IObit Protected Folder Authentication Bypass

Author: Kyriakos Economou

Date: 25d/05m/2012

Blog: http://anti-reversing.com

Aknowledgements

I would like to dedicate this article to all my friends, they know who they are, and to Irene, for her love and support.

Intro

From time to time I come across various security tools and utilities and sometimes I enjoy analysing them in order to evaluate their effectiveness, especially if they are not given for free. In order to be clear, I am not saying that a free security tool shouldn't be secure, especially if it claims to be.
However, if someone has also to pay for it, then of course he has the right to expect something more, regardless the price that he needs to pay for it.
Today, I came across the security application called 'Protected Folder' from IObit company, and while I was testing it something clicked in my head, so I decided to go deeper.
IObit company was founded back in 2004 and as they claim it *"is focused on providing consumers with innovative and comprehensive system utilities and security software for superior PC performance and security."*
Just to avoid the misunderstanding, this article is not targeting the company itself or all of their products, so no conclusions are going to be made for the rest of their software, apart from the software discussed in this article.

Protected Folder v1.1

At the time I was writing these lines, the version mentioned above was the latest one. Before proceeding I would like to focus for a while at what the company itself says about this software, in other words the way it presents it.

So, according to IObit Protected Folder v1.1 is/offers:

i) The most advanced and easiest privacy protection.
ii) Extremely easy to use.
iii) Safe and secure.
iv) No more Data theft, Data loss, or Data leaks.
v) Privacy Protection

In just a few words, IObit presents this software as reasonably secure. Well, that's what I liked, and maybe that's what made me dig more into it in the first place.
On the other hand, it is true that this application is very user-friendly and no particular knowledge or skills are required in order to work with it. However, is this the main reason why you would buy and use a security application?
So let's see how secure this software is after all…

Setting the Master Password

The first time you run this application it will require to set the master password, which is going to be used later for authentication in order for the user to be able to access the protected files and/or folders through this software.

Figure 1. Setting the Master Password

I decided to set as a password the word “secure”. Once the password is set we can have access to the main program.

Figure 2. The main application window

Protecting a dummy file for testing

In order to test its security levels, I created a text file with a content of my choice. Then, I made sure that all protection options were enabled, and finally I dragged and dropped it inside the main window of the application in order to protect it.

