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:

(Default)	REG_SZ	(value not set)
Code	REG_SZ	47 73 23 D7 F4 72 42 F0 D0 3F 7C 85 CB AD FE AD 7B 22 71 32 C4 31 99 1F CA 28 3A D4 DD…
Name	REG_SZ	Nieylana
Serial	REG_SZ	

- 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

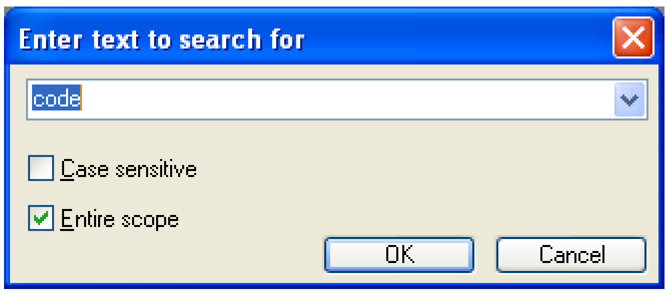
Find the Loading of Encrypted Data:

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].

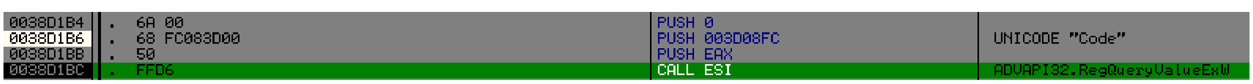
Once you have the UILib’s code in the CPU window search for all referenced text strings by right clicking and selecting Search For->All referenced text strings.

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:

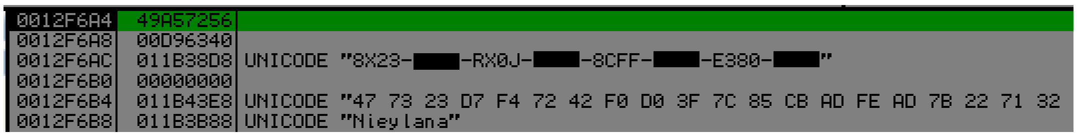


You can see here that it’s going to get the encrypted data from the registry. So we have found where the app loads the encrypted data. Next is to find a point at which it’s been decrypted. Then we will search in-between to find the Encrypt/Decrypt routine.

