# Cracking, The Anti

Dorian Bugeja
*Department of Computer Science*
*and Artificial Intelligence*
*University of Malta*
*Email: dbug0009@um.edu.mt*

## Abstract

*This paper will describe some techniques used to protect an application from being reversed and cracked. It will contain information and tricks about algorithms mostly used in the security industry. It is very important to have some kind of custom protection add-on to an off-the shelf protector to prevent generic unpacking, therefore we will discuss some anti-debugging, anti-disassembling and anti-dumping tricks and methods such as nanomites and special APIs offered by the Operating System. When looking to many protectors, we will notice that there are mainly three steps it will undergo to protect the executable file, therefore to understand those procedure, various known packers/protectors will be discussed during this paper, describing their capability to get the general idea. We will take a look at win32 protectors which includes but not limited to the Yoda Protector and Armadillo. We will see how these protectors uses particular functions available by the OS and how executable files work within the system to construct new algorithms. Therefore the reader should have a basic knowledge of the Portable Executable (PE) and what threads are. Assembly language will be used throughout the paper to describe the algorithm of the technique being discussed. Last but not least, we will take a look at the main application used in the cracking scene such as OllyDbg and Imprec. This will give us a lot of information on how to disable/fool them, since we will be able to understand how they work.*

## Index Terms

*Software Protection, Anti Cracking, Protectors, Win32 PE, Unpacking*

## 1. Introduction

Software cracking is defined as the modification of binary files to remove protection. This means that firstly, the cracker has to locate the part where the software decide the mode it is operating - trial or full - and than, being able to modify and store the executable file containing the new bytes that were modified. In this paper, we will be going thought many techniques used into today's software protection industry.

When constructing an Anti-Cracking solution, the best way is to design a system keeping in mind the step a cracker will undergo to defeat the protection. The tools chosen by the cracker will play a major role as some algorithm will work on one software, but won't on another. Firstly, we will take a look at the largely used tools in this community.

It is important that the software coder gives away the minimum amount of information. Procedure names should never have a real meaning. Names like *IsValidSerial* and *IsRegistered* should be avoided. Such names could be easily searched using a binary string. Specific functions should not be used when writing a protection scheme, since such procedure can just be disabled and changed to return a true value, without going through the actual procedure. Another important thing to avoid is to use null-terminating strings to notify the user that he successfully registered the product, especially when using MessageBoxA from the Windows API.

### 1.1. How is Software cracked?

Software, as explained above, is cracked by modifying particular piece of code. The best way to understand what cracking involves is using an example.

Let's take this code as an example

```
if ( a == 4 )
    b = 1
else
    b = 0
endif
```

This code is converted into this

```
0040105D  CMP EBX,4  ;
Compare a with 4
00401060  JNZ SHORT 0040106E ;
Jump if not zero
00401062  MOV DWORD PTR DS:[4030E4],1 ;
Set a to 2
0040106C  JMP SHORT 00401078 ;
Skip next instuction
0040106E  MOV DWORD PTR DS:[4030E4],0 ;
Set a to 0
```

If a is 4, then b is true, else b is false. Just a small change at address 00401060, from JNZ to JZ ( Jump is zero ), will always return b as true unless a is 4. If a is our serial number, we will be able to register our product with every serial number, unless the serial number is correct. A simple algorithm for an anti-cracking solution is to hide this procedure as much as possible. Checking its integrity to detect any modifications will follow next.

### 1.2. Self Modification

Self modification algorithms are used to encrypt, and therefore hides the procedure from debuggers and disassemblers. Usually, a XOR algorithm will be used twice on the same data. When XOR is applied on a value, X, using Y as the other variable, stored in Z, the function will return X when Z is XORed with Y.

### 1.3. Checksum

Checksum are really important to detect any kind of modification. A checksum is an algorithm that adds data into a variable, and usually comparing them to hard coded values. Even the easiest checksum that adds up bytes in series can be effective, and it's easily implemented.

