In Memory Patching
Three approaches
(C) Stone / UCF & F4CG '98

After reading MadMax's essay on kernel patching I decided that perhaps it was time for an essay on "in memory patching". Contrary to the general +HCU philosophy my approach will be purely theoretical - the sourcecode I provide will serve as an example for you to build on. While this document might seem to be relatively technical and advanced it is unfortunately only an introductory text into the wonderful world of abusing windows :)

Is something preventing a patch? Is your target encrypted, packed, CRC'ed or you need the program to run sometimes with the patch applied sometimes without (A game-trainer for instance).Wouldn't you just love if you could patch the program in memory after it loaded, unpacked, did the CRC checks etc. ? You can. In the dos days we had TSR's to do this job. In the windows world it's a bit more difficult as the programming interface (Win32 API) is dynamic in contrast to dos's static interupt system. However new methods which in many ways are similar to TSR's are now avaible.

Kernel patching as MadMax pointed out is generally a bad idea. We need a more gentle approach. Which critereas would we like our solution to conform to? The critereas I'll use is:
1) The approach should perform ok in terms of compatability. That is work on both NT and 95 and hopefully on future versions as well.

2) The operating system should not suffer any long term effects of the crack. That is after termination of the target the OS should be left unchanged.

3) Only ring 3 meassures should be used. (Some of the API-functions I'll use from ring 3 will actually switch to ring 0, but atleast there will be no foreign code introduced at ring 0) For a discussion of this issue see various issues of HCU ML (Participants: Quine, RCG & myself)

Common ground
Our immediate problem is that in a preemptive operating system like Windows each process runs in it's own addressing space. Each time that the operating system switches to another process the virtual mapping is changed to fit that of the current process. The whole idea with memory patching is providing means of patching the target in it's addressing space at a certain time (after unpacking, CRC'ing or whatever is done). However since a criterea of the memory patch is that we can't patch the operating system nor the program itself we need to find a way of gaining access to the target addressing space from another processes.

The next problem we got is one of timing. Obviously the target needs to be patched after the CRC check has been performed or after it is unpacked in memory. And possibly it needs to be unpatched again to pass later checks. In other words we need a reliable trigger mechanisms. It is in this respect that the three methods I'll present here differ.

The loader approach
The critical assumption I'll make here is that the USER of the program can tell us how to time the patch thru another program. This basically means we assume that the user can:

1) Identify when patching is appropriate.
2) Switch to another program to activate.

About the first assumption it can be said - if it's a trainer this will never be a problem. Obviously the user will know when he want's to have infinate lives. Often a messagebox or some other visable sign shows itself when a patch is needed. E.g. A messagebox saying "Insert correct CD in drive and press OK" - It'd be easy to write a doc saying that when this occurs the dear user should press OK in another window first, and then in the target's obnoxious messagebox. However this is a serious shortcomming. Who said the program will actually let the user make a retry? Most 30-day trials tell the user the program has expired and the just exit or trial mode or whatever. Perhaps many different locations has to be patch at many different time making user-controlled patching a cumbersome solution.

On assumption 2 can it be said that many games doesn't like switching tasks and it's not likely that users will enjoy having to switch out of their game to get a new handful of bullets or whatever.

Let's get a bit more technical. Windows is so nice to provide us with an interface to write in other processes addressing space. The API needed is:
kernel32!WriteProcessMemory
If one take a closer look at this you'll find that what it actually does utillize Windows's int 2eh interface to switch to ring 0 meaning that it has ring 0 priveledges and thus is able to override page protection. However the interface has build in a security feature so you cannot override ring 0 data/code. (The int 2eh interface is for NT - I figure Windows 95 does something similar but I havn't checked it. Anyways the result is the same)

For WriteProcessMemory to work we need to identify by handle which process we want patched. IMHO the best to find such a handle is to create the target process yourself - that is do a good old fashioned EXEC from within your patch/trainer code. The API is
Kernel32!CreateProcessA
Ofcause there is different means of finding process handles.