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 [redacted] 00 72 00 42 00 F0 00
011D38F0 [redacted] 00 3F 00 [redacted] 00 85 00 CB 00 AD 00 [redacted] 00 AD 00
011D3900 7B 00 22 00 71 00 32 00 C4 00 31 00 99 00 1F 00
011D3910 CA 00 28 00 3A 00 D4 00 [redacted] 00 27 00 83 00 D5 00
011D3920 30 00 F7 00 BF 00 78 00 41 00 C8 00 [redacted] 00 00 00
```

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

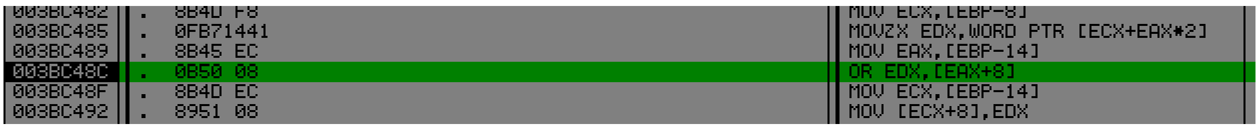
(It then compares the key length to 12d)

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"

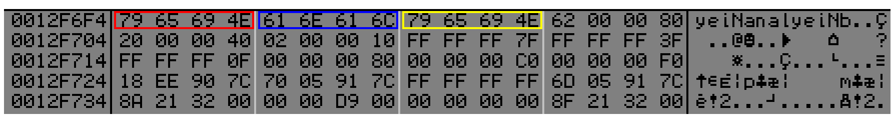
After the key has been padded to 12 characters long, it then continues with the rest of the routine.

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

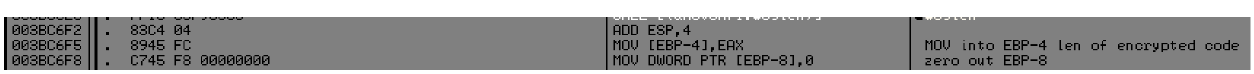
- DWORD1 = First 4 bytes of Key
- DWORD2 = Middle 4 bytes of Key
- DWORD3 = Last 4 bytes of Key

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:

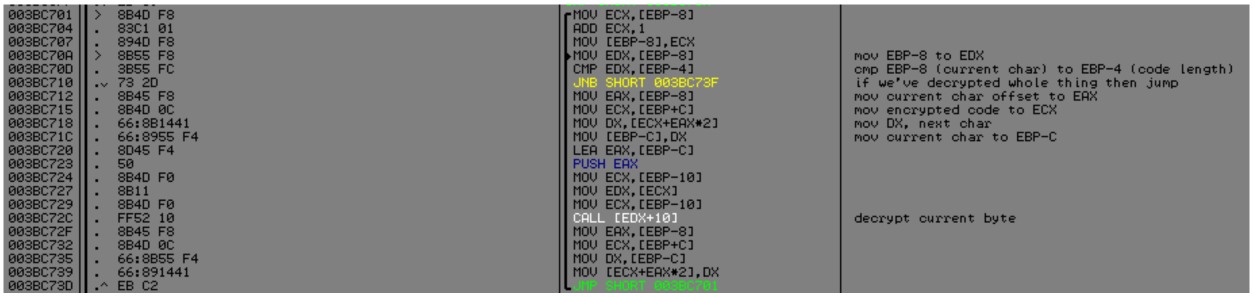
After these 3 values have been set, the following lines are executed:

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]

The next part of this routine is where the magic happens:



As you can see from the comments in OllyDbg, this loop goes through every byte in the encrypted data, and call [EDX+10] for each byte

Before we dig into [EDX+10], let's put what we've learned so far into perspective.

- The First Call we investigated converted the Unicode Hex string, into the hexadecimal equivalent.
- The Second Call initialized some constants that we don't really need to worry about
- The Third Call is effectively a call to DecryptString() which is passed the encrypted string, and the decryption key.
- This DecryptString() then calls [EDX+8] which is basically the Init routine for the Decryption, which sets 3 DWORDS to certain values based on the Key which was padded to 12 bytes.
- The DecryptString() then gets the length of the Encrypted String, and loops, each time PUSHing EAX to the stack, which contains the address holding the next byte to be decrypted, and CALLing [EDX+10] to decrypt the current byte

Byte Decryption:

Now that we have summed up what we've learned thus far, we are going to look into the CALL [EDX+10], which is the call that decrypts the individual bytes

Here is the beginning of the routine, remember this part is executed for every encrypted byte:

```
003BC560  55                 PUSH EBP                      preserve EBP
003BC561  8BEC               MOV EBP,ESP                   mov StackPointer to EBP
003BC563  83EC 14            SUB ESP,14                    sub 14h from ESP
003BC566  894D EC            MOV [EBP-14],ECX
003BC569  C745 F0 00000000   MOV DWORD PTR [EBP-10],0
003BC570  66:C745 FC 0000    MOV WORD PTR [EBP-4],0
003BC576  8B45 EC            MOV EAX,[EBP-14]
003BC579  8B48 0C            MOV ECX,[EAX+C]               mov ecx, Key2
003BC57C  83E1 01            AND ECX,1                     Test Odd/Even
003BC57F  894D F4            MOV [EBP-C],ECX               store result in EBP-C
003BC582  8B55 EC            MOV EDX,[EBP-14]
003BC585  8B42 10            MOV EAX,[EDX+10]              mov eax, Key3
003BC588  83E0 01            AND EAX,1                     Test Odd/Even
003BC58B  8945 F8            MOV [EBP-8],EAX               store result in EBP-8
003BC58E  C745 F0 00000000   MOV DWORD PTR [EBP-10],0      zero out loop counter
003BC595  EB 09              JMP SHORT 003BC5A0
```

Other than doing some stack set-up for the operation this part sets 2 initial values, one at [EBP-C] and [EBP-8]. [EBP-C] contains a 1 or 0 depending on the result of Key2 AND 1, same goes for [EBP-8] except Key3 was tested. This part also sets a ZERO to the loop counter at [EBP-10]. And finally the JMP statement jumps us into the loop.

The loop can seem quite intimidating, after all there's approx 114 lines of code in this whole routine, most of which is the loop. But relax, just step back a little and try to see a big picture. This loop contains a single decision, which will then branch us 1 of 2 ways. And once we branch we're faced with a single decision which again will branch us 1 of 2 ways.

So we need to find what determines these branches, and then once we find all possible paths, then start interpreting the code.

The First decision is made here:

```
003BC5AD  8B42 08   MOV EAX,[EDX+8]       mov eax, key1
003BC5B0  83E0 01   AND EAX,1             Test Odd/Even
003BC5B3  74 65     JE SHORT 003BC61A     jump if Key1 is Even
```

If Key1 was Even the next decision is here:

```
003BC631  8B48 10   MOV ECX,[EAX+10]      mov eax, Key3
003BC634  83E1 01   AND ECX,1             Test Odd/Even
003BC637  74 23     JE SHORT 003BC65C     jump if Key3 is Even
```

If Key1 was odd, and we didn't jump the next decision is located here:

```
003BC5D2  8B48 0C   MOV ECX,[EAX+C]       mov eax, Key2
003BC5D5  83E1 01   AND ECX,1             Test Odd/Even
003BC5D8  74 23     JE SHORT 003BC5FD     jump if Key2 is Even
```

Laying out the Loop

There are 4 possible paths from what we saw above:

- Key1 is Even – Key3 is Even
- Key1 is Even – Key3 is Odd
- Key1 is Odd – Key2 is Even
- Key1 is Odd – Key2 is Odd

The programming challenge we face here can be defeated by simply using nested If statements.

So now we will look at code that is executed for each of the 4 pathways.

Please note that all constants found in the comments can be found simply by stepping through the code.

These are the constants that were set in the 2nd call we looked at and I told you wasn't important, that because we'll see them here.

Key1 Is Even

This code is executed when Key1 is Even, regardless of the state of Key3:

```
003BC61A  >  8B55 EC      MOV EDX,[EBP-14]
003BC61D  .  8B42 08      MOV EAX,[EDX+8]      mov eax, Key1
003BC620  .  D1E8         SHR EAX,1            Shift Right 1 bit
003BC622  .  8B4D EC      MOV ECX,[EBP-14]
003BC625  .  2341 20      AND EAX,[ECX+20]     AND result with 7FFFFFFF
003BC628  .  8B55 EC      MOV EDX,[EBP-14]
003BC62B  .  8942 08      MOV [EDX+8],EAX      save result of math on Key1
003BC62E  .  8B45 EC      MOV EAX,[EBP-14]
003BC631  .  8B48 10      MOV ECX,[EAX+10]     mov eax, Key3
003BC634  .  83E1 01      AND ECX,1            Test Odd/Even
003BC637  .v 74 23        JE SHORT 003BC65C    jump if Key3 is Even
```

This code can be summarized with the following bullets:

- Right Shift Key1 by 1 bit
- AND result of RightShift with 7FFFFFFF
- Save result over top of Key1
- Test State of Key 3

Key1 is Even – Key3 is Even

After the execution of Key1 is Even Code, this is the code executed.

```
003BC65C  >  8B4D EC           MOV ECX,[EBP-14]
003BC65F  .  8B51 10           MOV EDX,[ECX+10]             mov edx, Key3
003BC662  .  D1EA              SHR EDX,1                    shift Key3 right 1 bit
003BC664  .  8B45 EC           MOV EAX,[EBP-14]
003BC667  .  2350 28           AND EDX,[EAX+28]             AND Key3 with 0FFFFFFF
003BC66A  .  8B4D EC           MOV ECX,[EBP-14]
003BC66D  .  8951 10           MOV [ECX+10],EDX             save maths on Key3
003BC670  .  C745 F8 00000000  MOV DWORD PTR [EBP-8],0      mov 0 to EBP-8
```

Notice that I stopped the Key3 is Even code at 003BC670, because at line 003BC677 all other pathways start executing as well so that's general code as well.

Key3 is Even code can be summarized with the following bullets:

- Right Shift Key3 by 1 bit
- AND result of RightShift with 0FFFFFFF
- Save result over top of Key3
- Move a 0 to [EBP-8]
- ([EBP-8] is used to hold the result of the branch taken if Key1 was even)
- Continue executing Common Code (Code executed by all branches)

Key1 is Even – Key3 is Odd

After the execution of Key1 is Even Code, this is the code executed.

