Writing a loader for an application packed with an unknown packer: the case of CDBank Cataloguer.

Shub-Nigurrath [ARTeam]

Version 1.0 – September 2005

1. Abstract ... 1
2. Target description ... 2
3. Target analysis ... 2
4. Cracking stage - Registering the application ... 4
5. Writing a debugger loader for the program, using Shub-Nigurrath's framework ... 5
   5.1. Approaching the problem ... 5
   5.2. Finding the SEH required to write the loader ... 7
   5.3. Resemble the patches ... 8
   5.4. Set the Victim Details ... 8
   5.5. Write the GateCondition ... 9
   5.6. Close the Loader and Hide the Debugger ... 11
   5.7. Run the Loader ... 11
6. References ... 12
7. Conclusions ... 13
8. History ... 13
9. Greetings ... 13

Keywords

Debugger Loader, Process, Debugging

1. Abstract

The question this tutorial tries to address is how to write a loader for an application which is packed with an unknown packer, what events to trace and how to proceed in order to faster get a working loader, able to patch the target.

I'm returning another time on loaders just because I felt this specific topic was not completely covered by existing tutorials. The loaders argument is very huge and there's always space left for some new things.
The tutorial will examine how to proceed writing a loader for an application and also which are the most common errors you might find writing them.

This time I will use my framework for loaders I already described in [1] and sources will also be released.
Of course the target application is only used as an example and thus we will not crack it completely, just to not transform this tutorial into a cracking tutorial, but leaving it as a studying document. You will have to complete on your own the way if you like it or just buy the program: developers deserves your money to continue to develop.

Have phun,
Shub-Nigurrath

2. Target description

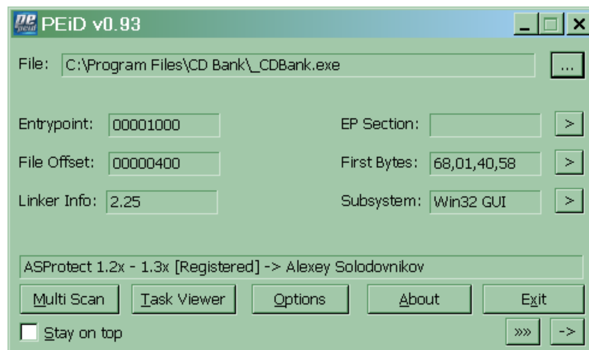
Some notes about the target:
CDBank Catalogue 2.7.0 build 222 – http://www.qunom.com
You may also get it at: http://intechhosting.com/~access/ARTeam/tools/cdbanksetup.exe

CD Bank Catalogue will help you maintain and organize your collection of CD-ROMs, audio CDs and DVDs. CD Bank's database makes the contents of your media available for offline browsing, searching, and copying on the hard drive. CD Bank is ideal in handling all your programs distributives, CD-R archives, music collections like MP3s or audio CDs, photos and graphics collections and documents, DVD and AVI disks.

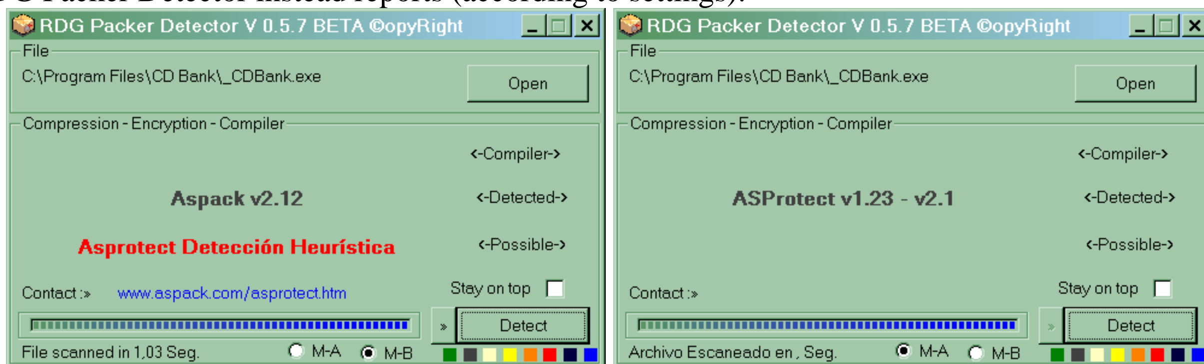
3. Target analysis

As usual we start looking at which type of compression/protection has been used for the distribution. We will see what some packer detectors have to tell about the target.