To summerize a in-memory-patcher of this kind:
CreateProcessA (Target)
Wait for the user to say apply patch - e.g. a messagebox
WriteProcessMemory

Sourcecodes at:
http://www.one.se/~stone/general/nttrain.zip (or something)

The API-Hook/Debug Approach

Obviously the assumptions made for the Loader Approach can be too restrictive. For instance 30-day trials often exit prior to offering the user any obvious point of introducing a patch. So does dongles. Players might not like to switch task out of their beloved game to get another 10 bullets or whatever. What we really need is the target to trigger the patch and this section is a way of doing this.

The whole idea here is to hook an API-call, and make it perform to our desire. That can be return fake values under certain circumstances
it could be to patch the main program or it's dll's in memory. In short

what we wish to do is to let the api-call the program performs be surrounded by our code so that we can make it perform in every way we wish. Certain side benefits will come along as well. That is the code I present will show how it's possible to introduce breakpoints in an automated debugger which is indeed something very useful for the creation of for instance unpackers.

Again let's get down to it. A PE-file "imports" the functions it wishes to make use of. Because MS-developers decieded on a dynamic structure for API's it's obviously neasesary for each program to declare what functions it uses. This is done in a so called import table. Let's now take a deeper look into what takes place between the importtable in the PE-file and the execution of an API call by the target.

3 basic types of information is stored in the importtable. The first is DLL names, the second is function names and the third is a Thunk-RVA.
The information is stored in a structure that looks something like this:

```
      DLL1-Name
            Function1-from-dll1- name or ordinal
                  Thunk-RVA of Function 1 of DLL 1
            Function2-from-1dll-name or ordinal
                  Thunk-RVA of Function 2 of DLL 1
            ....

      DLL2-Name
            Function1-from-dll2- name or ordinal
                  Thunk-RVA of Function 1 of DLL 2
            Function2-from-1dl2-name or ordinal
                  Thunk-RVA of Function 2 of DLL 2
            ....
      ...
```

What Windows does while loading the PE-file is traverse thru this table following this "pseudo code":

```
  While more DLL's do
     { Load DLL into process addressing space
         While More Functions imported from current DLL do
            { Find address of Function and write this to the Thunk-VA for
              this Function
             }
      }
      END Load Imports
```

The function may be listed by name or something called ordinal. In every DLL each function that it exports for use by other programs is listed in an export directory (which is where Windows find the address of the imported function) in this list each DLL is assigned a number and usually a name too. The number is called ordinal. Importing can be done either by referencing this ordinal value or by using the name.

What the program then does when it's in need of the API-function it is this:
CALL Dword ptr [Thunk VA of needed function]
As an unimportant side note it can here be noticed that Borland and old Microsoft Compilers does this perfectly equivilent thing, calling a jmp dword ptr [Thunk VA] instruction.

Lets for a second imagine that we could stop execution of the target process right before it started and then inject our own code in to it's addressing space. Then we could simply replace the value at any Thunk-VA with a pointer to our own code and our code would be executed every time the program decieded to use this API. We could even save the old pointer and use this to chain the original intended API-code. Weeeeeee.. "Isn't this just great?" as Oprah Winfrey would say. "No, it is not", as I would reply.

We are left with a new problem. Or rather two. The first is stopping Execution of the target process before the program runs the first instruction so that we can be sure that our new pointers are in order. Second we're left the great problem of having code in the target's addressing space.

Solving a problem at the time we start by examining how we can stop our target process. Many people always state that Windows is overbloated and perhaps they are right - but in this case I'd say that it's damn convinient that MS-engeneers made a full-featured debug interface while designing API calls so that we could with the greatest of ease program a debugger without having to do the low-level work ourselves. Infact they made it so that not one line of ring 0 code has to be written to make an application debugger. "Isn't this just great?" as Oprah would phrase it? "Yes it is, maam" as I would reply. Because it get's even better. Windows engineers must've actually been thinking the day they made Windows. What good is a full-featured debug interface if the poor programmer has to make a PE-loader before he can even start debugging. Hey after all they already made a loader and they decieded to be helpful. CreateProcessA can open a process in Debug mode. This means that inside of most Windows's procedures hides status breakpoints that'll turn over the control to our debugger thru that interface. One of these status breakpoints triggers just before Windows is about to turn over control to the just loaded PE-file. Convinient!