## 2. Anti-Debugging

### 2.1. Introduction

The most precious thing for a cracker is the debugger. There are mainly 2 types of debugger used, SoftICE, which is a ring0 debugger and mainly used in kernel-level debugging and OllyDbg, a ring3 debugger used in user-level applications. Checking for those tools at irregular intervals does help a lot, since the cracker will get annoyed by unexpected crashes. After all, software protection is all about annoying the cracker, since every piece of software can be cracked in a timely fashion.

### 2.2. SoftICE

SoftICE can be detected in various ways. Being a kernel-mode debugger, the easiest way is to open the driver installed by debugger. This can be done by opening the driver named *SICE* using *CreateFileA* from Windows API. We can detect any type of debugger that acts as a driver using the code below, as long as we know its name.

```
hFile = CreateFile( "\\\\.\\SICE",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ |
        FILE_SHARE_WRITE, NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL);
```

Instead of *SICE*, we can use different names such as *SIWVID*. Those drivers are installed on a Win9x system. To detect SoftICE on WinNT, *NTICE* should be used. Another way to detect is by using a back-door inside *SoftICE*. By sending an INT 3, with 'BCHK' into register EBP and value 4 into AX, if the debugger is present, 4 should be the return value inside AL.

```
MOV EBP,4243484B
MOV EAX,00000004
INT 3
```

### 2.3. Driver Based

The first method described in Detecting SoftICE, can be used to identify other special debugger like *Filemon* and *Regmon*. Such debugger are used to sniff and logs file operations and registry operations respectively from every running process. Therefore, such tools could identify the location the specific process is accessing ( serial number ). Most driver based debugger used has

```
SICE
SIWVID
NTICE
REGSYS
REGVXG
FILEVXG
FILEM
```

```
TRW
ICEEXT
```

### 2.4. OllyDbg

*OllyDbg*, being a user-mode debugger, can be detected by getting a list of running processes and compare the process name with ollydbg.exe. FindWindow, using OLLYDBG as a parameter is an alternative way to identify *OllyDbg*. Reading a number of bytes at a specific location - usually from the Code Segment ( CS ) - from every process and comparing them to previously store values could be a tougher task for the cracker to identify, since no strings are being used. Strings are always easy to find.

```
hWindow = FindWindow(0,"OLLYDBG");
```

A vulnerably found in the way *OllyDbg* handles string from *OutputDebugString* could lead to a crash from *OllyDbg*. By passing a multiple number of *'%s'* to *OllyDbg*, the message passed is showned into OllyDbg's status bar, which will lead to an invalid memory error. Such exploit will allowing remote execution.

```
OutputDebugString("%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s")
```

### 2.5. Special API

The Windows API comes with a set of function that are able to detect if an application is being debugged or not.

1) IsDebuggerPresent() returns true if the application is being debugged. The problem with such method is that the function is easily traceable and by changing the return value to 0, such protection will be bypassed. So we could re-create this function into our program. All this function does is accessing the Process Environment Block ( PEB ), and check byte 2 of the block. In such a way, we got ride of the *IsDebuggerPresent* import, with same results.

```
mov eax,dword ptr fs:[30]
cmp byte ptr ds:[eax+2],1
je debuggerActive
```

2) NtQueryInformationProcess() has the same capability. At class 7, a dword pointer will contain a non-zero value if a ring3 debugger is active. More functions are available by the operating system to detect debugger which we will not go through them due to the limit space on this paper. However, one should do some research on *CheckRemoteDebuggerPresent* and *NtQueryObject*

3) OutputDebugStringA() When a debugger is active, some functions change their behaviour. As described above, *OutputDebugStringA* will output a string to a debugger. If the debugger is present, and the string is caught by the debugger, the return value will be a non-zero value. One could check the return value of *GetLastError*, which should give back an error in case a debugger does not catch the string.

```
push f
call OutputDebugStringA
call GetLastError
test eax, eax
je activeDebugger
f : db "1",0
```

### 2.6. CloseHandle

The same concept can be used for *CloseHandle*. If an invalid handle is given as a parameter and a debugger is present, an exception ( 08h ) will occur. Such expection could be detected using a handler such as Structured Exception Handler ( SEH ). In such a scenario, where the debugger is present, the *CloseHandle* will not return an error, since the exception was caught by the debugger.

