

### PART III: MEMORY PATCHING HOW & WHEN?

ForWard:

After I have finished the second part of CIK I was in a pretty big dilemma: what should I talk about in CIK three? So I start thinking about the essay. Pretty soon I realized that an essay about memory patching would be the perfect one. Aldo memory patching is a pretty advanced subject, I say that a little theory wouldn't harm anyone (hope so).

The examples that I'll give here won't be very hard, but this is not the problem because it's good for you to have the basics about this cracking technique.

A little story will be told here so that you can understand that there is nothing to be afraid about memory patching.

Several months ago when I was just a very stupid cracker (not that I evolved in some way) I heard for the first time about the memory patching approach while looking throw fravia's old site. They were just two (maybe three) tutorials on this matter, so I quickly realized that only the gods of cracking master this method of cracking (I was only half-right). It was very strange for me to see that you could do make a little exe (a loader) that changed the way a target reacts to different commands. I was so fascinated that I spent a full night just reading those two essays. At the beginning I didn't understood a word. After that white night in my mind appeared the wish to learn assembly and advanced cracking. This is because, in order to make a loader you need to have some quite good knowledge of programming. But those two essays were just the basics for my need of knowledge. When I tried to find some other tutorials on memory patching, I failed. I had to find some things on my own (it went very slowly..)

Now that I know how the things work in the memory patching side of cracking, I want to share my knowledge with others, more or less newbies that are interested in this part of our art. Hope that everything is at clear as possible.

Let's start this advanced and long essay.

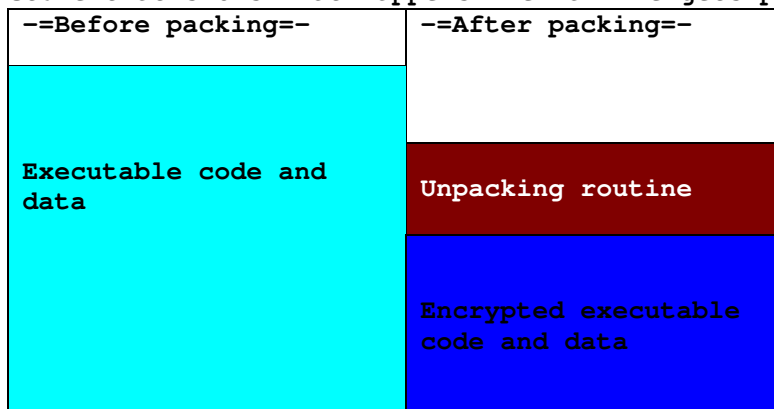
#### THEORY

##### 1. Some common ground on packers:

If a study would be made, it would show clearly that crackers are much more advanced in their techniques than protectionists are. This is true now, was true good years ago and will be true longer after us. This is because they think they are better then us, smarter. But there are outhere some very clever guys that fully realize our potential. So some years ago the concept of the packer and encryptor appeared. These tools do a very simple task: try to keep crackers away from the executable contents (through IDA or

W32dasm). This approach is successful when it deals with newbies, but fails when it comes to more advanced cracker.

This picture that shows what happens when a file gets packed:



Just as the previous picture showed, most of the times, after packing, the executable is much smaller in size. This is another advantage of commercial packers. This option is good for those lamers that code their "beautiful" programs in Visual Basic or any other overbloated programming language, instead of using the king: assembly language!! But that's another chapter of cracking (even programming) that we don't have the time to discuss here. (Maybe in a future edition of CIK, will see...).

So what does a packer??

Well, it first compresses the selected executable than it writes at the begin of the exe the unpacking routine. This unpacking routine will unpack the original code, so that it can run, as it would do normally.

In this crazy world, which is desperate in a good "protection", they are out there thousands of packers. Most of the authors promise the programmer that their applications will be much more resistant to crackers. But our advanced fellow crackers, driven by their hunger for knowledge, have studied very closely the way a packer 'does its job' and have come up with some anti tools =unpackers. So if a target you are trying to crack is packed, it's very probable that there is out there on the net some unpacker for that specific packer.