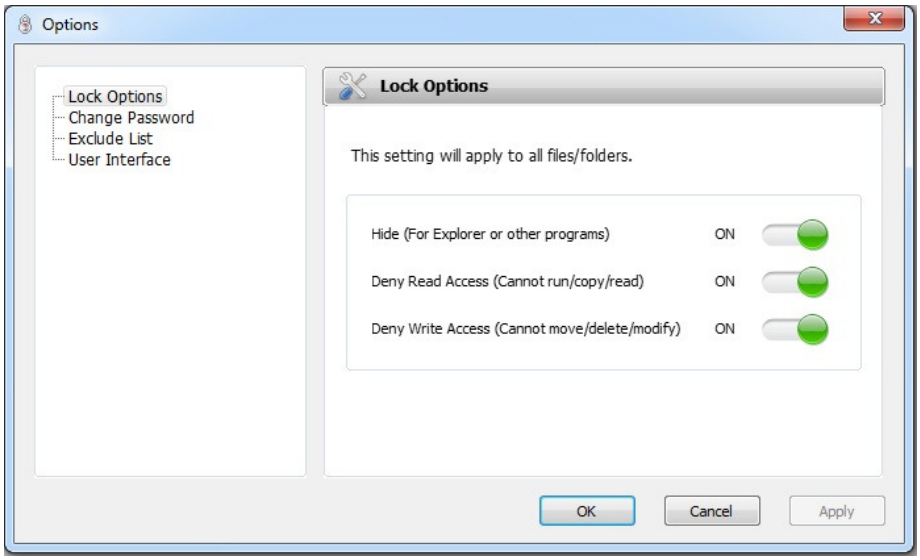

Figure 3. File/Folder Protection Options

The file is now protected, or at least that's what this application claims…

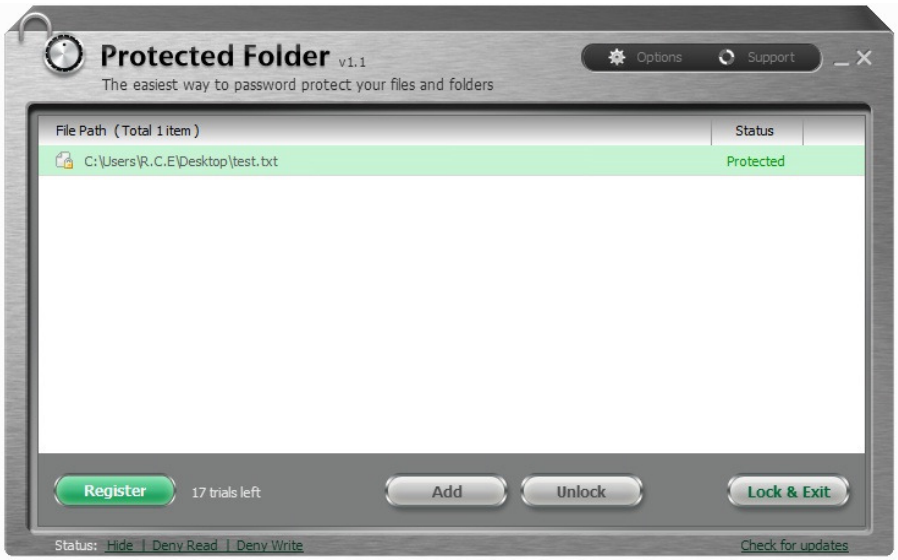

Figure 4. The file is now protected

The Protection Mechanism

In order to enable all these protection options we saw before, this application uses a driver called **'pfilter.sys'**, which apparently does exactly what its name indicates. I didn't analyse this file further because it is not necessary in this case, but most probably it hooks some functions at kernel level that have to do with file manipulation such as the **ZwCreateFile**, **ZwReadFile**, **ZwWriteFile,** native Windows APIs and in addition, it filters the parameters related with the protected file and the access rights requested.
For example, the ZwCreateFile API is used in order to obtain a handle to a file with some specific access rights, such as read, write, read|write etc…
So, depending on the protection options, the application is filtering the first API mentioned based on the absolute path of the file or folder, in other words if no access is permitted, it won't let Windows Explorer to obtain a valid handle to the file or folder so it basically hides it from the user.
On the other hand, if file or folder listing is permitted, then it will check and adjust the requested access rights to meet the protection options.

The Authentication Mechanism

The application is only using one password for authentication. This means that the user has to login only once to the application every time he starts it and then he has access through there to all the files and folders that were previously protected.

Figure 5. Authentication Request

In order to start analysing it, I tried at first with a false password while dynamically analysing it on runtime. I was particularly interested to see how it was going to verify that the correct password was entered.

I noticed that when I was trying to login, the application was successfully obtaining a handle to the following file:

CPU Stack
Address Value Comments
0012F82C /01584C84 ; |FileName = *"C:\ProgramData\IObit\Protected Folder\drawposs.db"*
0012F830 |80000000 ; |DesiredAccess = *GENERIC_READ*
0012F834 |00000000 ; |ShareMode = *0*
0012F838 |00000000 ; |pSecurity = *NULL*
0012F83C |00000003 ; |CreationDistribution = *OPEN_EXISTING*
0012F840 |00000080 ; |Attributes = *FILE_ATTRIBUTE_NORMAL*
0012F844 |00000000 ; \hTemplate = *NULL*

However, when I looked in that directory, I was unable to see this file. Clearly the driver used from this application is hiding this file, but why?
The application requested only read access to this file, but I wanted to have this file available too. I was just curious…what was wrong with that?

