



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 Cataloguer 2.7.0 build 222 – <http://www.qunom.com>

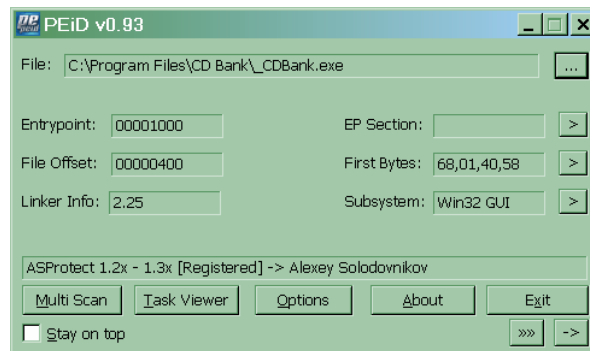
You may also get it at: <http://intechhosting.com/~access/ARTeam/tools/cdbanksetup.exe>

CD Bank Cataloguer 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.

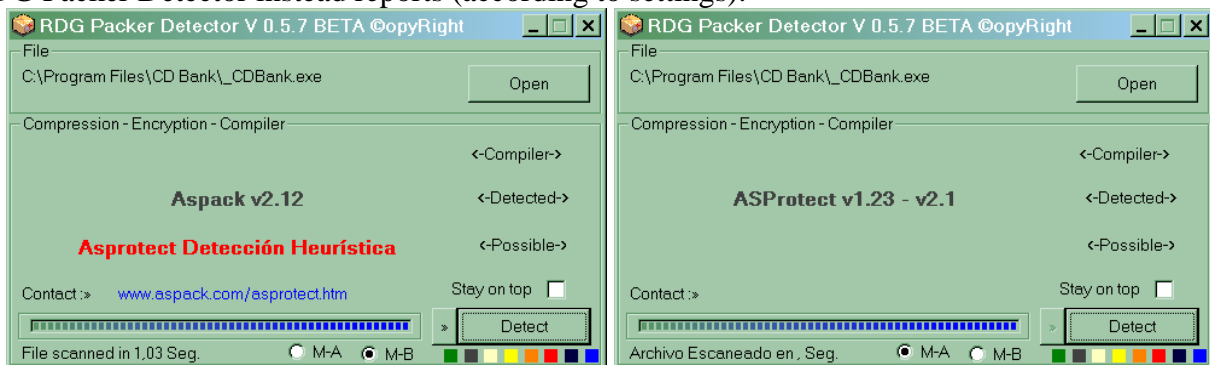
3. Target analysis

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.

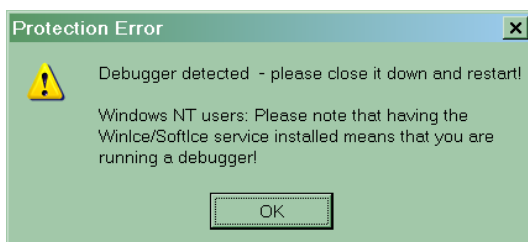
PEiD reports:



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, FakeSigner¹ 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.0040100E
0040100A	C3	RETN	
0040100E	C3	RETN	

Figure 1 - packed application entrypoint

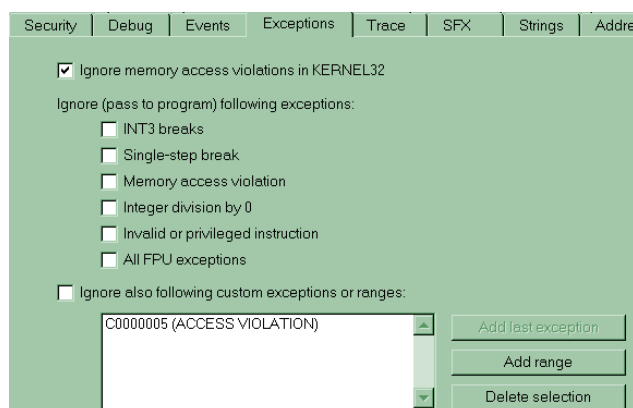


Figure 2 - OllyDbg settings to not ignore exceptions

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 MUP²
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.