Back to our theme. As I explained, after packing the file has an unpacking routine and the encrypted executable code and data (like Import Table). So any code you might see in IDA or Wdasm is just trash. If you see only trash the option of classic patching is eliminated right from the start. So we need other methods.

## 2. Some common grounds on memory patching:

The whole idea behind memory patching is providing means of patching the target in it's addressing space, after a specific event has happened (after the target has been unpacked, after the CRC has been done..)

This would be a fairly easy thing to do if Windows wasn't such a preventive system. In windows every running program has it's one addressing space. This means that two different running processes can't interact one with other, and by interacting I mean that a process can't change directly something in the addressing space of the other one. So if we can't do it directly, will need to find some other way.

This "other" way consists in making a separate program that will become the parent of the process we want to patch. The parent means that it has all the rights on the process, like reading and writing. You will get what I mean when the examples will come (have a little patience).

Let's think for now at a possible scenario: you want to crack a target. After some live cracking in Sice, you find out that the jump that gives out the fact that you haven't registered your copy is at address 0040XXXXh. But when you try to disassemble the target, you see that the given disassembly is full of garbage instructions. This tells that you have in front of you a packed target.

Now that you know that your target is packed you have two options:

1. Find a unpacker for that specific packer, unpack the target and patch it as you would usually do...
2. Make a loader that will patch the target at runtime.

Well we will choose option number two. For number 2, if we have to be honest, we had another option: getting ourself a memory-patching generator, which will generate the loader for us. But all lamers can do this; it's not the case for us: we are true crackers!!

So this imaginary problem has lead us to the problem of memory patching.

To underline even more the need for some memory patching knowledge, let's think of another imaginary situation.

Just as in the first case, let's say you have to crack a hard target (hard working cracker!).

The target is not packed or anything, so you think your troubles are over. Quickly you find the places you want patch, you make a patcher. The patcher works flawlessly, but when you try to run the 'improved' target it just says that the executable is compromised (or something like that..). This little mishap tells us this: the target checks some checksum on itself, to see if it has been tampered with. If so it just refuses to run.

Just like in the first case, you have again two options:

1. Patch the code that makes the checksum
2. Make a loader that will patch the target at runtime, after the checksum check.

I hope that the given examples show the importance of memory patching. Those imaginary situations are just the tip of the iceberg, coz with memory patching you can do almost anything you want to, all you need is a bit of imagination :)

Now let's see the general approach for this two situations:

### I. How to patch a packed target

You'll find that this situation gets more and more common, because of those programmers that think that a lamer packer will get rid of crackers: deadly wrong my friend :(

Well we already said that in order to be able to change the data in the addressing space of the target we must have some rights like reading and writing. This example will show the most common way to memory patch a target.

Here are the speps for a begginer to learn:

- 1.Find the place to patch (fairly easy!)
- 2.Code the loader (quite hard for begginers but you'll get used after some practice!)

The loader will work like this:

- 1.Start the target we want to patch, hint:CreateProcess
- 2.Wait so that the target is fully initialized, hint: WaitForInputIdle
- 3.Patch the code, hint: WriteProcessMemory
- 4.Close the loader and let the improved target run

Now let's study the needed api's. Try to understand what they do and memoryze them (if you can :))  
So we need to make a loader.This loader will start the process, thus having all the rights on the child process and then will patch the target after it has been unpacked in memory.  
First we need to find the api that will start the process:

#####From Win32.hlp#####

The CreateProcess function creates a new process and its primary thread. The new process executes the specified executable file.

BOOL CreateProcess(

```
LPCTSTR lpApplicationName,      // pointer to name of executable module
LPTSTR lpCommandLine,          // pointer to command line string
LPSECURITY_ATTRIBUTES lpProcessAttributes, // pointer to process security attributes
```

```

LPSECURITY_ATTRIBUTES lpThreadAttributes,    // pointer to thread security attributes
BOOL bInheritHandles,      // handle inheritance flag
DWORD dwCreationFlags,     // creation flags
LPVOID lpEnvironment,     // pointer to new environment block
LPCTSTR lpCurrentDirectory, // pointer to current directory name
LPSTARTUPINFO lpStartupInfo, // pointer to STARTUPINFO
LPPROCESS_INFORMATION lpProcessInformation // pointer to PROCESS_INFORMATION
);

```

## Parameters

### lpApplicationName

Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and filename of the module to execute.