So my next step was to modify the parameters that the application is normally using in order to read this file.
I could also wait to "hook" the ReadFile API later on, but since I was really anxious to have this file in my hands, plus I didn't know if there would be a partial or a full file read operation, I changed the parameters as follows in order to have complete read/write access to the file:

CPU Stack
Address Value Comments
0012F82C /01584C84 ; |FileName = *"C:\ProgramData\IObit\Protected Folder\drawposs.db"*
0012F830 |C0000000 ; |DesiredAccess = *GENERIC_READ|GENERIC_WRITE*
0012F834 |00000003 ; |ShareMode = *FILE_SHARE_READ|FILE_SHARE_WRITE*
0012F838 |00000000 ; |pSecurity = NULL
0012F83C |00000003 ; |CreationDistribution = *OPEN_EXISTING*
0012F840 |00000080 ; |Attributes = *FILE_ATTRIBUTE_NORMAL*
0012F844 |00000000 ; \hTemplate = *NULL*

Once the valid handle was obtained, I made a few code and data modifications. First, I injected the following function:

CPU Disasm
Address Hex dump Command Comments
0045DF2B 60 PUSHAD
0045DF2C 6A 00 PUSH 0
0045DF2E 68 044E5801 PUSH 1584E04 *; UNICODE "C:\ProgramData\IObit\Protected Folder\drawposs.xx"*
0045DF33 68 844C5801 PUSH 1584C84 *; UNICODE "C:\ProgramData\IObit\Protected Folder\drawposs.db"*
0045DF38 E8 BA8B0A77 CALL CopyFileW
0045DF3D 61 POPAD

This function, I just injected, will attempt to make a copy of the **drawposs.db** file and name the copy as **drawposs.xx**, leaving the original one intact, which is hidden by the protection mechanism.
You can notice the data modification regarding the path to the new copy of the file on the code snippet above. Of course, I restored everything once my inline function was done.
The original file with the .db extension was still hidden, but I didn't really care anymore since I had an exact copy of it.

After this step, I had an exact copy of this hidden file, so I opened it with a hex editor, and I found inside the following sequence of bytes:
26 E5 89 3A 3D BE 24 D5 E4 2C B0 26 7A 18 CD 2D A8 80 46 39 88

Well, at this point I guessed that this could probably be some type of hash of the original password, but because of the unusual number of the bytes which was 21, I wasn't sure yet.

In fact, later on the execution reached the following loop which was transferring the first 20 bytes to another memory location:

CPU Disasm
```
Address   Hex dump          Command
00596A70  |> /8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
00596A73  |. |8B17          MOV EDX,DWORD PTR DS:[EDI]
00596A75  |. |0FB60410      MOVZX EAX,BYTE PTR DS:[EDX+EAX]
00596A79  |. |8B17          MOV EDX,DWORD PTR DS:[EDI]
00596A7B  |. |8882 5C925F00 MOV BYTE PTR DS:[EDX+5F925C],AL
00596A81  |. |FF07          INC DWORD PTR DS:[EDI]
00596A83  |. |833F 14       CMP DWORD PTR DS:[EDI],14
00596A86  |.^\7C E8         JL SHORT 00596A70
```

So maybe it is actually a SHA-1? Let's see…

Not much later, I came across some Windows Crypto APIs:

CPU Disasm
```
Address   Hex dump          Command                          Comments
005968A4  |.  E8 1BFEFFFF   CALL <JMP.&advapi32.CryptAcquireContextW ; Jump to advapi32.CryptAcquireContextW
```

CPU Disasm
```
Address   Hex dump          Command                          Comments
005968F7  |.  E8 E0FDFFFF   CALL <JMP.&advapi32.CryptCreateHash>     ; Jump to advapi32.CryptCreateHash
```

CPU Disasm
Address Hex dump Command Comments
00596914 |. E8 CBFDFFFF CALL <JMP.&advapi32.CryptHashData> *; Jump to advapi32.CryptHashData*

