



Writing Generic Patchers Using Dynamic Code Cave Injection[™]

potassium of ARTeam

Version 1.0 - 12th March 2006

IMPORTANT NOTE (READ THIS):	1
1. Abstract	2
2. Theory	2
3. Conclusions	5
4. Greetings	5

Keywords

Generic patcher, code caves, asm injection

IMPORTANT NOTE (READ THIS):

In contrast with other groups/individuals on the scene, neither the author of this document nor the organization named ARTeam distributes patches, cracked binaries or serial/activation codes. Besides contributing to the team with your own personal knowledge, this is the main criterion for remaining a member of ARTeam. Software developers; (yes, I'm quite sure you are reading these tutorials as well) please do not see this article as a threat to your organization or as a loss of income, but as a possibility for you to better yourselves and your products.

Yours truly, potassium / ARTeam 2006

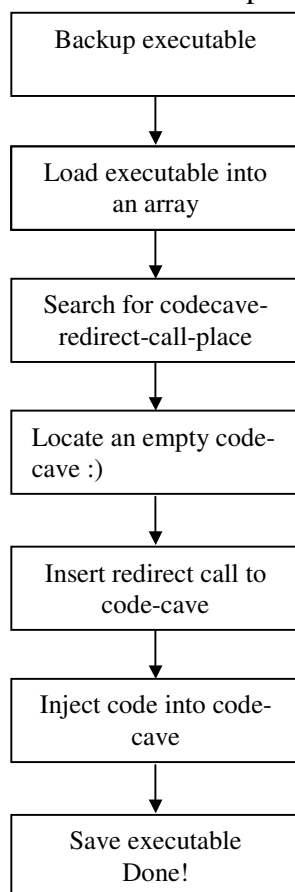


1. Abstract

Writing patchers is a very simple approach to compromise the security of an application. The goal of patching a target application is often to overcome a registration routine, time limitation or to activate disabled features. This is very doable in a non encrypted/packed application, but things gets more complicated when you want to patch a encrypted or packed application, since you have to figure out the encryption algorithm to be able to patch it. An alternative is of course to use a loader, but what if you don't know how to code a loader on your own? What do you do?! Answer: Continue to read this tutorial and you will find out :)

2. Theory

The starting point for writing a generic patcher is of course that you have gathered sufficient information about the application/protection in question so you know how to approach with applying the necessary changes to the executable. Similar to loaders, the single most important thing is finding that critical point in the execution of code where the encrypted memory section has just become decrypted and you apply your changes in real-time. But hey, a patcher can't do that, can it? No, it can't. But imagine that your patcher could determine where to inject assembler code that would search the correct place in memory to patch and do those necessary patches, assign proper values to some pointer of interest and then continue execution as normal. This however, CAN be done. Here is a simple schematic of how our patcher should operate:





Okay, this looks simple, right? Well, actually it's quite tricky to get this working on let's say, 100 games from publisher X who uses protection Y. You will have to find search patterns that will stay the same all the time, from application to application. So, to locate our redirect call to the code we need to know certain things about our target application. Lets assume, as an example, that the target in question uses WriteProcessMemory to write the newly decrypted data to memory. Then we might have something that looks like this:

```
00404D3D |. 50          PUSH EAX                      ; /pBytesWritten
00404D3E |. FFB5 5CEDFFFF PUSH DWORD PTR SS:[EBP-12A4]    ; |BytesToWrite
00404D44 |. FFB5 78EDFFFF PUSH DWORD PTR SS:[EBP-1288]    ; |Buffer
00404D4A |. 56          PUSH ESI                      ; |Address
00404D4B |. FFB5 68EDFFFF PUSH DWORD PTR SS:[EBP-1298]    ; |hProcess
00404D51 |. FF15 A8A04300 CALL DWORD PTR DS:[<&KERNEL32.WriteProce>; \WriteProcessMemory
00404D57 |> FFB5 78EDFFFF PUSH DWORD PTR SS:[EBP-1288]
00404D5D |. E8 FB7A0200 CALL xxxxxx.0042C85D
00404D62 |. 59          POP ECX
00404D63 |> FFB5 68EDFFFF PUSH DWORD PTR SS:[EBP-1298]    ; /hObject
00404D69 |. FF15 20A24300 CALL DWORD PTR DS:[<&KERNEL32.CloseHandl>; \CloseHandle
00404D6F |. 834D FC FF   OR DWORD PTR SS:[EBP-4],FFFFFFF
00404D73 |. 8D8D 84FEFFFF LEA ECX,DWORD PTR SS:[EBP-17C]
00404D79 |. E8 9CC40100 CALL xxxxxx.0042121A
00404D7E |> B0 01      MOV AL,1
00404D80 |> 8B4D F4     MOV ECX,DWORD PTR SS:[EBP-C]
00404D83 |. 5F          POP EDI
00404D84 |. 5E          POP ESI
00404D85 |. 64:890D 000000>MOV DWORD PTR FS:[0],ECX
00404D8C |. 8B4D F0     MOV ECX,DWORD PTR SS:[EBP-10]
00404D8F |. 5B          POP EBX
00404D90 |. E8 B76B0200 CALL xxxxxx.0042B94C
00404D95 |. C9          LEAVE
00404D96 | \ C3        RETN
```