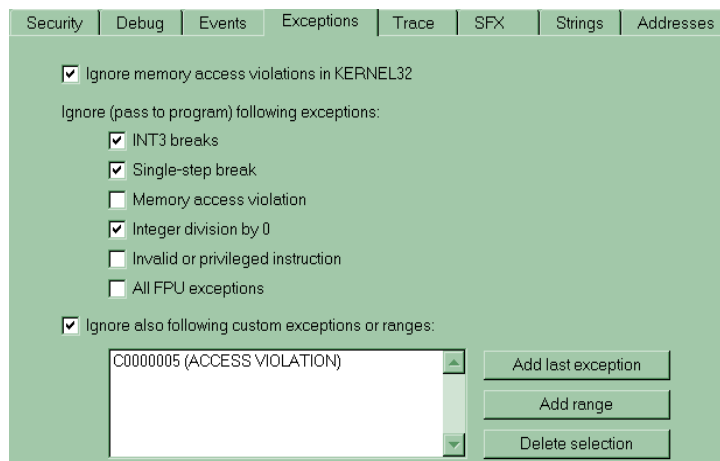
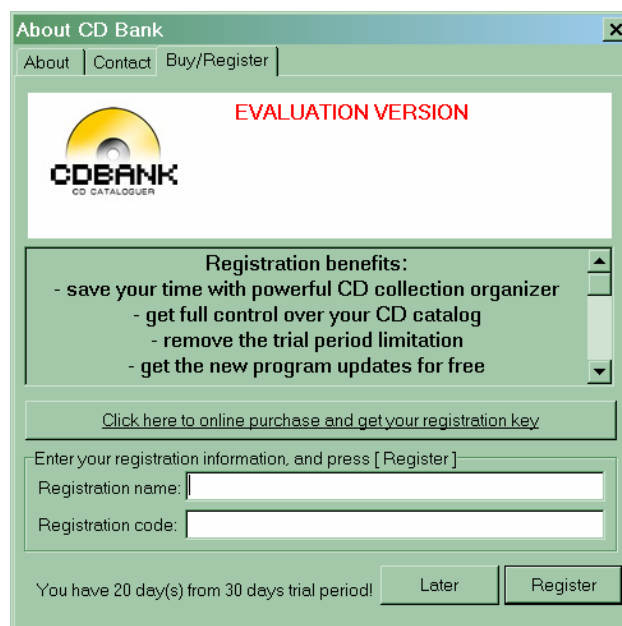


Figure 3 - OllyDbg Settings to ignore exceptions

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):

- wait till the memory location you want to patch is unpacked by the protector's unpacker;
- patch the memory location only **after** the integrity check is done;
- 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".

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



At the end we will also have to worry about hiding our loader: being a debug loader the target will detect it. Fortunately in [1, 2, 3] we already introduced the required code.

5.2. Finding the SEH required to write the loader

As told in previous section we are trying to find the proper SEH where the loader will be able to safely patch the victim's program.

I will see the steps briefly, letting you to in-deep experiment them directly.

1. Launch the application and immediately place a Breakpoint on Memory on Access at the patch location (use CTRL-G to 004FC8B2).
2. Configure OllyDbg to **ignore** all exceptions as told before and run the application pressing F9.
3. If you have properly configured OllyDbg you will stop at the location 00F32663:

00F32663	F3:A5	REP MOVSD WORD PTR ES:[EDI],WORD PTR DS:[ESI]	
00F32665	89C1	MOV ECX,EAX	
00F32667	83E1 03	AND ECX,3	
00F3266A	F3:A4	REP MOVSD BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]	
00F3266C	5F	POP EDI	
00F3266D	5E	POP ESI	
00F3266E	C3	RETN	
00F3266F	8D740E FC	LEA ESI,DWORD PTR DS:[ESI+ECX-4]	
00F32673	8D7C0F FC	LEA EDI,DWORD PTR DS:[EDI+ECX-4]	
00F32677	C1F9 02	SAR ECX,2	
00F3267A	78 11	JS SHORT 00F3268D	

where OllyDbg tells that there's a memory breakpoint when **writing**:

