WIBU protection

knuth20 implementation analysis

by anonymous – 15.09.2006.

Intro

Couple of weeks ago, I faced WIBU dongle protection. Before that, I had no expirience with it. The first thing I've checked was CrackZ great site about dongles. It was very helpful since I found WIBU API documentation. Except that, there was couple of tutorials which explained only patching the right calls in target's code. At the beginning, I thought my target is also a example of lame dongle implementation and by simple patching the conditional jump(s) I could solve the whole issue. After reading a WIBU API documentation and tracing a target, I saw that there was no easy way solving this. The target that I was working on used WIBU to decrypt important data, so there was no simple IsDonglePresent() function call. Next thing I tried to look for WIBU emulator or at least monitor which would help me tracing the calls and arguments. Of course, same could be done by setting the BP in debugger, but I prefer dongle monitoring tool. You are free to call me lazy ;-)

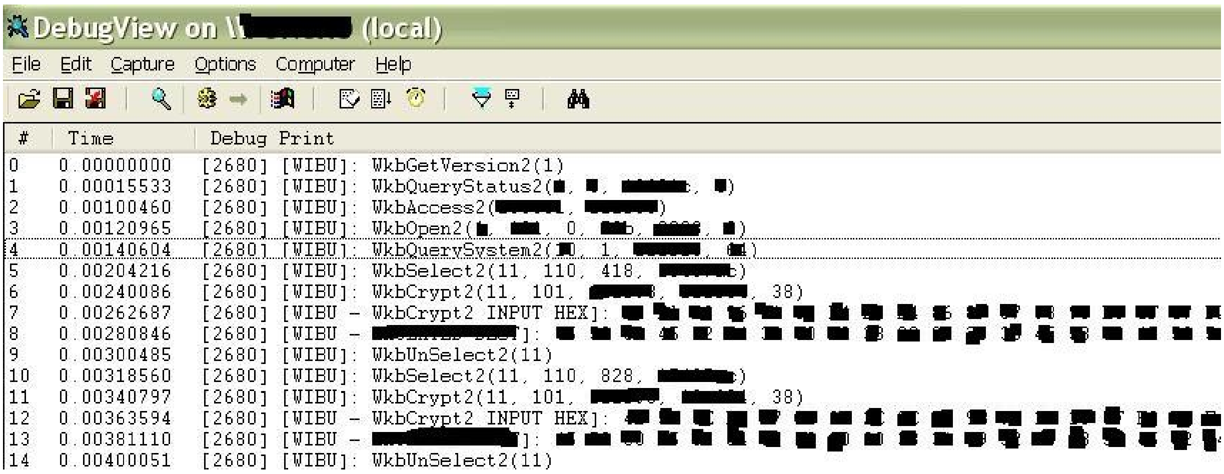
I had no luck searching WIBU monitoring tool or emulator. All I got was responses from dongle-emulator sellers trying to get the dump data and sell me their emulators. No useful information there there, not even an answer to single question about the protection. After all, that is the reason why I'm writting this short document.

Thanks to some nice guy on IRC #c4n I got my hands on WIBU SDK. Shortly after that, I started to write my own WIBU DLL (WkWin32.DLL). It wasn't anything special, I've coded most of the functions just to return correct value (TRUE). I wanted to keep the cracking on dongle-ground, without touching the target code itself if possible.

I've hit the wall when I faced WkbCrypt2 function. Returning the correct value in function wasn't enought. The target application implemented WkbCrypt2 function inside vital target's functions such as *Save Project*, *Open Project*, *Import* and similiar.

1st step – monitor the target

First thing, we got our monitor up & running. After that, start the target. We have to determinate which calls are being made and is there any encryption/decryption involved.



While reading the WIBU SDK, we see that WkbCrypt2 works with WkbSelect2. WkbSelect2 is used for algorithm selection and internal key generation.