```
003BC639  .  8B55 EC           MOV EDX,[EBP-14]
003BC63C  .  8B45 EC           MOV EAX,[EBP-14]
003BC63F  .  8B4A 10           MOV ECX,[EDX+10]             mov ecx, Key3
003BC642  .  3348 1C           XOR ECX,[EAX+1C]             xor Key3 by 10000002
003BC645  .  D1E9              SHR ECX,1
003BC647  .  8B55 EC           MOV EDX,[EBP-14]
003BC64A  .  0B4A 34           OR ECX,[EDX+34]              OR Key3 by F0000000
003BC64D  .  8B45 EC           MOV EAX,[EBP-14]
003BC650  .  8948 10           MOV [EAX+10],ECX
003BC653  .  C745 F8 01000000  MOV DWORD PTR [EBP-8],1      mov 1 into EBP_8
003BC65A  .v EB 1B             JMP SHORT 003BC677
```

This code can be summarized with the following bullets:

- XOR Key3 by 10000002
- OR the result by F0000000
- Move a 1 to [EBP-8]
- ([EBP-8] is used to hold the result of the branch taken if Key1 was even)
- JMP to Common Code (Code executed by all branches)

Key1 is Odd

This is code that will be executed in the event that Key1 is Odd, regardless of the state of Key2

```
003BC5B5 | .   | 8B4D EC  | MOV ECX,[EBP-14]   |
003BC5B8 | .   | 8B55 EC  | MOV EDX,[EBP-14]   |
003BC5BB | .   | 8B41 08  | MOV EAX,[ECX+8]    | mov eax, Key1
003BC5BE | .   | 3342 14  | XOR EAX,[EDX+14]   | XOR Key1 with 80000062
003BC5C1 | .   | D1E8     | SHR EAX,1          | shift EAX right 1 bit
003BC5C3 | .   | 8B4D EC  | MOV ECX,[EBP-14]   |
003BC5C6 | .   | 0B41 2C  | OR EAX,[ECX+2C]    | OR Key1 by 80000000
003BC5C9 | .   | 8B55 EC  | MOV EDX,[EBP-14]   |
003BC5CC | .   | 8942 08  | MOV [EDX+8],EAX    | Save results of math to Key1
003BC5CF | .   | 8B45 EC  | MOV EAX,[EBP-14]   |
003BC5D2 | .   | 8B48 0C  | MOV ECX,[EAX+C]    | mov eax, Key2
003BC5D5 | .   | 83E1 01  | AND ECX,1          | Test Odd/Even
003BC5D8 | .v  | 74 23    | JE SHORT 003BC5FD  | jump if Key2 is Even
```

This code can be summarized with the following bullets:

- XOR Key1 with 80000062
- Shift Result Right 1 bit
- OR Result by 80000000
- Save result to Key1
- Test state of Key2 (Odd/Even)
- Jump accordingly

Key1 is Odd – Key2 is Even

This code is executed after Key1 is Odd code.

```
003BC5FD | >   | 8B4D EC           | MOV ECX,[EBP-14]         |
003BC600 | .   | 8B51 0C           | MOV EDX,[ECX+C]          | mov edx, Key2
003BC603 | .   | D1EA              | SHR EDX,1                | Shift EDX right 1 bit
003BC605 | .   | 8B45 EC           | MOV EAX,[EBP-14]         |
003BC608 | .   | 2350 24           | AND EDX,[EAX+24]         | AND Result with 3FFFFFFFF
003BC60B | .   | 8B4D EC           | MOV ECX,[EBP-14]         |
003BC60E | .   | 8951 0C           | MOV [ECX+C],EDX          | Store result of math on Key2
003BC611 | .   | C745 F4 00000000  | MOV DWORD PTR [EBP-C],0  |
003BC618 | >.  | EB 5D             | JMP SHORT 003BC677       |
```

This code can be summarized with the following bullets:

- Shift Key2 right 1 bit
- AND result with 3FFFFFFF
- Save result to Key2
- Store a 0 in [EBP-C]
- ([EBP-C] is used to store result of branch taken in Key1 Is Odd branch)
- Jump to Common Code (Code executed by all branches)

Key1 is Odd – Key2 is Odd

This code is executed after Key1 is Odd code.