Memory breakpoint when writing to [004FC8B0]

4. Press F9 again to continue the application and you should land at the location

00F3D4F7	037A 30	ADD EDI,DWORD PTR DS:[EDX+30]	
00F3D4FA	03CF	ADD ECX,EDI	
00F3D4FC	8BF9	MOV EDI,ECX	
00F3D4FE	C1E7 03	SHL EDI,3	
00F3D501	C1E9 1D	SHR ECX,1D	
00F3D504	0BF9	OR EDI,ECX	
00F3D506	8BCF	MOV ECX,EDI	
00F3D508	8BFB	MOV EDI,EBX	

where OllyDbg tells that there's a memory breakpoint when **reading**:

Memory breakpoint when reading [004FC8B0]

5. Configure OllyDbg to **NOT ignore** all exceptions as told before and continue to run the application pressing SHIFT-F9 (pay attention because SHIFT-F9 passes the exception to the target, this is extremely important if the target is doing something important within the exception handler).
6. Continue pressing SHIFT-F9 till you see an exception which is somehow different from all the previous exceptions. For this example the situation happens few exceptions later at 00F459D7:

00F459D7	3100	XOR DWORD PTR DS:[EAX],EAX	
00F459D9	64:8F05 00000000	POP DWORD PTR FS:[0]	0012FF80
00F459E0	58	POP EAX	0012FF80
00F459E1	51	PUSH ECX	
00F459E2	52	PUSH EDX	
00F459E3	53	PUSH EBX	
00F459E4	56	PUSH ESI	
00F459E5	57	PUSH EDI	
00F459E6	FC	CLD	



where the pattern of bytes around the exception point is extremely different from all the other past and future exceptions.

7. Write down the instructions surrounding the exception you choose to use, you will use it later when writing the loader

5.3. Resemble the patches

Refer to [1] for details on the code. We are setting the final patch details already identified in Section 4, using the `InitializePatchStack` method exposed by the framework of [1].

```
<-----Code Snippet----->

//Receives
// - the Stack of Patch elements that must be properly filled
// - the victim file name, containing a valid path to the patched file
BOOL Loader::InitializePatchStack(growing_arraystack<Patch> &stkPatches)
{
    //NB 0x00 must explicitly converted to BYTE because otherwise the compiler confuses
    //it with a NULL pointer and doesn't know which constructor of class Patch to use.

    DWORD dwPatchaddrInj[IMAXINDEXINJ] = { 0x004FC8B2, 0x004FC8B3, 0x004FC8B4, 0x004FC8B5,
                                             0x004FC8B6, 0x004FC8B7 };
    int iOriDataInj[IMAXINDEXINJ] = { 0x0F, 0x84, 0xA0, 0x00, 0x00, 0x00 };
    int iPatchDataInj[IMAXINDEXINJ] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90 };

    int idx=0;
    for (idx=0; idx<IMAXINDEXINJ; idx++)
        stkPatches.push( Patch(dwPatchaddrInj[idx], (BYTE)iOriDataInj[idx],
                               (BYTE)iPatchDataInj[idx]));

    return TRUE;
}

<-----End Code Snippet----->
```