Syntax

```
INT WkbSelect2(HWKBENTRY hwkbe, ULONG flCtrl, ULONG ulSelectCode, VOID *pvCtrl)
```

We see that 2nd parameter is used for algorithm selection and 3rd parameter is selection code. Based on selection code, WkbSelect2 generates internal key for knuth20 algorithm.

Another look at WIBU SDK gives us:

> ***flSelCtrl*** is only used if the WKB_AREA_SELECT flag is set and contains the selection flags which must be one of the following values:
>
> - **WKB_SEL_DIRECT** (0x0000) direct encryption/decryption via the WIBU-BOX
> - **WKB_SEL_KNUTH20** (0x0010) indirect Knuth encrypt/decryption
> - **WKB_SEL_DEC_FEAL** (0x0020) indirect original FEAL decryption
> - **WKB_SEL_ENC_FEAL** (0x0030) indirect original FEAL encryption
> - **WKB_SEL_DEC_PERM** (0x0040) indirect permutation decryption.
> - **WKB_SEL_ENC_PERM** (0x0050) indirect permutation encryption.
> - **WKB_SEL_SYMFEAL** (0x0060) indirect symmetric FEAL encryption/decryption.
> - **WKB_SEL_USE_CACHE** (0x00F0) uses a previously set and cached indirect selection, which was stored into the cache because the WKB_SEL_SET_CACHE was set in the **WKBSELECT** structure. The returned cache identification is stored in the *ulSelectCode* member instead of a Selection Code.
> - If the WKB_AREA_SELECT flag is not set, this value must be set to 0.

Based on that info, we are sure that that it uses knuth20 algorithm. This is a positive thing for us since this algorithm is not strong.

As I said, WkbCrypt2 is somehow implemented in Save Project, Open Project and similiar vital functions inside the target. The data which is decrypted is never the same. This excludes the table approach (record the data in/out and always return the correct data with fake DLL). I extracted the knuth20 algorithm from the official WkWin32.DLL and it goes like this:

```
                mov     ecx, cbSrc
                mov     eax, offset key
                mov     edx, ecx
                push    esi
                mov     esi, [eax+8]
                dec     ecx
                push    edi
                mov     edi, [eax+0Ch]
                push    ebx
                mov     ebx, pvDest
                inc     ecx
                push    ebp
                mov     ebp, pvCtrl
                mov     [esp+0e4h], cbSrc
loop_1:
                mov     cl, [eax+edi+18h]
                mov     dl, [eax+esi+18h]
                add     dl, cl
                mov     [eax+esi+18h], dl
                mov     cl, dl
                xor     edx, edx
                and     ecx, 0FFh
                mov     dl, [ebp+0]
                xor     edx, ecx
                inc     ebp
                test    ebx, ebx
                jz      end_of_pvDest
                mov     [ebx], dl
                inc     ebx
end_of_pvDest:
                test    esi, esi
                jz      loc_2000F442
                dec     esi
                jmp     taken_7
loc_2000F442:
                mov     esi, [eax+4]
taken_7:
                test    edi, edi
                jz      loc_2000F44C
                dec     edi
                jmp     taken_8
loc_2000F44C:
                mov     edi, [eax+4]
taken_8:
                mov     ecx, [esp+0e4h]
                dec     ecx
                mov     [esp+0e4h], ecx
                jnz     loop_1
                pop     ebp
                pop     ebx
                pop     edi
                pop     esi
```

2^{nd} step - Internal key calculation

Ok, when we apply this algorithm inside our fake WkbCrypt2 function, there's only one thing missing: internal key. As I said before, this key is generated somewhere inside WkbSelect2 function. At this moment, I do not have algorithm for that.
I was lucky that my target had fixed select codes passed in WkbSelect2 which guaranteed the same internal keys. The next thing you should ask yourself is : how do I retrieve that key?

You can set a BP in WkWin32.DLL inside this algorithm and check it out or you could calculate it based on the encrypted/decrypted data.