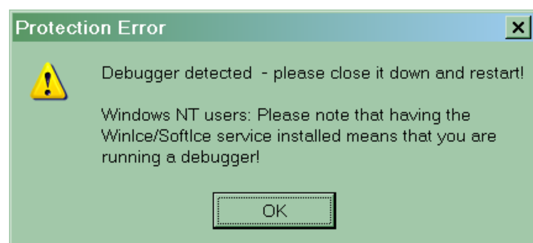
PEiD reports:



RDG Packer Detector instead reports (according to settings):



Well, apparently PEiD and RDG are reporting different things, but who of the two got the real used packer? Doing some more tests we get this error when trying to debug the application:



The application has some anti-debugging protection inside (you would easily discover it as far as you run a non hidden OllyDbg on it), but as far as I remember AsPack never had those protections. So at the end it seems like it might be an AsProtect target, but which version?

Let do another try with Stripper and see if it is able to unpack it: Stripper 2.11 is able to unpack the program while earlier version not. So apparently what PEiD reports, a version 1.2x, 1.3x, is not true because otherwise Stripper older versions should have been able to decompress it. It might be that RDG is the most correct detector, but these days we cannot ever be sure about what these packer detectors reports, because there are some known ways to fool their results (Execryptor is able to simulate signatures of other packers, FakeSigner[1] is another one).

NOTE
There are several ways to fool protectors' signatures. There are some out-of-the-shelf tools such as FakeSigner (www.dotfix.net), but also custom methods in tutorials such as "Killing PEiD detection Tutorial by KaGra" or other ideas such as to reproduce the code at the OEP of a packer or protector and put it in a new section. Then place your code there and change the EP to this direction. You just need to find out where you can place a jump to the OEP in this code without destroying the stack or the registers.

So generally we cannot trust too much these tools, it's always better to start from a blind position.
Ok we rounded around for too much now then it is time to open OllyDbg and see directly on your own what happens inside the program and see which surprises it's holding for us..

```
00401000   68 01405800   PUSH _CDBank.00584001
00401005   E8 01000000   CALL _CDBank.0040100B
0040100A   C3            RETN
0040100B   C3            RETN
```

Figure 1 - packed application entrypoint

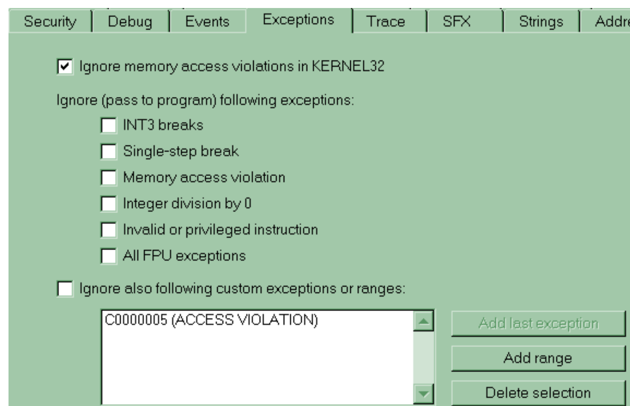


Figure 2 - OllyDbg settings to not ignore exceptions

The entrypoint looks like in Figure 1. Before running the application, take care to have all the exception handlers disabled in OllyDbg as in Figure 2: we are sure that OllyDbg will not pass any important exception to the program.
After a long series of exceptions we are not still recognizing any of the already known AsProtect signatures discussed in [2, 3]: not PUSH 0C or IRETD instructions .. we are not able to understand which exactly is the AsProtect version. But apteral it's not important because we want to find a general approach to such cases, where we are into a not clearly identifiable case (due to our own knowledge limitations or due to detectors tools limits).