### 2.7. GetTickCount

Another trick widely implements makes use of the *GetTickCount* function. By calling such a function twice at regular internals, the debugger will be detected if the cracker is going through the code using single step since the difference in time passed from one execution of the *GetTickCount* function to the other one is greater, compared to the result of a normal running process.

```
call GetTickCount
xchg ebx, eax
call GetTickCount
```

```
sub eax, ebx
cmp eax, 1
jnb activeDebugger
```

### 2.8. SuspendThread

If a process is found to be the son of another process which is not "'Explorer.exe"', some can conclude that the process is attached to another software, probably a debugger. This isn't always the case. Some packers such as *Yoda's Protector* Suspends the main thread of the parent process ( the Debugger ). This is a very affective function in ring3 debuggers such as OllyDbg.

### 2.9. SetInformationThread

"Windows 2000 introduced an explicitly anti-debugging API extension, in the form of an information class called HideThreadFromDebugger. It can be applied on a perthread basis, using the ntdll SetInformationThread() function.

```
push 0
push 0
;HideThreadFromDebugger
push 11h
push -2 ;GetCurrentThread()
call NtSetInformationThread
```

When the function is called, the thread will continue to run but a debugger will no longer receive any events related to that thread. Among the missing events are that the process has terminated, if the main thread is the hidden one."

### 2.10. TLS-callback

This anti-debug trick was not so well-known a few years ago. The first protector using it was ExeCryptor. Code found in the TLS entry is executed before the instructions found at the entrypoint. The TLS entry can then perform anti-debug checks in a stealthy way. Old debugger like OllyDbg are note aware of TLS-callbacks, therefore they are not detected by them.

## 3. Breakpoints

When debugging, breakpoints can be very handy. They are divided into 2, hardware and software breakpoints. Both of them can be neutralized.

### 3.1. Hardware Breakpoints

Such breakpoints are tougher to detect than software breakpoints. A hardware breakpoint can be seen as a fault by a hardware. All information about those breakpoints can be found inside the *Debug Register*. So, by generating an exception, and telling the *KiUserExceptionDispatcher* the address at which the exception should be handle, we can reset every value inside the *Debug Register*. To return the breakpoint address, one should get the values inside DR0-DR3 instead of resetting their values. Only 4 Hardware breakpoints can be registered inside the register. To overcome this limitation, software breakpoints are used when debugging.

### 3.2. Software Breakpoints

Those breakpoints uses special opcode that the debugger will track when executed. For example, the opcode 0xCC ( INT3 ) is used in *OllyDbg* and many other debuggers. The execution of the program will stop as soon as that opcode is executed. This means the such opcode must be inserted in the program code. Therefore, by checking the integrity of the code, we can detect software breakpoints.

Many anti-cracking solution continuously check for API breakpoints. By getting the address location of the function, the first byte is compared with 0xCC. If true, it can be concluded that a breakpoint is present. Such method will only work if the breakpoint is toggled at the beginning of the function.

We can use WriteProcessMemory to remove breakpoints. By overwriting a specific section of the code using predefined data, any modifications done on the original code is changed back to normal. Therefore, the INT3 command is removed from that specific location.

Another way to check for breakpoints is the checksum method described above. Such procedure could be used to detect breakpoints while debugging, and to detect any code manipulation.

## 4. Anti-Dumping

### 4.1. Introduction

Anti-cracking solutions does more than detecting an active debugger. In this section, we will be looking at special techniques used by the packers to protect

the original code, but mostly, to deny any kind of modifications to the executable file. The packer will usually compress and encrypt all the executable file, and creates a new executable file with the algorithms needed to de-encrypt into memory. The working EXE is never stored onto the hard disk, therefore, the cracker must read data directly from the main memory and store the working executable onto the hard disk. Such process is called Dumping. We will be looking mainly at algorithms used by Armadillo and ASProtect.