Take 1^{st} and 4^{th} byte from encrypted & decrypted data.

0xA7 ... 0x96 ... - encrypted
0x45 ... 0x45 ... - decrypted

A step)
0xA7 XOR 0x45 = 0xE2 – proceed with this – *FIRST BYTE*
0x96 XOR 0x45 = 0xD3 – remember this – *SECOND BYTE*

B step)
0xE2 / 0x2 (this is constant) = 0x71 – *potencial key*

0x71 * 3 (this is constant) = 0x153

Note: WkbSelect2 works only with one byte, so always discard other bytes

C step)
Compare: 0x153 & 0xD3(*second byte*) – not same!

Repeat the process from step B, but add byte '1' to in front first byte.

0x1E2 / 0x2 = 0xF1 – *potencial key*

0xF1 * 3 = 0x2D3 – *remember: only one byte, discard the rest. Correct is: 0xD3*

Compare: 0xD3 and 0xD3 (*second byte*) – same!

We have the internal key and it's 0xF1.

3rd Internal key structure

```
unsigned char key_1[] = {0x10, 0x00, 0x00, 0x0, // 0x10
                         0x13, 0x00, 0x00, 0x00, // 0x13
                         0x13, 0x00, 0x00, 0x00, // 0x13
                         0x02, 0x00, 0x00, 0x00, // 0x20
                         0x0D, 0xF0, 0xAD, 0xBA, // 0xbaadf00d
                         0x0D, 0xF0, 0xAD, 0xBA, // 0xbaadf00d
                         0xF1, 0xF1, 0xF1, 0xF1, // fill this with key
                         0xF1, 0xF1, 0xF1, 0xF1, // fill this with key
                         0xF1, 0xF1, 0xF1, 0xF1, // fill this with key
                         0xF1, 0xF1, 0xF1, 0xF1, // fill this with key
                         0xF1, 0xF1, 0xF1, 0xF1, // fill this with key
                         0x0D, 0xF0, 0xAD, 0xBA, // 0xbaadf00d
                         0xAB, 0xAB, 0xAB, 0xAB, // 0xabababab
                         0xAB, 0xAB, 0xAB, 0xAB}; // 0xabababab
```

As you can see, the knuth20 key size of 56 bytes.
Now with this knowledge, we can create a key for every select code that we need.
I've placed a switch-case in my WkbSelect2 function which does the following:

```
switch (ulSelectCode) {
	case 0x418:
		memcpy(key_current, key_1, 56);
		break;
	case 0x999:
		memcpy(key_current, key_2, 56);
		break;
	default:
		error("unknown select code!");
		break;
}
```

Inside my WkbCrypt2, algorithm always uses key_current as the internal key data and that's it! I got my dongle emulated.

As you can see, this is very limited emulation. I was lucky that this was enought for my target. There was 25 fixed select codes, so that means 25 internal keys. I've calculated them and added in my fake DLL ... added knuth20 algorithm which uses it, that's it.

Conclusion

The right thing to do would be to dump all possible keys, so if your target uses the knuth20 algorithm, but select code is different every time, you should code a dumper and retrieve all possible keys.

That would look something like:

```
call WkbAccess2
call WkbOpen2
for (i = 0; i < 0xFFFF; i++) {
 call WkbSelect2 // use 'i' as SelectCode
 call ReadProcessMemory // use 0x200175A4 harcoded addr (in wkwin32) to get addr_xxx_address
 call ReadProcessMemory // read whole key block (56 bytes)
 call SQL_write_somewhere_selectcode_and_keyblock
 call WkbUnSelect2
}
call WkbClose2
```

My target WkbSelect2 ulSelectCode was limited to 0xFFFF possible keys, you should also check if that's in your case too.

Or even better would be to extract the internal key generation algorithm from WkbSelect2! ;-)

Thank you, this is all I have to say about this. Maybe someone will find it useful!

All this wouldn't be possible without Sab's help!

Good luck!