```
003BC5DA  .  8B55 EC          MOV EDX,[EBP-14]
003BC5DD  .  8B45 EC          MOV EAX,[EBP-14]
003BC5E0  .  8B4A 0C          MOV ECX,[EDX+C]             mov ecx, Key2
003BC5E3  .  3348 18          XOR ECX,[EAX+18]            xor Key2 by 40000020
003BC5E6  .  D1E9             SHR ECX,1                   shift result right by 1 bit
003BC5E8  .  8B55 EC          MOV EDX,[EBP-14]
003BC5EB  .  0B4A 30          OR ECX,[EDX+30]             OR result with C0000000
003BC5EE  .  8B45 EC          MOV EAX,[EBP-14]
003BC5F1  .  8948 0C          MOV [EAX+C],ECX             store result of math on Key2
003BC5F4  .  C745 F4 01000000 MOV DWORD PTR [EBP-C],1     mov 1 to EBP-C
003BC5FB  .v EB 1B            JMP SHORT 003BC618
```

This code can be summarized with the following bullets:

- XOR Key2 with 40000020
- Shift Result Right by 1 bit
- OR Result with C0000000
- Store Result over Key2
- Move a 1 to [EBP-C]
- ([EBP-C] is used to store result of branches taken in Key1 Is Odd branch)
- Jump Common Code (Code executed by all branches)

Common Code

This code is executed regardless of branch taken, but is executed after branches have been taken.

```
003BC677  >  0FB755 FC        MOVZX EDX,WORD PTR [EBP-4]  mov DecryptionKey to EDX and extend zeros
003BC67B  .  D1E2             SHL EDX,1                   shift EDX LEFT 1 bit
003BC67D  .  8B45 F4          MOV EAX,[EBP-C]             result of Key1 is Odd branch
003BC680  .  3345 F8          XOR EAX,[EBP-8]             xor Key1 Is Odd branch with Key1 is Even Branch
003BC683  .  0BD0             OR EDX,EAX                  or DecryptionKey with Result
003BC685  .  66:8955 FC       MOV [EBP-4],DX              store low word of result as NEW decryption key
003BC689  .^ E9 09FFFFFF      JMP 003BC597
```

This is a more complicated piece of code, but I will summarize with bullets:

- Take current DecryptionKey and multiply by 2 (Left Shift by 1 bit)
- BranchResult = [EBP-C] XOR [EBP-8]
- OR DecryptionKey by BranchResult
- Store low word of math as NEW Decryption Key
- Loop up to the top

After it jumps to the top of the loop we see it execute this code:

```
003BC597  >  8B4D F0        MOV ECX,[EBP-10]
003BC59A  .  83C1 01        ADD ECX,1
003BC59D  .  894D F0        MOV [EBP-10],ECX
003BC5A0  >  837D F0 08     CMP DWORD PTR [EBP-10],8    cmp Counter to 8
003BC5A4  .v 0F8D E4000000  JGE 003BC68E                jump if 8
```

This simply increments the loop counter [EBP-10] by 1, and then compares it to 8, if we've looped 8 times, jump out of the loop and continue Byte Decryption Routine.

Back to Byte Decryption Routine:

After jumping out of the loop, the following code is executed:

```
003BC68E  >  8B4D 08      MOV ECX,[EBP+8]                mov Encrypted Byte to ECX
003BC691  .  0FB711       MOVZX EDX,WORD PTR [ECX]       mov single encrypted byte to EDX and zero extend
003BC694  .  0FB745 FC    MOVZX EAX,WORD PTR [EBP-4]     mov computed val(figure out) to EAX (single byte)
003BC698  .  33D0         XOR EDX,EAX                    XOR the 2 values (decrypt byte)
003BC69A  .  8B4D 08      MOV ECX,[EBP+8]
003BC69D  .  66:8911      MOV [ECX],DX                   Store decrypted byte
003BC6A0  .  8B55 08      MOV EDX,[EBP+8]
003BC6A3  .  0FB702       MOVZX EAX,WORD PTR [EDX]
003BC6A6  .  85C0         TEST EAX,EAX
003BC6A8  .v 75 12        JNZ SHORT 003BC6BC
```

The code can be summarized with the following bullets:

- XOR Encrypted Byte with DecryptionKey (Figured out by loop described Above)
- Store Decrypted Byte in place of Encrypted Byte
- Jump, if Decrypted Byte is NOT Zero, to Return Statement

Pseudo-Code for DecryptByte Routine:

Now that we've discussed in depth, the Byte Decryption routine, we should be able to make a pseudo-code outline of the function

There were 3 parts to the Decrypt Byte Routine:

- Code before Loop
- Loop
- Code after Loop