5.4. Set the Victim Details

Using the method `SetVictimDetails` exposed by the used framework, we will also set the target file system details: where the target is, its CRC and the parameters the loader uses to create the victim's process (there's not too much choice here, if you want to create a debuggable process).

```
<----- Code Snippet----->

//Simply used to specify the victim's filename, received the storing variable.
BOOL Loader::SetVictimDetails(TextString &victimFileName)
{
    #ifndef _DEBUG
        victimFileName=TextString(".\\_CDBank.exe");
    #else
        victimFileName=TextString("c:\\Program Files\\CD Bank\\_CDBank.exe");
    #endif

    //Set this parameter to true when you want the loader to check the CRC of the file!
    SetVictimCRC(0xe1ec0ccc);

    SetCreateProcessFlags(DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS | CREATE_NEW_CONSOLE);

    return TRUE;
}

<-----End Code Snippet----->
```




5.5. Write the GateCondition

```
<----- Code Snippet----->
//number of bytes to read around the EIP when the GateCondition is tested
#define BYTES2READATGATE 3

BOOL Loader::GateProcedure()
{
    BOOL bRet=FALSE;

    DEBUG_EVENT DebugEv; // debugging event information
    DWORD dwContinueStatus = DBG_CONTINUE; // exception continuation
    // Define the CONTEXT structure used to load the victim process context
    // when debugged process break due to exception event
    CONTEXT victimContext;
    int iExceptionCounter = 0;

    //will hold the code pointed by the EIP, plus 4 bytes forward.
    BYTE OridataRead[BYTES2READATGATE];
    int OridataExceptionCount=0;

    //reset the buffer
    memset(OridataRead, 0, BYTES2READATGATE*sizeof(BYTE));

    try {
        for(;;)
        {
            // Wait for a debugging event to occur. The second parameter indicates
            // that the function does not return until a debugging event occurs.
            // We are waiting for infinite time, then wait for each Debug Event.
            WaitForDebugEvent(&DebugEv, INFINITE);

            // Process the debugging event code.
            switch (DebugEv.dwDebugEventCode)
            {
                case EXCEPTION_DEBUG_EVENT: {
                    // Process the exception code. When handling
                    // exceptions, remember to set the continuation
                    // status parameter (dwContinueStatus). This value
                    // is used by the ContinueDebugEvent function.

                    // Increment exception counter (not used)
                    iExceptionCounter++;

                    #ifdef _DEBUG
                    // Show the current exception number
                    char str[256];
                    sprintf(str, "Exception number %d", iExceptionCounter);
                    ::MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
                    #endif

                    // Check if this is the right exception by reading the context
                    // structure for the victim process. Before to do it set the ContextFlags
                    // to READ_ALL
                    victimContext.ContextFlags = 0x1003F;

                    // Fill the process CONTEXT with the process information
                    GetThreadContext(GetPI()->hThread, &victimContext);

                    // Now I've to scan the process memory in order to see if I can patch the code!
                    // 00B159D7 3100 XOR DWORD PTR DS:[EAX],EAX
                    // 00B159D9 64:8F05 00000000 POP DWORD PTR FS:[0] ; 0012FF80
                    // 00B159E0 58 POP EAX ; 0012FF80
                    // 00B159E1 51 PUSH ECX
                    // 00B159E2 52 PUSH EDX
                    // 00B159E3 53 PUSH EBX
                    // 00B159E4 56 PUSH ESI
                    // 00B159E5 57 PUSH EDI
                    // 00B159E6 FC CLD
                    DWORD dwEIPOffset=0x09;

                    ReadProcessMemory(GetPI()->hProcess, (LPVOID)((victimContext.Eip) + dwEIPOffset),
                        OridataRead, BYTES2READATGATE, NULL);

                    //max index of the OridataRead must be BYTES2READATGATE-1
                    if ( (OridataRead[0] == 0x58) && (OridataRead[1] == 0x51) &&
```