### 4.2. Code Splicing

Code Splicing copy some of the code from the original executable file, but are not de-encrypted into the code section of the original software. Instead, the packer will create a new memory block, and place the code into this area. When in need to access this data, the packer will jump to the start address of the allocated memory. Dumping software such as LordPE does not get data outside the memory location allocated by the PE loader. Another feature added to this function makes it harder to predicate the location of the allocated data since the starting block vary from one execution to another. This technique was introduced in ASProtect, usually referring to it as Stolen Bytes.

### 4.3. Debug-Blocker

In Armadillo, we find another feature called Debug-Blocker. Armadillo creates 2 processes, referred to them as father ( or parent ) and child. The father process acts as a debugger, trying to protect the child from other debuggers. 2 Ring3 debugger cannot debug one process, therefore, the debugger used by the cracker cannot be attached to the running process. Such process was easily defeated by the crackers, by attaching their own debugger and closing the father process at the right time. Therefore, the next step for Armadillo was to introduce for the first time nanomites.

### 4.4. Nanomites

Such function was introduced so that the father process will be of some importance. While generating the packed EXE, Armadillo will scan for conditional jumps through the original executable. A table is generated and stored into the father's process. The jumps effecting the table are changed to an INT3 instruction ( which acts as a software breakpoint ). So now, some important instructions are not even located into the child process, but they are stored inside the father process. At every INT3 that occurs, the father process will look inside the generated table to retrieve the original code.

### 4.5. CopyMem

Guard Pages were introduced in Shrinker. Armadillo had integrated a variation of such algorithm which is referred to Copymem II. By using this function, the code is never uncompressed completely inside the memory. Instead, data is uncompressed in pages. This will decrease loading time since the code is not de-encrypted as soon as the program is executed, but only when the required page is needed.

### 4.6. Import Address Table

Almost every packer tries to hide the Import Address Table ( IAT ). Import function reveals a lot about the application. For example, if no RegQueryValue is found inside the IAT, the cracker can conclude that such program does not use the Windows Registry. Therefore, the serial or information about the trial period is stored into a file. Windows API provides enough functions to dynamically load the DLL and retrieve the starting address of the function required. LoadLibrary and GetProcAddress will load the DLL and gets the function starting address at runtime.

### 4.7. Original Entrypoint

Another common procedure is to hide the Original Entrypoint ( OEP ). The Entrypoint ( EP ) of the packed executable differs from the OEP. If the OEP is not found, the dump process will not work, since some code is bypass or some from another function is executed before starting, usually resulting into an illegal instruction.

## 5. Conclusion

As we shown during this paper, there are many different types of anti-unpacking techniques. We saw that new methods are being discovered over and over again. In a near future, we will be dealing with dongles. It is impossible to describe every protection used by every packer. This paper has tried to explain some anti-cracking algorithms, giving a beginner guide during the first section and ending into a more advance protect scheme.

## 6. References

ANTI-UNPACKER TRICKS by Peter Ferrie
http://pferrie.tripod.com/papers/unpackers.pdf

Windows Anti-Debug Reference by Nicolas Falliere
http://www.securityfocus.com/infocus/1893

Armadillo, Nanomites and vectored exception-handling by Greg Jenkin
http://www.ring3circus.com/rce/armadillo-nanomites-and-vectored-exception-handling/

A catalog of NTDLL kernel mode to user mode callbacks, part 2: KiUserExceptionDispatcher
http://www.nynaeve.net/?p=201

Debugger Breakpoints by Cristian L. Vlasceanu
http://www.zerobugs.org/debuggerbreakpoints

http://arteam.accessroot.com/ - General Reverse Engeneering Group

http://www.woodmann.com/crackz/ - General Reverse Engeneering Forums

http://www.tuts4you.com/ - General Reverse Engeneering Tutorials