CPU Disasm
Address Hex dump Command Comments
0059693A |. E8 95FDFFFF CALL <JMP.&advapi32.CryptGetHashParam> *; Jump to advapi32.CryptGetHashParam*

CPU Disasm
Address Hex dump Command Comments
0059694F |. E8 98FDFFFF CALL <JMP.&advapi32.CryptDestroyHash> *; Jump to advapi32.CryptDestroyHash*

CPU Disasm
Address Hex dump Command Comments
0059695B |. E8 6CFDFFFF CALL <JMP.&advapi32.CryptReleaseContext> *; Jump to advapi32.CryptReleaseContext*

By analysing the parameters passed to **CryptCreateHash** API, I noticed that the parameter related the hash algorithm selection was 0x8004, which corresponds to SHA-1 according to MSDN:

CALG_SHA 0x00008004 SHA hashing algorithm. This algorithm is supported by the Microsoft Base Cryptographic Provider.

CALG_SHA1 0x00008004 Same as **CALG_SHA**. This algorithm is supported by the Microsoft Base Cryptographic Provider.

So, I was right that this was a **SHA-1** hash and to be more specific it was created using **UTF-16 (Little Endian) encoding**.
At this point it is very important to mention that the hashing was done for the password that the user entered during the authentication phase.

After that, it was trivial to locate the final comparison algorithm that verifies the input password by comparing the two hashes:

CPU Disasm
Address Hex dump Command Comments
0045CDD7 |. 8B1C01 MOV EBX,DWORD PTR DS:[EAX+ECX]
0045CDDA |. 3B1C11 CMP EBX,DWORD PTR DS:[EDX+ECX]
0045CDDD |. 75 72 JNE SHORT 0045CE51
0045CDDF |. 8D1C01 LEA EBX,[EAX+ECX]
0045CDE2 |. 83C1 04 ADD ECX,4

```
0045CDE5  |.  83E3 03       AND EBX,00000003
0045CDE8  |.  29D9          SUB ECX,EBX
0045CDEA  |.  7F 2E         JG SHORT 0045CE1A
0045CDEC  |>  8B1C01        MOV EBX,DWORD PTR DS:[EAX+ECX]
0045CDEF  |.  3B1C11        CMP EBX,DWORD PTR DS:[EDX+ECX]
0045CDF2  |.  75 5D         JNE SHORT 0045CE51
0045CDF4  |.  8B5C01 04     MOV EBX,DWORD PTR DS:[EAX+ECX+4]
0045CDF8  |.  3B5C11 04     CMP EBX,DWORD PTR DS:[EDX+ECX+4]
0045CDFC  |.  75 53         JNE SHORT 0045CE51
0045CDFE  |.  83C1 08       ADD ECX,8
0045CE01  |.  7F 17         JG SHORT 0045CE1A
0045CE03  |.  8B1C01        MOV EBX,DWORD PTR DS:[EAX+ECX]
0045CE06  |.  3B1C11        CMP EBX,DWORD PTR DS:[EDX+ECX]
0045CE09  |.  75 46         JNE SHORT 0045CE51
0045CE0B  |.  8B5C01 04     MOV EBX,DWORD PTR DS:[EAX+ECX+4]
0045CE0F  |.  3B5C11 04     CMP EBX,DWORD PTR DS:[EDX+ECX+4]
0045CE13  |.  75 3C         JNE SHORT 0045CE51
0045CE15  |.  83C1 08       ADD ECX,8
0045CE18  |.^ 7E D2         JLE SHORT 0045CDEC
```

The algorithm above is part of a Boolean function that will return either 0 or 1 depending on the case. In other words if the two hashes match it will return 1, otherwise it will return 0.
At this point, it is quite obvious that we only have to force the function to return true and see what it will happen next…

So, we tricked the application to think that the correct password was entered. After that, I noticed that it was trying to obtain read access to the following file:

CPU Stack

```
Address   Value      Comments
0012FCDC  \0045DF2B  ; /RETURN from kernel32.CreateFileW to
ProtectedFolder.0045DF2B
0012FCE0  /015D670C  ; |FileName = "C:\ProgramData\IObit\Protected
Folder\ucpl.dat"
0012FCE4  |80000000  ; |DesiredAccess = GENERIC_READ
0012FCE8  |00000000  ; |ShareMode = 0
0012FCEC  |00000000  ; |pSecurity = NULL
0012FCF0  |00000003  ; |CreationDistribution = OPEN_EXISTING
0012FCF4  |00000080  ; |Attributes = FILE_ATTRIBUTE_NORMAL
0012FCF8  |00000000  ; \hTemplate = NULL
```

However, this was not important for the scope of this article, and the final goal which was gaining access to the protected file without knowing the original authentication password.

The authentication mechanism was successfully bypassed and we can now have full access to the protected file, even completely unlock it from the application.

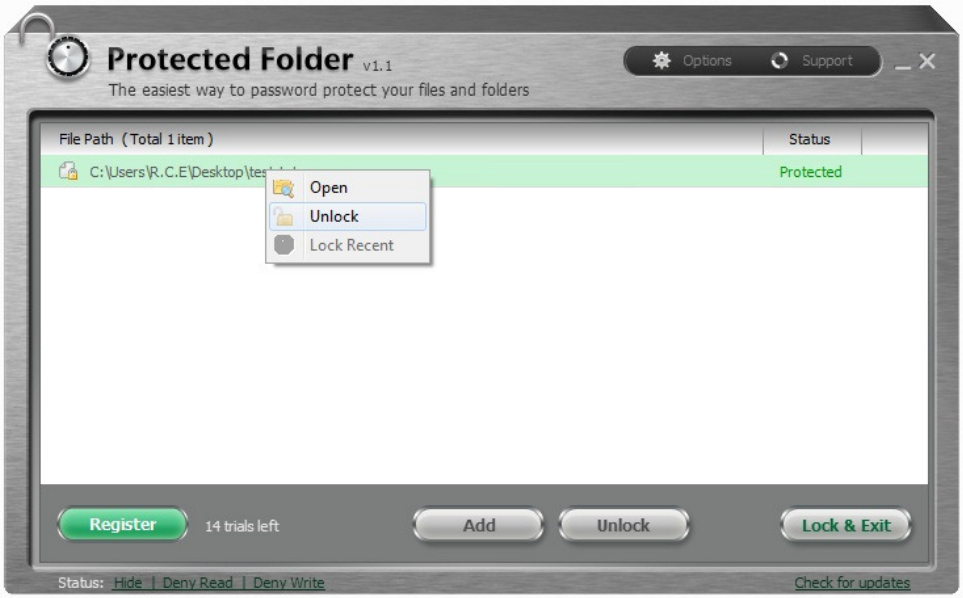

Figure 6. Authentication Bypassed

A couple of unpleasant scenarios…

Imagine the case in which a malicious user, instead of making a copy of the protected file that holds the SHA-1 hash of the original password, which he doesn't need to, since patching the algorithm is enough to have access to all the protected files and folders, he decides to use the same trick, but this time using the **WriteFile** API to substitute the SHA-1 with random bytes or with a another hash that corresponds to a password of his choice.
He could literally lock out the legitimate user and create a lot of trouble by denying the access to important personal or business data.
Another, scenario would be to modify the **exclude.cds** file which contains a list critical directories of the operating system that the user cannot lock, and add just "C:\". This would not affect the already protected programs, unless he unlocks them and tries to lock them back again, but the user wouldn't be able to protect anymore files or folders located under the C:\ directory.

Conclusion

Unfortunately, this application allows different types of attacks, going from the **Denial of Service** that we just mentioned to a **complete bypass of the authentication mechanism**.
We managed to successfully defeat this security related software, but we mostly had fun and learned something from this. I really hope that this article will be seen as useful feedback, also for the people that created this software.

I am not going to make any further comments to the quality and the effectiveness of this software, because I really believe that through this article I made my point clear, and because I would like you to make your own.