Anyway it's time to take a decision of how you would approach this target:

1. do a full dump of the program, using an AUP Tool (e.g. Stripper) or doing a MUP[2]
2. write a loader able to patch the program in memory.

Of course the two ways differs: the first produces bigger distribution files and it's much easier (less than 1 minute) in case of AUP, while the latter is in my humble opinion much more elegant and less lame.

[1] www.dotfix.net
[2] Automatic UnPacking (AUP) vs Manual UnPacking (MUP)

We will approach the second one.

4. Cracking stage - Registering the application

In order to not waste time explaining how to crack the program the only thing I will show on this application is just a minor modification of the code, which will allow entering any serial you like. This modification will not be enough to crack the whole program but will be enough to write a loader for the application.
The patch itself is so easy that I will not explain how I found it, it should be clear if you are not just a beginner.

Launch the victim with OllyDbg and set the exception's settings as in Figure 3.

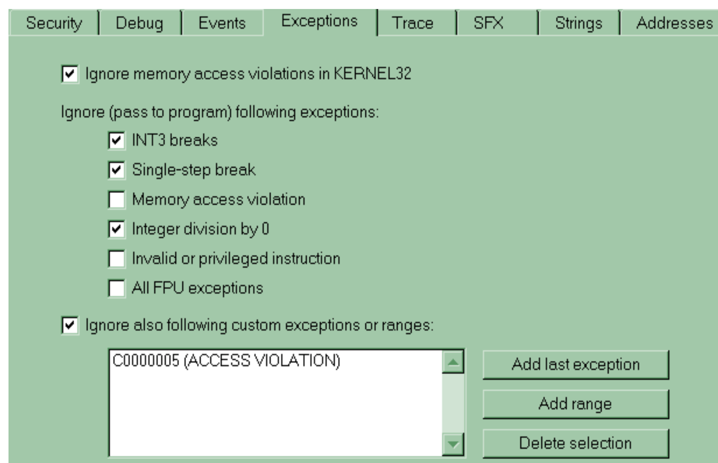
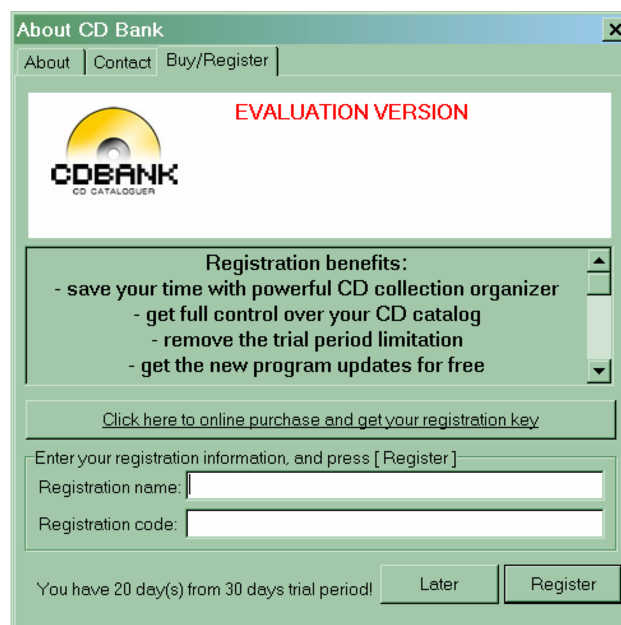


Figure 3 - OllyDbg Settings to ignore exceptions

Now let it run freely, pressing SHIFT-F9, till the main program's window appears. Go to the "About" menu voice and you should see the following registration dialog. Where you can enter any name (e.g a@b.c) and serial you like to get the bad boy message.