The string can specify a partial name. In that case, the function uses the current drive and current directory to complete the specification.

The lpApplicationName parameter can be NULL. In that case, the module name must be the first white space-delimited token in the lpCommandLine string.

The specified module can be a Win32-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

Windows NT: If the executable module is a 16-bit application, lpApplicationName should be NULL, and the string pointed to by lpCommandLine should specify the executable module. A 16-bit application is one that executes as a VDM or WOW process.

### lpCommandLine

Pointer to a null-terminated string that specifies the command line to execute.

The lpCommandLine parameter can be NULL. In that case, the function uses the string pointed to by lpApplicationName as the command line.

If both lpApplicationName and lpCommandLine are non-NULL, \*lpApplicationName specifies the module to execute, and \*lpCommandLine specifies the command line. The new process can use GetCommandLine to retrieve the entire command line. C runtime processes can use the argc and argv arguments.

If lpApplicationName is NULL, the first white space-delimited token of the command line specifies the module name. If the filename does not contain an extension, .EXE is assumed. If the filename ends in a period (.) with no extension, or the filename contains a path, .EXE is not appended. If the filename does not contain a directory path, Windows searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. Windows 95: The Windows system directory. Use the GetSystemDirectory function to get the path of this directory.

Windows NT: The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory. The name of this directory is SYSTEM32.

4. Windows NT: The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

If the process to be created is an MS-DOS - based or Windows-based application, lpCommandLine should be a full command line in which the first element is the application name. Because this also works well for Win32-based applications, it is the most robust way to set lpCommandLine.

#### lpProcessAttributes

Pointer to a SECURITY\_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpProcessAttributes is NULL, the handle cannot be inherited.

Windows NT: The lpSecurityDescriptor member of the structure specifies a security descriptor for the new process. If lpProcessAttributes is NULL, the process gets a default security descriptor.  
Windows 95: The lpSecurityDescriptor member of the structure is ignored.

#### lpThreadAttributes

Pointer to a SECURITY\_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpThreadAttributes is NULL, the handle cannot be inherited.

Windows NT: The `lpSecurityDescriptor` member of the structure specifies a security descriptor for the main thread. If `lpThreadAttributes` is `NULL`, the thread gets a default security descriptor.

Windows 95: The `lpSecurityDescriptor` member of the structure is ignored.

#### `bInheritHandles`

Indicates whether the new process inherits handles from the calling process. If `TRUE`, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

#### `dwCreationFlags`

Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted:

##### Value Meaning

###### `CREATE_DEFAULT_ERROR_MODE`

The new process does not inherit the error mode of the calling process. Instead, `CreateProcess` gives the new process the current default error mode. An application sets the current default error mode by calling `SetErrorMode`. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for `CreateProcess` is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.

###### `CREATE_NEW_CONSOLE`

The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the `DETACHED_PROCESS` flag.

###### `CREATE_NEW_PROCESS_GROUP`

The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the `lpProcessInformation` parameter. Process groups are used by the `GenerateConsoleCtrlEvent` function to enable sending a `CTRL+C` or `CTRL+BREAK` signal to a group of console processes.

###### `CREATE_SEPARATE_WOW_VDM`

Windows NT only: This flag is valid only when starting a 16-bit Windows-based application. If set, the new process is run in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications are run as threads in a single, shared VDM. The advantage of running separately is that a crash only kills the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that are run in separate VDMs have separate input queues. That means that if one application hangs momentarily, applications in separate VDMs continue to receive input.

#### CREATE\_SHARED\_WOW\_VDM

Windows NT only: The flag is valid only when starting a 16-bit Windows-based application. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is TRUE, this flag causes the CreateProcess function to override the switch and run the new process in the shared Virtual DOS Machine.