Hmm.. Those MOV AL,1 and so on, at 00404D7E looks nice. Suppose we have studied 10 targets and they all look the same it probably likely that the others look the same as well. So, B0 01 8B 4D F4 goes into our search string for the placement of the redirect to code-cave call. If these bytes are found and you replace these bytes with your redirect call, you must also move and execute these operations in your code-cave as well, or else... Crash :(

Locating a code-cave is easy. Just search the executable for lots of zeros, preferable as many zeros you need for your assembler injection. Injecting the assembler code is also rather easy, probably the best approach, at least the one that I've come up with, is to code inline in OllyDbg and then simply do a binary copy, paste the bytes into a string. Then simply convert this string into decimal values and let the patcher write them to the array. Something like this, suppose you want to inject this section (this is junk code and nothing of interest) of code:

```
00404D97 $ 55          PUSH EBP
00404D98 . 8BEC        MOV EBP,ESP
00404D9A . 83EC 10     SUB ESP,10
00404D9D . 53          PUSH EBX
00404D9E . 56          PUSH ESI
00404D9F . EB 09      JMP SHORT xxxxxxxx.00404DAA
00404DA1 . EB 44      JMP SHORT xxxxxxxx.00404DE7
```

This code when copied as binary would look like this: 55 8B EC 83 EC 10 53 56 EB 09 EB 44
OllyDbg nicely places a space between every byte, so we can let our patcher simply cut every three chars and convert to decimal (or hex) and store into the array.