Obviously if a process is in debug mode execution is suspended everytime a debug event occurs. A debug event is any non-handled exception. Pagefaults, breakpoints, division overflows, etc. And there is 6 different types of status breakpoints inside Windows that'll be triggering like Rambo in Iraq. So basically we need to send a message from our debugger process that it's ok to continiue every time we have encountered such an event. Ofcause if it's the event we've been looking for we need to do whatever it is we wish to do before giving the green light to run on. This is the reason behind the loop of
kernel32!WaitForDebugEvent
and
kernel32!ContiniueDebugEvent
in my code.

So now we know how to stop the program before it actually started. If you read the previous section you'll know how to exchange pointers. This leaves us with a grave problem. Injecting our code into the target's addressing space. Now this can be done in many ways indeed. We'll just be looking the one I chose.

What I'll try to obtain is making the program load a DLL for me. This ofcause isn't something the program is willing to do without force. Fortunately for the moment I'm President Clinton and the security counsil has agreed to bomb the target until it conforms to my ideas. The scene is set at the status breakpoint just before the target is about to start execution. It is fully

loaded and ready to go. However we're sitting comfortably with it suspended far far away in our own addressing space. The first thing we got to agree on is how it is we actually want's the target to do. Load OUR dll, find the process address of OUR function, replace the one found at the THunk-VA of the original. We now constuct code that will do just that in deltaoffset so that it can be inserted anywhere. Prior to actually running the program we found a page within the target that allowed execution. Most pages in the target allows execution but we just need one. We now read the page out the Process space of the target into our own and stores it safely. This is done thru another subfunction of INT 2eh which ofcause also overrides pageprotection etc. The API is:
kernel32!ReadProcessMemory
See Natzguls essay for a more thourough breakdown of this function. Now we write our own code that loads a DLL, finds the address of our function and replaces the Thunk-VA entry of the function with ours.

Now were ready to go? No. We're left with the problem that execution should be left otherwise unchanged so that we've written a page somewhere is bad news. So in addition to the code we appended we add an INT 3 which will when executed cause a debug event and once more suspend the target allowing us to restore the page. Unfortunately EIP of the target does not neassesarely point to our page, further we use all the registers and those needs to be restored too. So where do we turn? Windows internal knowledge. Upon creation that is prior to running any actual program code any one process has one and one thread only. Further Windows allows debuggers to fetch the Context of a thread. That is all relevant information about the threads current status. Such a context was originally intended for preemptive multitasking so that when ever the OS suspended execution of the thread to do another the context was saved, the address space swapped and another threads context was restored it's process's address space swapped in place and it was allowed to continiue. One should be aware that while a thread indeed has full context it's partly shared with that of the other threads in the process. E.g. the FPU is shared between threads in a process. Since we only got one thread in our process the terms of thread and process is incidental. I will not get dirty with the exact definitions of thread and process since a breakdown of operating system concepts is beyound the scope of this here text. Iceman (1998) has a brief description.
We ofcause now reads the context of our target's single thread, saves it then changes the EIP in it an resets it to point to our page of code in the target processspace. Ofcause our code will now execute till the int 3 we inserted is reached, then it's suspended and control is back with us. We now reset the context of the thread and restore the page we abused for our code. Then we simply let it run.

There is one last unfortunate thing about letting it run. If a process was created in Debug mode it stays in debug mode till it's terminated. That means that we need to stay in a loop of WaitForDebugEvent/ContiniueDebugEvent until that time where the process is actually terminated or the program will suspend itself and wait for our instructions. This wasn't too smart MS!