#### CREATE\_SUSPENDED

The primary thread of the new process is created in a suspended state, and does not run until the ResumeThread function is called.

#### CREATE\_UNICODE\_ENVIRONMENT

If set, the environment block pointed to by lpEnvironment uses Unicode characters. If clear, the environment block uses ANSI characters.

#### DEBUG\_PROCESS

If this flag is set, the calling process is treated as a debugger, and the new process is a process being debugged. The system notifies the debugger of all debug events that occur in the process being debugged. If you create a process with this flag set, only the calling thread (the thread that called CreateProcess) can call the WaitForDebugEvent function.

#### DEBUG\_ONLY\_THIS\_PROCESS

If not set and the calling process is being debugged, the new process becomes another process being debugged by the calling process's debugger. If the calling process is not a process being debugged, no debugging-related actions occur.

#### DETACHED\_PROCESS

For console processes, the new process does not have access to the console of the parent process. The new process can call the AllocConsole function at a later time to create a new console. This flag cannot be used with the CREATE\_NEW\_CONSOLE flag.

The dwCreationFlags parameter also controls the new process's priority class, which is used in determining the scheduling priorities of the process's threads. If none of the following priority class flags is specified, the priority class defaults to NORMAL\_PRIORITY\_CLASS unless the priority class of the creating process is IDLE\_PRIORITY\_CLASS. In this case the default priority class of the child process is IDLE\_PRIORITY\_CLASS. One of the following flags can be specified:

Priority	Meaning
----------	---------

HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is Windows Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.
---------------------	---



**IDLE\_PRIORITY\_CLASS** Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.

**NORMAL\_PRIORITY\_CLASS** Indicates a normal process with no special scheduling needs.

**REALTIME\_PRIORITY\_CLASS** Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

#### **lpEnvironment**

Points to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

name=value

Because the equal sign is used as a separator, it must not be used in the name of an environment variable. If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section. An environment block can contain Unicode or ANSI characters. If the environment block pointed to by lpEnvironment contains Unicode characters, the dwCreationFlags field's **CREATE\_UNICODE\_ENVIRONMENT** flag will be set. If the block contains ANSI characters, that flag will be clear.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

#### **lpCurrentDirectory**

Points to a null-terminated string that specifies the current drive and directory for the child process. The string must be a full path and filename that includes a drive letter. If this parameter is NULL, the new process is created with the same current drive and directory as the calling process. This option is

provided primarily for shells that need to start an application and specify its initial drive and working directory.

lpStartupInfo

Points to a STARTUPINFO structure that specifies how the main window for the new process should appear.

lpProcessInformation

Points to a PROCESS\_INFORMATION structure that receives identification information about the new process.

#### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

#####End win32.hlp#####

Well this function has a lot of parameters, but don't get scared coz most of them can be NULL  
Let's have a look at them again:

lpApplicationName	= pointer to name of executable module	
lpCommandLine	= NULL	// pointer to command line string
lpProcessAttributes	= NULL	// pointer to process security attributes
lpThreadAttributes	= NULL	// pointer to thread security attributes
bInheritHandles	= FALSE	// handle inheritance flag
dwCreationFlags	= NORMAL_PRIORITY_CLASS	// creation flags
lpEnvironment	= NULL	// pointer to new environment block
lpCurrentDirectory	= NULL	// pointer to current directory name
lpStartupInfo	= pointer to a STARTUPINFO structure	
lpProcessInformation	= pointer to a PROCESS_INFORMATION structure	

Now let's look at the two structures:

```
typedef struct _STARTUPINFO { // si
    DWORD   cb;
    LPTSTR  lpReserved;
    LPTSTR  lpDesktop;
```

```

    LPTSTR   lpTitle;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwXSize;
    DWORD    dwYSize;
    DWORD    dwXCountChars;
    DWORD    dwYCountChars;
    DWORD    dwFillAttribute;
    DWORD    dwFlags;
    WORD     wShowWindow;
    WORD     cbReserved2;
    LPBYTE   lpReserved2;
    HANDLE   hStdInput;
    HANDLE   hStdOutput;
    HANDLE   hStdError;

} STARTUPINFO, *LPSTARTUPINFO;

```

Well once again it's enough to initialize all of them with zero and cb to sizeof(cb).