```
~Function DecryptByte

        //Function is passed the EncryptedByte as it's ASCII value
        //Key1,Key2, and Key3 are Global Variables that have been set by InitCrypt()
        //Key1IsEven,Key1IsOdd as variables declared locally
        //DecryptionKey, LoopCounter are Declared Locally
~Init Variables
        Key1IsOdd =  Key2 AND 1
        Key1IsEven = Key3 AND 1
        LoopCounter = 0
~Begin Loop
        Do Until LoopCounter = 8
                If Key1 Is Even
                    o   Key1 = (SHR(Key1,1) AND 0x7FFFFFFF)
                        If Key3 Is Even
                            ▪   Key3 = (SHR(Key3,1) AND 0x0FFFFFFFF)
                        Else
                            ▪   Key3 = (Key3 XOR 0x10000002) OR 0xF0000000
                        End If
                    o   Key1IsEven = Key3 AND 1
                Else
                    o   Key1 = (Key1 XOR 0x80000062)
                    o   Key1 = SHR(Key1,1)
                    o   Key1 = Key1 OR 0x80000000
                        If Key2 Is Even
                            ▪   Key2 = (SHR(Key2,1) AND 0x3FFFFFFF)
                        Else
                            ▪   Key2 = (Key2 XOR 0x40000020)
                            ▪   Key2 = SHR(Key2,1)
                            ▪   Key2 = Key2 OR 0xC0000000
                        End If
                    o   Key1IsOdd = Key2 AND 1
                End If
                LoopCounter = LoopCounter + 1
                DecryptionKey = DecryptionKey  * 2
                BranchResult = Key1IsEven XOR Key1IsOdd
                DecryptionKey = DecryptionKey OR BranchResult
                DecryptionKey = DecryptionKey AND 0xFFFF
        Loop
        EncryptedByte = EncryptedByte XOR DecryptionKey
        Return Value of Encrypted Byte
```

Pseudo-Code for DecryptString

~ Function DecryptString()

// Function is Passed the EncryptedData as String, and the Key as a String

// EncryptedData is must be converted from string to an Array of Bytes, this is needed if the EncryptedData was a HexString

```
InitCrypt(Key)
LoopCounter = 1
Do until LoopCounter = Length of EncryptedData (be mindful of Hex Strings)
    NextChar = Next byte to Decrypt
    NextChar = DecryptByte(NextChar)
    DecryptedString = Concatenate newest Decrypted Byte to end of DecryptedString
Loop
Return DecryptedString
```

Summary:

In this tutorial we have covered many things some of them include:

- Locating Encrypted Data in Registry
- Catching Application in act of Loading Encrypted Data
- Stepping over until we found Decrypted Data
- Investigating each call made in-between
    - Call1 Simply converted the EncryptedString back into an Array of Bytes
    - Call2 Set up some constants, we didn't need to worry about this call
    - Call3 Did all the dirty work of decrypting the string
        - Call3 (DecryptString) called a function we'll call InitCrypt
            - InitCrypt() took the key we passed it and padded to 12 bytes
            - InitCrypt() also set 3 global variables Key1,Key2, and Key3 to their initial values
        - Call3 then looped for each character and passed the encrypted byte to DecryptByte()
        - DecryptByte went through some semi-complicated loops, and ultimately decrypts the byte
        - DecryptString then takes the returned (Decrypted) Byte and appeneds it to the rest of the DecryptedBytes

Conclusion:

In conclusion, We have successfully found, reversed, and written Pseudo-Code for routines to decrypt serial numbers that have been stored by Xilisoft Products. Where this would be applicable is if you have legally purchased a Xilisoft Product and registered your program. Then later if you have to format/reload, and you no longer have the Serial they gave you, you can obtain it from the encrypted data.

I hope you all had as much fun reading and learning about this as I did learning and writing about it. It is always my goal to make sure my tutorials are as easy to read and follow as possible. It is my sincerest hope that you walk away from this paper knowing a little be more about finding the crypt routines, and ways of analyzing them.

I have included Full Sources (and exe) for my Xilisoft Cryptor (Written in VB), as well as sources made for C/C++ programmers coded by Ghandi. All Sources are in the Zip File that came with this tutorial.

Also on the last page of this tutorial is a FlowChart of the Byte Cryption routine produced by IDA, this is provided to help visualize the layout of the function. I have overwritten the parameters and given more descriptive names in order to further aid in this.

Graph of Crypt_Byte