At this point press pause in OllyDbg and use ALT-K to see the Calls Stack: you should land at the top of the calls stack at 0x004FC7EC. Take a look at the routine (or trace it step by step after having placed a BP and after having inserted another serial) and you will easily see that there's a conditional jump which is showing us the bad boy message.

```
004FC8B2  |. /0F84 A0000000     JE _CDBank.004FC958
```

The patch our loader will be responsible to perform is the change of this JE to a NOP. So the patch will be:

Address	Original	Patched
0x004FC8B2	0F	90
0x004FC8B3	84	90
0x004FC8B4	A0	90
0x004FC8B5	00	90
0x004FC8B6	00	90
0x004FC8B7	00	90

NOTE

If this is the only limitation left or not or if it's the best way to crack the program it's not the point: this document is meant only for educational purposes and not to release any specific crack, so I chose to patch it this way because it's functional to the tutorial.

5. Writing a debugger loader for the program, using Shub-Nigurrath's framework

For the most specific parts of the framework used I would suggest reading [1].

5.1. Approaching the problem

Generally speaking the problem of writing a loader for a protected application isn't that simple, especially when it's protected with complex protectors such as AsProtect. These protectors are able to detect the debugger's presence, have anti-tampering techniques and decompress the program only at the right time (not speaking of the other tricks of the most advanced protectors such as execryptor or armadillo).

You have to satisfy these conditions (keep them in mind):

A. wait till the memory location you want to patch is unpacked by the protector's unpacker;
B. patch the memory location only **after** the integrity check is done;
C. patch the memory location **before** the instructions are executed or used by the target;

The idea is then the same already described into the tutorial [1]; write a debug loader, a sort of mini debugger, able to essentially do what you already do with OllyDbg.
For this reason our goal is to find the right condition, satisfied which our loader will safely write its patches to memory.

Technically speaking what described in [1] is a SEH debugger, a debugger which uses the Structured Exception mechanism to trap the debug events raised by the target's application or by the system. So what we have to find is a proper SEH which will satisfy the conditions A-C above said.
I would state that the more exceptions the packer raises the easier is the writing of a working loader!

Keep in mind also that you might fall into several loader failure cases:

1. you write the memory too early and the protector overwrites your patch with the just unpacked program's code;
2. you cannot write to a specific memory location due to page permissions;
3. the program is relocated (loaded by the windows' loader at a different base location) then you are indeed patching the wrong memory location;
4. the protector detects you and counteract in some ways what your loader is trying to do.

Generally speaking when creating a loader is a good thing to launch the program with debugging tracing from within the VC++ IDE (if you are using VC++) and pause just after the writing to memory of the patches. Then launch into another task a memory inspection tool such as WinHEX, open the target's process memory and see at the supposed patched location what's there. Using this approach often you can understand at least if you patched the memory location too early.

So our steps will be:

1. open the target in OllyDbg
2. place a memory on access breakpoint at the location we want to patch (0x004FC8B2);
3. configure OllyDbg to handle all SEH (as in Figure 3);
4. launch the application with F9;
5. stop at the breakpoint on reading at the patch location.

Doing this way we will get two stops: the first when the patch location is written by the unpacker (use CTRL-G to check that at the destination address the final code is present), the second one when the patch location is read from the packer to check the program's integrity.
Doing this way we will skip problems of type A and B (see before).

After this technically we are able to patch the program but we will have to wait for a characteristic SEH, which has to be unique and not already seen before. The concept is that the loader will check each SEH the target program generates till it sees the right one. The right SEH must have a unique pattern of course (see also following sections)

Quoting all the old tutorials speaking of good boy and bad boy messages, I would say that we are seeking the **Good SEH** ;-)

To complete this part with OllyDbg we have to:

1. configure OllyDbg to **not** handle all SEH (as in Figure 2);
2. press SHIFT-F9 to pass each exception to the program and let it stop in OllyDbg;
3. see if the exception where we landed had something "particular".

Dong this way we should able to see which the proper SEH is and where our loader will be able to patch the target.