#####From Win32.hlp#####

The PROCESS\_INFORMATION structure is filled in by the CreateProcess function with information about a newly created process and its primary thread.

```

typedef struct _PROCESS_INFORMATION { // pi
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;
    DWORD  dwThreadId;
} PROCESS_INFORMATION;

```

## Members

### hProcess

Returns a handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread

Returns a handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId

Returns a global process identifier that can be used to identify a process. The value is valid from the time the process is created until the time the process is terminated.

dwThreadId

Returns a global thread identifiers that can be used to identify a thread. The value is valid from the time the thread is created until the time the thread is terminated.

#####End Win32.hlp#####

This structure will be returned by CreateProcess and it includes the process handle (the most important for us). Now we got all parameters to create the process, but after it was created we have to wait until it was initialized. Therefore we must call WaitForInputIdle (hProcess, INFINITE) to make sure the whole virtual memory of the new process was committed, before we can patch it.

So when we call CreateProcess the target will be launched as a process and then it will start to unpack itself into memory. After the WaitForInputIdle will return, the target is beautifully unpacked just waiting for us to patch it. To patch it we need usually need WriteProcessMemory. But related to this api is it's ReadProcessMemory that might prove to be useful in future. Here I'll present both of them:

#####From Win32.hlp#####

The WriteProcessMemory function writes memory in a specified process. The entire area to be written to must be accessible, or the operation fails.

```
BOOL WriteProcessMemory(  
    HANDLE hProcess,    // handle to process whose memory is written to  
    LPVOID lpBaseAddress, // address to start writing to  
    LPVOID lpBuffer,    // pointer to buffer to write data to  
    DWORD nSize,        // number of bytes to write  
    LPDWORD lpNumberOfBytesWritten // actual number of bytes written  
);
```

## Parameters

### hProcess

Identifies an open handle to a process whose memory is to be written to. The handle must have `PROCESS_VM_WRITE` and `PROCESS_VM_OPERATION` access to the process.

### lpBaseAddress

Points to the base address in the specified process to be written to. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access. If this is the case, the function proceeds; otherwise, the function fails.

### lpBuffer

Points to the buffer that supplies data to be written into the address space of the specified process.

### nSize

Specifies the requested number of bytes to write into the specified process.

### lpNumberOfBytesWritten

Points to the actual number of bytes transferred into the specified process. This parameter is optional. If `lpNumberOfBytesWritten` is `NULL`, the parameter is ignored.

## Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call `GetLastError`. The function will fail if the requested write operation crosses into an area of the process that is inaccessible.

## Remarks

WriteProcessMemory copies the data from the specified buffer in the current process to the address range of the specified process. Any process that has a handle with PROCESS\_VM\_WRITE and PROCESS\_VM\_OPERATION access to the process to be written to can call the function. The process whose address space is being written to is typically, but not necessarily, being debugged. The entire area to be written to must be accessible. If it is not, the function fails as noted previously.

#####End#####

And it's brother (or sister, haven't checked): ReadProcessMemory

#####From Win32.hlp#####

The ReadProcessMemory function reads memory in a specified process. The entire area to be read must be accessible, or the operation fails.

```
BOOL ReadProcessMemory(  
    HANDLE hProcess,           // handle of the process whose memory is read  
    LPCVOID lpBaseAddress,     // address to start reading  
    LPVOID lpBuffer,          // address of buffer to place read data  
    DWORD nSize,               // number of bytes to read  
    LPDWORD lpNumberOfBytesRead // address of number of bytes read  
);
```

#### Parameters

hProcess

Identifies an open handle of a process whose memory is read. The handle must have PROCESS\_VM\_READ access to the process.

lpBaseAddress

Points to the base address in the specified process to be read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access. If this is the case, the function proceeds; otherwise, the function fails.

lpBuffer

Points to a buffer that receives the contents from the address space of the specified process.

nSize

Specifies the requested number of bytes to read from the specified process.

lpNumberOfBytesRead

Points to the actual number of bytes transferred into the specified buffer. If lpNumberOfBytesRead is NULL, the parameter is ignored.

#### Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

The function fails if the requested read operation crosses into an area of the process that is inaccessible.

#### Remarks

ReadProcessMemory copies the data in the specified address range from the address space of the specified process into the specified buffer of the current process. Any process that has a handle with PROCESS\_VM\_READ access can call the function. The process whose address space is read is typically, but not necessarily, being debugged.

The entire area to be read must be accessible. If it is not, the function fails as noted previously.

#####End#####

To call both functions we need :

HANDLE hProcess = ?	// handle of process
LPVOID lpBaseAddress, = your choice	// address to start reading/writing
LPVOID lpBuffer, = your choice	// address of buffer to read/write data
DWORD cbWrite, = your choice	// number of bytes to read/write
LPDWORD lpNumberOfBytesWritten = NULL	// actual number of bytes read/written

The only thing that we don't know (seems so) is the handle of the process we want to patch. If you have been paying attention you know that we get the handle of the process from the ProcessInformation structure, that gets filled after the process has been created (by CreateProcess)

To make things easier to grasp I have made a little program that shows a nice dialogbox that displays the current tickcount( milliseconds since Windows was started), but first we have the nag (that we must disable).To save space I will not give here the source of the target, but is included with this part of CIK.

To make things more interesting I've packed the example with Aspack and put it in the archive that came with this.

So start the target, press "The Time" button and you'll get the "Are you crazy?" message. From the theory explained earlier we will have to make a loader that will destroy the nag.

So let's follow the given steps:

1.Find the place we want to patch

Now the easiest way to patch is to NOP so will use this method. The call will be translated into 5 nops, coz

the call takes 5 bytes.To find where to patch just put in SoftIce a breakpoint on MessageBoxA and then press

"The Time".You'll be back in Sice at the call to the nag.Note the adress, that if i remeber is:0040100E. We know this:

-name of the target (duhh..)

-the address of the place we want to patch

-the buffer we'll need to patch (the nops)

Now we can get into coding.

Here is the source for the loader in win32asm(MASM v.7),coz i'm not really good C/C++

#####Start loader source#####

.386

.model flat,stdcall

option casemap:none

include \masm32\include\windows.inc

include \masm32\include\user32.inc

include \masm32\include\kernel32.inc

includelib \masm32\lib\user32.lib

includelib \masm32\lib\kernel32.lib

.data

target db "Example no.1.exe",0

writebuf qword 90909090h

er\_01 db "Couldn't create process..",0 ;if something goes wrong

er\_02 db "Couldn't write ...",0 ;same



```

.data?
hInstance dd ?
startinfo STARTUPINFO <?> ;the startupinfo structure
pi PROCESS_INFORMATION <?> ;the process_information structure
.code
start:
;-->get handle of our loader
invoke GetModuleHandle,NULL
mov hInstance,eax
;-->create the process
invoke CreateProcess,addr target,NULL,NULL,NULL,FALSE,NORMAL_PRIORITY_CLASS,NULL,NULL, addr startinfo,addr
pi
cmp eax,0 ;error??
jz eror_1
;-->wait for full initialization (hehe unpacking :))
invoke WaitForInputIdle,pi.hProcess,INFINITE
;-->finally write and exit
invoke WriteProcessMemory,pi.hProcess,0040100Eh,addr writebuf,5,NULL
cmp eax,0 ;error??
jz eror_2
fin:
invoke ExitProcess,NULL
eror_1:
invoke MessageBoxA,NULL,addr er_01,NULL,NULL
jmp fin
eror_2:
invoke MessageBoxA,NULL,addr er_02,NULL,NULL
jmp fin
end start

```

Compile it and see for yourself that it works, had any doubt in mind?? :>  
 Hope that everything was clear on cracking packed files.Let's go on.