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

```
(OridataRead[2] == 0x52) )
{
    // Key location found, now we can apply the patch
#ifdef _DEBUG
    char str[256];
    sprintf(str, "Found Gate location at %X",
        (LPVOID)((victimContext.Eip) + dwEIPOffset));
    MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
#endif

    throw TRUE;
}

//Gives out all the exceptions to the target program
//because we don't need any specific elaboration
dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;
}

case CREATE_THREAD_DEBUG_EVENT: {
    // As needed, examine or change the thread's registers
    // with the GetThreadContext and SetThreadContext functions;
    // and suspend and resume thread execution with the
    // SuspendThread and ResumeThread functions.
}
break;

case CREATE_PROCESS_DEBUG_EVENT: {
    // As needed, examine or change the registers of the
    // process's initial thread with the GetThreadContext and
    // SetThreadContext functions; read from and write to the
    // process's virtual memory with the ReadProcessMemory and
    // WriteProcessMemory functions; and suspend and resume
    // thread execution with the SuspendThread and ResumeThread
    // functions. Be sure to close the handle to the process image
    // file with CloseHandle.
    dwContinueStatus = DBG_CONTINUE;
}
break;

case EXIT_PROCESS_DEBUG_EVENT: {
    // Target Process is closed from user, then we have
    //to stop the debugger work and exit from loader
    // Exit form loader
    ContinueDebugEvent(DebugEv.dwProcessId, DebugEv.dwThreadId, DBG_CONTINUE);
    throw FALSE;
}
break;
}

// Resume executing the thread that reported the debugging event.
ContinueDebugEvent(DebugEv.dwProcessId, DebugEv.dwThreadId, dwContinueStatus);
} //end for(;;)
} //end try
catch(BOOL bRet) {
    return bRet;
}
}
<----- End Code Snippet----->
```



The overall structure should be familiar to those who read document [1], but anyway I would like to underline (see arrows in the code) few points:

1. We used the exception identified in Section 5.2. What makes it unique are the instructions at EIP+09, those POP and PUSHes: it's something you have not seen in previous SEH and als something you will not anymore see in future SEH, so in this sense it's unique, because the bytes surrounding the address where the exception breaks are not met for other similar breaks.



2. I used the first three instructions (bytes) read starting from EIP+09 to test the `GateCondition`. When these bytes match those we identified for the **Good SEH** the `GateCondition` throw a `TRUE` value (1) which is catch by the catch at point 3. The function then returns `bRet` which is equal to the value thrown, thus equal to `TRUE` (which is 1). I used 3 bytes after the EIP+09 point just as any other option. The point is: select the minimal pattern which the loader has not seen before till now, around an exception break.

All the other instructions are required to handle all the other debug events and are not specific of the target application; they have been anyway very deeply explained in [1], so instead of repeating them here I would suggest to read that document also.

5.6. Close the Loader and Hide the Debugger

As also explained in [1] we also worried to hide the debugger using the framework's method and to detach it from the program (if supported by the system) once the patch has been performed.

```
<-----Code Snippet----->
BOOL Loader::ActionsAfterGateProcedure()
{
    //Stop debugger action and let program run freely
    DWORD dwProcessId = GetProcessId(GetPI()->hProcess);
    BOOL bDbgStopFlag = DebugActiveProcessStop(dwProcessId);

    return TRUE;
}

//This function is called just before the process has been created but it is still in waiting mode
BOOL Loader::ActionsAfterCreateProc()
{
    HideDebugger(GetPI()->hThread, GetPI()->hProcess);
    return TRUE;
}
<-----End Code Snippet----->
```

Note that the `DebugActiveProcessStop` called by the function above is not the real Windows API (which is available only since Windows XP), but rather the method exposed by the class `NTInternals` (see [1]) which also ensure compatibility with Windows 9x/NT/2000. When the system is before WinXP it will simply do nothing at all.

`HideDebugger` on the other hand is a custom function which accessing the PEB masks the being-debugged state to the process owning the given thread (so to any process). I have already described it in details in [1] so I will not repeat myself here.