Practical notes on the debug approach
A last side note should be mentioned here. For reasons of alignment the Context structure should be on an address A so that (A and 3)=0. This is indeed not documented in MS's documentation of GetThreadContext. It is this that left me to believe that there was a bug in Windows NT in a previous edition of this text. Thanks to Quine for this correction. The reason that this alignment is required seems to be one of processing speed as x86'ers does not require this

type of alignment.

Further finding the ChunkVA of an imported function can easily be done by dumping the PE-file with Matt Pietreks PE-dump or similar. He gives the first chunk for each DLL, if your function isn't the first you add 4 bytes each time you need to move a line down to find our function.

The sourcecodes for this can be found at:
http://www.one.se/~stone/general/stnapih.arj

The MessageHook Approach

I'll skip relatively lightly over the in-depth technical issues of this method. It is simply far beyond this text to go into it. Maybe someday I'll write a book or something :) sorry..

The above method has it's advantages - and disadvantages. It's cumbersome. Indeed in many instances access to foreign addressing spaces can be gained easier. I will now examine one such method. The method was first described in MSJ 1994. I first noticed the potency of this method examing Grudge's crack for SubSpace. My sourcecodes and approach as such bares many resemblances with Grudge's initial work.

Most likely you've all encountered Windows Messageing system at one point or another. Breakpoints on "BMSG", HWND command in Winice is indeed breakpoints on messages and list possible recievers of messages. The whole idea behind messages goes back to the fact that we have a multitasking operating system. Several tasks needs to share equipment that can only be used by one at the time.
The obvious example is the mouse - the user will have only ONE mouse total, not one for each thread. Another is the keyboard, keyboard input is often ment for only one of the running threads. A total breakdown of the windows messageing and windowing system is far far beyound the scope of this small text. It'll suffice to say that any window made by any thread in any process is controlled thru messages from the Windows operating system.

Again we'll exploit that Windows is an overbloated operating system - or well as I would rather put it - a very potent equipped OS. The feature we'll be exploiting here is that of a Hook in the message system. For many reasons Microsoft decieded that even at ring 3 people should be able to intercept messages send to windows. Because they wanted this hook to be usuable for Computer Based Training they decieded that a hook would be no good if it did not have direct access to the addressing space of the process belonging to the thread it captured a message for. So they decieded that a MessageHookHandling procedure should be loaded into any process in which it captured a message. Further developers must've felt generous the day they designed this. They allowed a hook not just to one process - but to all!

Let's get a bit more technical on this. The API that installs the hook is:
User32!SetWindowsHookExA

The first problem about using this API to hook windows messages globally is the way it gain access to the address space of the thread which it intercepted a message to. To get real deep on this issue is again far beyound the scope of this text, but it has to do with how modules is mapped in pages thru out

the various memory contexts (Process addressing spaces).
The result is that you cannot have the hook within the EXE file that installs the hook - rather you need to have it in a DLL.

In other words we start out by loading the DLL in which we have our hook, then find the address of our hook procedure in it and feed this to SetWindowsHookExA.

The next problem we encounter is that of designing a messagehandling system. Since a LOT of messages is send out to windows system wide all the time it's important that we design our hook to be relatively fast or we'll be slowing the system. The easist way of doing this is only acting upon messages of a specific type. Choosing type depends on how you wish to time your patch. You can in this way time it to hit on keyboard activity in a window, mouse activity, windows being put to background and so on and so forth.

I've choosen the real simple hook type intended for ComputerBasedTraining. This hooks a large number of messages related to windows giving us plenty of things to act on without intercepting everything. Further in the hook procedure I test weather the message send was one that is supposed to create a window. This allows me to patch right after the "main" window of the program is created.
Without intercepting too much and even when intercepting I only run a few bytes of my code unless ofcause it happens that the program is creating a window. Obviously I might end up patching more than since this message type can be send many times. Other strategies could be hooking the keyboard, all messages etc. I very much believe that which type of hook and the exact design of your hook-handler is an issue that should be solved in relation to the specific problem you're dealing with. For instance in training hooking the keyboard and only acting on specific keys would be a good idea instead of acting on the window.