## II. HOW TO PATCH A TARGET WITH A CHECKSUM CHECK

In this subchapter i'll give you the ground knowledge for cracking targets that have checksum checks with the memory patching aproach.

If you don't know by now, a checksum is a value that a program generates from a file, that could be an .exe or even .txt.I'm sure that you have heard about checksums in archiving utilities like RaR or Zip.You

have meet the situation when you download a program archived but when you try to open it, it says: Invalid Checksum(CRC32). This means that the file has been changed from the original version (from whom the checksum was made).

Now the general method of cracking this kind of applications is the same as for the packed ones. So you could memory patch a checksummed target using the previous method- the loader one. As i said, in order to have the rights to write in the address space of another process we could be the parent of the process (loader) or simply take over the all ready running process. Please keep in mind that both methods have their goods and bads.

This method can be described like this:

1. Launch the target as you would normally do
2. Launch an app made by you that will see if the target has been launched and if so take full control over the target.
3. After the taking over has been finished, do what you need to do and close the app

This is the way the app takes control over the target:

1. First find if the target is running -> FindWindow. This api will return us (if target has been found) a handle to the found window.
  2. Using the handle previously found will find the processid of the target -> GetWindowThreadProcessId
  3. Using the processid will get the handle of the process -> OpenProcess
- And thats it.

As you have seen we need some new api functions, that i'll give you here in order of the steps:

#### 1. FindWindow:

#####From Win32.hlp#####

The FindWindow function retrieves the handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows.

HWND FindWindow(

```
    LPCTSTR lpClassName,    // pointer to class name
    LPCTSTR lpWindowName    // pointer to window name
);
```

#### Parameters

lpClassName

Points to a null-terminated string that specifies the class name or is an atom that identifies the class-name string. If this parameter is an atom, it must be a global atom created by a previous call to the

GlobalAddAtom function. The atom, a 16-bit value, must be placed in the low-order word of lpClassName; the high-order word must be zero.

lpWindowName

Points to a null-terminated string that specifies the window name (the window's title). If this parameter is NULL, all window names match.

#### Return Values

If the function succeeds, the return value is the handle to the window that has the specified class name and window name.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

#####END Win32.hlp#####

#### 2. GetWindowThreadProcessId

#####From Win32.hlp#####

The GetWindowThreadProcessId function retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window. This function supersedes the GetWindowTask function.

DWORD GetWindowThreadProcessId(

    HWND hWnd,     // handle of window

    LPDWORD lpdwProcessId     // address of variable for process identifier

);

#### Parameters

hWnd

Identifies the window.

lpdwProcessId

Points to a 32-bit value that receives the process identifier. If this parameter is not NULL, GetWindowThreadProcessId copies the identifier of the process to the 32-bit value; otherwise, it does not.

## Return Values

The return value is the identifier of the thread that created the window.

#####END Win32.hlp#####

## 3.OpenProcess

#####From Win32.hlp#####

The OpenProcess function returns a handle of an existing process object.

```
HANDLE OpenProcess(  
    DWORD dwDesiredAccess,    // access flag  
    BOOL bInheritHandle,     // handle inheritance flag  
    DWORD dwProcessId        // process identifier  
);
```

## Parameters

### dwDesiredAccess

Specifies the access to the process object. For operating systems that support security checking, this access is checked against any security descriptor for the target process. Any combination of the following access flags can be specified in addition to the STANDARD\_RIGHTS\_REQUIRED access flags:

Access	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the CreateRemoteThread function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the DuplicateHandle function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the GetExitCodeProcess and GetPriorityClass functions to read information from the process object.
PROCESS_SET_INFORMATION	Enables using the process handle in the SetPriorityClass function to set the priority class of the process.
PROCESS_TERMINATE	Enables using the process handle in the TerminateProcess function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the VirtualProtectEx and WriteProcessMemory functions to modify the virtual memory of the process.
PROCESS_VM_READ	Enables using the process handle in the ReadProcessMemory function to read from the virtual memory of the process.

