from stack
003BC6D6  .   894D F0         MOV [EBP-10],ECX
003BC6D9  .   8B45 08         MOV EAX,[EBP+8]                mov name to EAX
003BC6DC  .   50              PUSH EAX                       PUSH NAME
003BC6DD  .   8B4D F0         MOV ECX,[EBP-10]               mov pointer to
003BC6E0  .   8B11            MOV EDX,[ECX]
003BC6E2  .   8B4D F0         MOV ECX,[EBP-10]
003BC6E5  .   FF52 08         CALL [EDX+8]                   Init Key Routine
003BC6E8  .   8B45 0C         MOV EAX,[EBP+C]
003BC6EB  .   50              PUSH EAX                       s
003BC6EC  .   FF15 08F93C00   CALL [<&MSVCR71.wcslen>]       wcslen
003BC6F2  .   83C4 04         ADD ESP,4
003BC6F5  .   8945 FC         MOV [EBP-4],EAX                MOV into EBP-4 len of encrypted code
003BC6F8  .   C745 F8 00000000 MOV DWORD PTR [EBP-8],0       zero out EBP-8
003BC6FF  .v  EB 09           JMP SHORT 003BC70A
003BC701  >   8B4D F8         MOV ECX,[EBP-8]
003BC704  .   83C1 01         ADD ECX,1
003BC707  .   894D F8         MOV [EBP-8],ECX
003BC70A  >   8B55 F8         MOV EDX,[EBP-8]                mov EBP-8 to EDX
003BC70D  .   3B55 FC         CMP EDX,[EBP-4]                cmp EBP-8 (current char) to EBP-4 (code length)
003BC710  .v  73 2D           JNB SHORT 003BC73F             if we've decrypted whole thing then jump
003BC712  .   8B45 F8         MOV EAX,[EBP-8]                mov current char offset to EAX
003BC715  .   8B4D 0C         MOV ECX,[EBP+C]                mov encrypted code to ECX
003BC718  .   66:8B1441       MOV DX,[ECX+EAX*2]             mov DX, next char
003BC71C  .   66:8955 F4      MOV [EBP-C],DX                 mov current char to EBP-C
003BC720  .   8D45 F4         LEA EAX,[EBP-C]
003BC723  .   50              PUSH EAX
003BC724  .   8B4D F0         MOV ECX,[EBP-10]
003BC727  .   8B11            MOV EDX,[ECX]
003BC729  .   8B4D F0         MOV ECX,[EBP-10]
003BC72C  .   FF52 10         CALL [EDX+10]                  decrypt current byte
003BC72F  .   8B45 F8         MOV EAX,[EBP-8]
003BC732  .   8B4D 0C         MOV ECX,[EBP+C]
003BC735  .   66:8B55 F4      MOV DX,[EBP-C]
003BC739  .   66:891441       MOV [ECX+EAX*2],DX
003BC73D  .^  EB C2           JMP SHORT 003BC701
003BC73F  >   8BE5            MOV ESP,EBP
003BC741  .   5D              POP EBP
003BC742  L.  C2 0800         RET 8
```

(This is the entire routine, we will now dissect it as small as we need to understand what's going on)

The first part of interest in this routine is the PUSH EAX statement followed by the CALL [EDX+8], set a BP on the CALL [EDX+8], so we can see what's passed to it with the PUSH EAX statement. After getting to the CALL [EDX+8] statement, look at EAX, it contains our decryption key (Nieylana in my case).

Let's step into the CALL [EDX+8]:

CALL [EDX+8]

This routine is quite long so I won't go and explain every single line, but only the lines that need special mention. The first line to mention is the call to wcslen which returns the length of the decryption key. (so mine will return 8).

```
003BC3C8  .   51              PUSH ECX
003BC3C9  .   FF15 08F93C00   CALL [<&MSVCR71.wcslen>]       s
003BC3CF  .   83C4 04         ADD ESP,4                      wcslen
003BC3D2  .   8945 F4         MOV [EBP-C],EAX
003BC3D5  .   837D F4 0C      CMP DWORD PTR [EBP-C],0C
```

(It then compares the key length to 12d)

After this CMP is a JNB, meaning if the key length is NOT BELOW 12, jump, otherwise continue on.

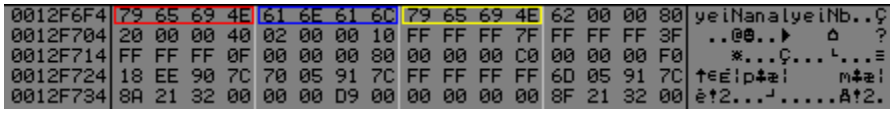
If it didn't jump (your key is less than 12 chars long), you will enter some loops that will pad the key to 12 characters, so my "Nieylana" becomes "NieylanaNiey"

After the key has been padded to 12 characters long, it then continues with the rest of the routine.

The main work of this function is done at 003BC461, the way they coded it makes it quite hard to understand so what I recommend is to go to the highlighted line:

Follow the address in [EAX+8] in dump, and then set a BP on the line after the loop which should be MOV EAX, [EBP-14]. Press F9 and run to BP.

The dump pane for my Key now looks like this:



It appears they have set 3 DWORDS to values based on the Key… the pattern for such is

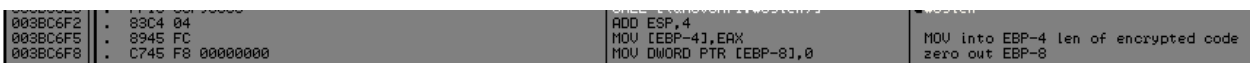
- DWORD1 = First 4 bytes of Key
- DWORD2 = Middle 4 bytes of Key
- DWORD3 = Last 4 bytes of Key

These DWORD (from now on referred to as Key1, Key2, and Key3) will be used later on. Just remember how they set these.

The Third CALL (003BC6D0) Again:

After these 3 values have been set, the following lines are executed:

This moves the address of the Encrypted Data to EAX, and the calls wcslen on that string which returns the length of it, should be 0x27 (or 39 decimal)



Next, we move the length of the string into [EBP-4] (this serves as the counter so we know when we've looped for the whole encrypted string). And then we zero out whatever is in [EBP-8]