Okay, now we are ready to discuss the assembler code that we are going to inject. Lets say the you have a pointer that holds the registration status (eg. 1 is registered) but that pointer is located at a very obscure part of memory that is allocated at run-time. If that is the case (well it probably is otherwise you wouldn't need a code cave to do it for you, right) your injected assembler code must find it and assign the right value to it for you during run-time. Again, the strength of you search pattern is crucial to succeed. As an example here I'll use the pattern: 80 7E 04 00 75. In fact, you often also find a pattern to locate the 'registered' pointer as well, if it varies from application to application.

```
00439EA4  /$ B0 01          MOV AL,01
00439EA6  |. 8B4D F4         MOV ECX,DWORD PTR SS:[EBP-C]
00439EA9  |. BA 00004100      MOV EDX,xxxxxxx.00410000
00439EAE  |> 803A 80          /CMP BYTE PTR DS:[EDX],80
00439EB1  |. 75 2C            |JNZ SHORT xxxxxxxx.00439EDF
00439EB3  |. 807A 01 7E       |CMP BYTE PTR DS:[EDX+1],7E
00439EB7  |. 75 26            |JNZ SHORT xxxxxxxx.00439EDF
00439EB9  |. 807A 02 04       |CMP BYTE PTR DS:[EDX+2],4
00439EBD  |. 75 20            |JNZ SHORT xxxxxxxx.00439EDF
00439EBF  |. 807A 03 00       |CMP BYTE PTR DS:[EDX+3],0
00439EC3  |. 75 1A            |JNZ SHORT xxxxxxxx.00439EDF
00439EC5  |. 807A 04 75       |CMP BYTE PTR DS:[EDX+4],75
00439EC9  |. 75 14            |JNZ SHORT xxxxxxxx.00439EDF
00439ECB  |. C602 C6          |MOV BYTE PTR DS:[EDX],0C6
00439ECE  |. C642 01 46       |MOV BYTE PTR DS:[EDX+1],46
00439ED2  |. C642 03 01       |MOV BYTE PTR DS:[EDX+3],1
00439ED6  |. C642 04 EB       |MOV BYTE PTR DS:[EDX+4],0EB
00439EDA  |. BA FFFF4200      |MOV EDX,xxxxxxx.0042FFFF
00439EDF  |> 42              |INC EDX
00439EE0  |. 81FA 00004300    |CMP EDX,xxxxxxx.00430000
00439EE6  |.^75 C6           \JNZ SHORT xxxxxxxx.00439EAE
00439EE8  |. BA 00104000      |MOV EDX,xxxxxxx.00401000
00439EED  |> 803A 83          /CMP BYTE PTR DS:[EDX],83
00439EF0  |. 75 21            |JNZ SHORT xxxxxxxx.00439F13
00439EF2  |. 807A 01 C4       |CMP BYTE PTR DS:[EDX+1],0C4
00439EF6  |. 75 1B            |JNZ SHORT xxxxxxxx.00439F13
00439EF8  |. 807A 02 0C       |CMP BYTE PTR DS:[EDX+2],0C
00439EFC  |. 75 15            |JNZ SHORT xxxxxxxx.00439F13
00439EFE  |. 807A 03 A2       |CMP BYTE PTR DS:[EDX+3],0A2
00439F02  |. 75 0F            |JNZ SHORT xxxxxxxx.00439F13
00439F04  |. 8B5A 04          |MOV EBX,DWORD PTR DS:[EDX+4]
00439F07  |. C603 01          |MOV BYTE PTR DS:[EBX],1
00439F0A  |. C643 01 01       |MOV BYTE PTR DS:[EBX+1],1
00439F0E  |. BA FFFF4200      |MOV EDX,xxxxxxx.0042FFFF
00439F13  |> 42              |INC EDX
00439F14  |. 81FA 00004300    |CMP EDX,xxxxxxx.00430000
00439F1A  |.^75 D1           \JNZ SHORT xxxxxxxx.00439EED
00439F1C  |. 33DB            |XOR EBX,EBX
00439F1E  |. 33D2            |XOR EDX,EDX
00439F20  \. C3             |RETN

Stolen bytes :)
Stolen bytes :)
Start of search
is first search byte = 80 ?
if not.. jump sucker
is next search byte = 7e ?
if not ... and so on.

mov byte [esp+04],1

and jmp instead of jnz
to end the search

if search not complete ^^^
look for combination:
83 C4 0C A2

@EDX+4 is our registered pointer
Put 1 in it
And in this one too
end search.

reset to zero
reset to zero
continue execution
```

That's it! We're ready to go.. Save the changes to disc and test the application.



3. Conclusions

This is patching made a little more advanced and if your search patterns are strong, that is they are very generic commands and not depending too much on specific registers and pointers, you could simply make ONE patcher for like everything from 1-X products in one sweep instead of X unique patchers for each application. ;) Of course to write something like this requires a lot of knowledge of the target application, the language you write the patcher in and some knowledge of assembler to do the real-time patching. It is also fully possible to add flexibility to the search criterions like; instead of searching for B0 01 74 F4 maybe it's B1 01 74 F4 the next time, and then simply compensate for that as well. This is patching with dynamic code cave injection in a nutshell. :)

The possibilities are endless! Till we meet again ..

Best regards from potassium, citizen of central code cave city.

4. Greetings

To the whole ARTeam, friend-teams out there and of course you who are reading this.