PROCESS\_VM\_WRITE Enables using the process handle in the WriteProcessMemory function to write to the virtual memory of the process.

SYNCHRONIZE Windows NT only: Enables using the process handle in any of the wait functions to wait for the process to terminate.

bInheritHandle

Specifies whether the returned handle can be inherited by a new process created by the current process. If TRUE, the handle is inheritable.

dwProcessId

Specifies the process identifier of the process to open.

#### Return Values

If the function succeeds, the return value is an open handle of the specified process.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

#### Remarks

The handle returned by the OpenProcess function can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

When you are finished with the handle, be sure to close it using the CloseHandle function.

#####END Win32.hlp#####

These are the parameters that we need to pass to the functions

-for FindWindow

LPCTSTR lpClassName = pointer to the name of the class

LPCTSTR lpWindowName = pointer to the name of the window

If we know one of them then the other is set to NULL. Usually more easy to find is the WindowName, because this is the name on the Titlebar(duh)

-for GetWindowThreadProcessId

HWND hWnd =the handle returned by the previous function

LPDWORD lpdwProcessId =pointer to a 32bit variable that receives the id

-for OpenProcess

DWORD dwDesiredAccess = PROCESS\_ALL\_ACCESS(ain't that beautifull?)

BOOL bInheritHandle =FALSE

DWORD dwProcessId                               =the id returned by the previous function

After getting the handle of the process you can do what you want-read or write to the addressing space of that process.

Now because the lack of time (and maybe sleep) i'll use the same target as in the packed target example, except that this time the target will not be packed. So i must ask you to imagine that you couldn't patch the target because of some checksum check somewhere in the target. So if the target is the same then the job is the same: eliminate the messagebox that appears when we want the time. Here is the source for the memory patcher:

```
#####Source of mem_patcher#####
.386
.model flat,stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
windowname db "Cracking Ideas for Kids",0
writebuf db 90h,90h,90h,90h,90h
er_01 db "Didn't find the target!.You sure you started it??",0 ;just in case
er_02 db "Couldn't write ...",0 ;same
er_03 db "Couldn't open process..",0 ;here too

.data?
hInstance dd ?
windowhandle dd ?
processid dd ?
hProcess dd ?

.code
start:
;--> get the handle of our module
```

```

    invoke GetModuleHandle, NULL
    mov hInstance, eax
;-->see if the target has been launched
    invoke FindWindow, NULL, addr windowname
    test al, al
    jz eror_1
    mov windowhandle, eax
;-->get the processid from the windowhandle
    invoke GetWindowThreadProcessId, windowhandle, addr processid
;--> finally open the process
    invoke OpenProcess, PROCESS_ALL_ACCESS, FALSE, processid
    test al, al
    mov hProcess, eax
    jz eror_3
;-->write and exit
    invoke WriteProcessMemory, hProcess, 00401046h, addr writebuf, 5, NULL
    cmp eax, 0
    jz eror_2
fin:
    invoke ExitProcess, NULL
eror_1:
    invoke MessageBoxA, NULL, addr er_01, NULL, NULL
    jmp fin
eror_2:
    invoke MessageBoxA, NULL, addr er_02, NULL, NULL
    jmp fin
eror_3:
    invoke MessageBoxA, NULL, addr er_03, NULL, NULL
    jmp fin
end start

```

;error??

If you have noticed that this app does his job even if the target is packed then you have been paying attention.

This method can be extended to very complex targets(can it??), but it's usually used for game cheats.

This method of cracking has the advantedge that the app doesn't have to be in the same directory as the target.

Final Words:

This being said this edition of CIK is finshed.

Once again i hope this tutorial is not writen in vain and you have learned something.

If you have some time please give me some feedback:

mycherynos@yahoo.com

Greetings:

-to all K23,CK,and DC members and especially to Acid\_Cool,Solata and Voodoo(happy bithday!!)  
yet to come:

PART IV:

The IAT HOOK APROACH or Hooking API calls via IAT

©bLaCk-eye 2002