Mammon/HCU suggest a boolean variable to see if you already applied a patch. This can be a very good idea. Even better would be if you shut down your hook-installing process (program) and the hook along with it after you've patched. You should however be aware that this method has a caveat if your user deciedes to run multiple versions of the program you patched or if you're patching kernel32.dll in memory since this DLL might need to be patched more than once because an unpatched version might later on be mapped into your target's addressing space. Thanks to Mammon/HCU for a beneficial suggestion.

The next problem is that we need to hinder that we patch all processes. Again I abuse the concept of modules. By getting the current module filename and compareing it to the filename of the file we desire to patch I can identify weather this message was send to the target or to another program windows.

And the last problem is ofcause that we cannot keep messages from the target's windows and expect it to perform like it's supposed to. What we do is that we chain other possible hooks using the neatly provided API:
user32!CallNextHookEx

One final things is worth noting Exiting the process who "owns" the hook will destroy it - in other words we cannot shut down until we're sure we've patched the program and we cannot shut down if need multiple patches (as patching e.g. kernel32.dll (which is a bad idea anyway) would require. The good thing about it is that we can simply call ExitProcess when we're done and windows will take down the hook for us with no further adue... Pretty clever MS!

Since we in the hook-procedure have direct access to the addressing space of the target process we wouldn't need to use WriteProcessMemory, however it's a very good idea to do so. First and foremost WPM as described earlier overrides pageprotection. Second and also important - there is a difference in how pages in a process is handled between Windows 95 and Windows NT. If you patch a program's pages in Win NT (if it's not shared) it'll be copied and then the copy will be patched - thus the patch will not affect other processes utillizing the same module. In windows 95 this is not so. However WriteProcessMemory in Windows 95 has build in this mechanism ensuring that if you use WriteProcessMemory you'll not suffer differences between NT and 95. (This is not true if you patch above the 2g limit - which I btw cannot see why you'd do)

Here at the end I'll shortly describe the caveats of this method. It requires a window. Without a window in the target process you can't do didly with this method. However you can use this method to inject a DLL into the address space after the window has disappeared and then patch the IAT to make a API-hook of it like in the Debug-approach section.

Sourcecodes is avaible at:
http://www.one.se/~stone/general/stnmsgh.zip

Litterature

MadMax (1998) - Cracking using kernel32??, by MadMax Feb 1998.
      http://fravia.org

Natzgul (1998) - Unknown tittle, by Natzgul Jan 1998.
      http://fravia.org

Iceman (1998) - Tweaking with Windows 95 memory, by Iceman jan 1998
      http://fravia.org/iceman.htm

Pietrek, Matt - Windows 95 System Programming Secrets, IDG books 1995.

MSJ (1995) - Microsoft Systems Journal May 1995, Jeffrey Ritcher.

Various sourcecodes by Me :).. all can be found on my page
http://www.one.se/~stone

Thanks must go to:
Patriarch / PWA, friend, roomate and local expert.
Random / Xforce, God of the PE-format
Net Walker / Brazil
Quine / HCU
Acpizer / UCF, nah.. couldn't use what you send me.. but thnx anyways.
Grudge / CLS, it's been a pleassure abusing your work :)
Mammon / HCU
IceMan / HCU
United Cracking Force, my personal benefactor.
All of which I had many enlightning discussions with.

Also I'd like to thank

HalVar/HCU
WayneKerr/F4CG
LordByte/UCF
Madmax/HCU
G-Rom / Phrozen Crew

The Owl - Zen god of WinIce and a master of windows who have thought me a lot.
and many others
for their encouragements

email: stone@one.se
http://www.one.se/~stone
Stone / UCF & F4CG
2nd&mi!