5.7. Run the Loader

Resuming, so far we did these important steps:

1. understood roughly what packer has been used for the target and understood that it's one of the AsProtect versions
2. found the patch to do
3. found the SEH schema the target follows and the most useful SEH where the loader will be free to patch (not too early, not too late)
4. hidden our loader to the debugger presence checks of the target
5. wrote the full loader using the framework already introduced in [1]

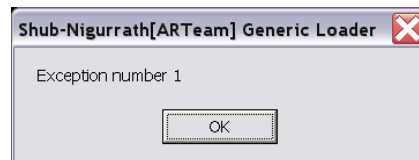
Now it's time to run our little creation and see if it's working properly.



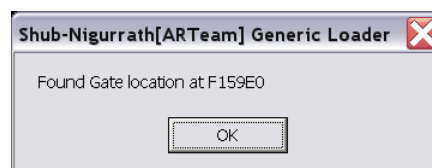
Note that if you compile the loader in “Debug Mode” the framework behaves differently being more interactive and telling you what’s happening under the hood.

Try to run on the target program and you would see:

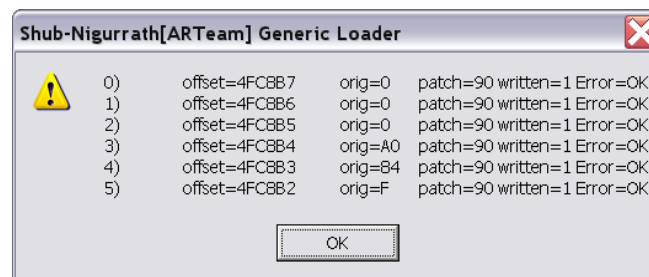
1. an exception counter dialog for each exception matched



2. a messagebox telling where the program matched the right exception (you should note that the address is the same as in Section 5.5, considering the required process’s relocations).



3. a final messagebox reporting the applied patches, the address, the final error status (OK if all went fine).



If you compile the same thing in Release Mode the patcher will run silently³ and detaches itself at the end, using `DebugActiveProcessStop` (if supported by the system), before the main program’s window appears.

6. References

- [1] “Cracking with Loaders, theory, methods and a framework”, Shub-Nigurrath, ThunderPwr, ARTeam, <http://tutorials.accessroot.com>
- [2] “Writing Loader 2 Patch Apps Protected With Asprotect 2.0 V10”, Shub-Nigurrath, Thunderpwr, ARTeam, <http://tutorials.accessroot.com>
- [3] “Writing Loader 2 Patch Apps Protected With Asprotect 1.2x And Earlier V10”, Shub-Nigurrath, Thunderpwr, ARTeam, <http://tutorials.accessroot.com>

³ Note that in release mode the Visual Studio project included has a post-build action that you might want to tune. The post built actions are commands issues after the build terminates. The command is:

```
"upx.exe" "$(TargetPath)" -9 --force -q --compress-icons=1 --all-methods
```

which means that the program just after compiled is packed with UPX at it best.



7. Conclusions

Well this is the end of the story. I tried to describe the process you should follow (or which I follow) when a new or unknown packer comes across your PCs and you want to write a loader to patch it.

Moreover the loader we created is quite simple because doesn't handle complex situations such as:

- multi-threaded applications: the target used is single threaded, or there's anyway a thread which is executes almost all.
- debug-blocker or other nice things such as nanomities and so on: the target application is protected with a one-shoot protector, a protector that once unpacked the application on memory essentially stops its action. The protector is only responsible to rebuild the whole application in memory and to protect this activity from intrusions somehow. Once the application is running most of the protections are gone. The only things left at the execution stage are static protections such as modifications of the calls, modifications of the ASM, but nothing that seriously not allow an attacker to modify the application. Protections such as nanomities (just to tell the most known) are more advanced in this sense and require more complex loaders.

All these complex things will be left for another tutorial which I will release one of these days..till then have phun!

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.

8. History

- Version 1.0 – First public release!

9. Greetings

I wish to tanks all the ARTeam members of course and who read the beta versions of this tutorial (thank a lot mates),... and of course you, who are still here at the end of this document!



<http://cracking.accessroot.com>