Keygen Injecton

Kwazy Webbit

Abstract

This essay will teach you a technique I've first developed while writing my essay on cracking PaneKiller. A lot of people have come up to me recently, and told me they really enjoyed it. This technique is what I call "keygen injection". (I called it "Kwazy Webbit style" in the other essay, but that seems arrogant and immature afterwards. Hence, I changed the name.) Basically, what it does is the following: If the serial is correct the program still registers like normal. If the serial is incorrect, control is transferred to our code placed INSIDE the program essentially turning the program into its own keygen. Sounds pretty leet, huh? Well.. Read on is all I can say.

Keywords: *Keygen Injection; Adding Functionality; Notepad*

I. Practical uses

As a cracker, I'm sure you already know what a keygen is. It is used for name/serial protections. It is a program that lets you enter your name in an editbox, and when you press the button the program starts calculating. It calculates the registration code that matches the entered name. (Either by brute-force or by a reversed hashing algorithm). It then enters the code in another editbox. The general idea is, as you see, fairly simple. But the calculations that go on behind the scenes differ with (almost) every target program. Some protections take quite some time and effort to completely reverse. Just think of the endless deadlistings we've all seen from time to time. It is not uncommon for a program to endlessly hash the username into a code, which is then compared with the entered registration code.

To get a valid serial for these protections is (needless to say) piss-easy. All you have to do, is find out where it compares the two codes and use a debugger to 'fish' the serial. (If you do not understand that, I suggest you read my essay 'Cracking Like Kwazy' or one of the other million available cracking essays on the subject)

To create a keygen for this type of protection however, can take a lot of effort. You have to copy the entire username-hashing routine into your keygen, and then only the trivial task of displaying the matching reg code remains.

With the technique I'm about to show you, you will be able to skip the first part. (Yup, the username-hashing..) This will save you a lot of time and effort, simply by understanding how programs work.

II. Getting started

Alright, sounds nice. How do I do it?

Well, we start with serial-fishing. I'm assuming we all know how that is done. Now you've already found the memory location where it compares the serials. After that, there is usually a conditional jump. One way leads to the "Wrong Code" messagebox, and the other to the "Thank you" messagebox. You all know the routine. Now what if we changed the location that jump points to? What if we made it point to our own code at a certain location? We could run our own keygen-code, instead of displaying that "Wrong Code" messagebox, and after that just do whatever the program would have normally done.

This is all fairly abstract, maybe an example will clear it up :

Original code:

```
CMP EAX, EBX            ; EAX=our serial  |  EBX=correct serial
JNZ WrongCode           ; Not the same? Then display "Wrong Code"
..                      ; This is where the "Thank You" stuff is.
```

Code after changes:

```
CMP EAX, EBX            ; EAX=our serial  |  EBX=correct serial
JNZ KeyGen              ; Not the same? Then run the built-in keygen.
..                      ; This is where the "Thank You" stuff is.
..
..                      ; All the other code of the program.
..
Keygen:                 ; We place our keygen at a convenient location.
..                      ; Display the RIGHT serial.
JMP WrongCode           ; Continue the program as normal.
                        ; (We wouldn't want the program to find out
                        ; we were messing with it, now would we? =)
```

Of course we haven't made that built-in keygen yet, we'll get to that later. What is important now, is that you understand how you can easily intercept the jump to the "Wrong Code" box, and use it to your own advantage. There is a difficulty in doing this. For example, you have to make sure you dont run into any problems with stack and you have to preserve certain registers. Don't worry, you won't have to simply think up what will have to be done. You can simply copy what the program would have done if it were to get a wrong serial. I'll give you another (more concrete) example to illustrate :

```
..
:00401269 39058B104000   cmp dword ptr[0040108B],eax     ;our code == correct code?
:0040126F 7516           jne 00401287                    ;jump if not equal
:00401271 6A00           push 00000000                   ;push MB_OK
:00401273 6866104000     push 00401066                   ;push lpGoodWork
:00401278 685A104000     push 0040105A                   ;push lpYouDidIt
:0040127D FF7508         push [ebp+08]                   ;push hWin
:00401280 E869020000     call 004014EE                   ;call MessageBoxA
:00401285 EB14           jmp 0040129B           ;jump past the "bad code" messagebox
:00401287 6A00           push 00000000                   ;push MB_OK
:00401289 687D104000     push 0040107D                   ;push lpBadCracker
:0040128E 6871104000     push 00401071                   ;push lpYouSuck
:00401293 FF7508         push [ebp+08]                   ;push hWin
:00401296 E853020000     call 004014EE                   ;call MessageBoxA
:0040129B E97D010000     jmp 0040141D
..
:0040141D FF7514         push [ebp+14]                   ;push lParam
:00401420 FF7510         push [ebp+10]                   ;push wParam
:00401423 FF750C         push [ebp+0C]                   ;push uMsg
:00401426 FF7508         push [ebp+08]                   ;push hWin
:00401429 E89C000000     call 004014CA                   ;call DefWindowProcA
:0040142E C9             leave                           ;
:0040142F C21000         ret 0010                        ;return from the WndProc
..
```

This (very simplistic) code works as follows:

It checks if the code we entered is correct. If so, It displays a MessageBox with a text "registered" or something to that effect.

If not, it displays a similar MessageBox, with a "bad code" like text. That it displays a messagebox is not interesting to us at this point. What IS interesting however is the fact that AFTER the "bad code"-MessageBox, we see:

```
:0040129B E97D010000              jmp 0040141D
```

As we can see, it jumps to the end of the WndProc and calls the DefWindowProcA function, while keeping ESP just the way it is (no POPs, PUSHes or other funny business)

The hell you say! It pushes 4 things onto the stack!

Well, DefWindowProcA is defined as being

```
LRESULT DefWindowProc(
    HWND  hWnd,             // handle of window
    UINT  Msg,              // message identifier
    WPARAM  wParam,         // first message parameter
    LPARAM  lParam          // second message parameter
);
```

So, they're just parameters to the DefWindowProcA function. Also, notice that none of the parameters are dependant on registers (no PUSH EAX or anything like that). So we can just redirect the

```
:0040129B E97D010000              jmp 0040141D
```

to point to our keygen routine. And, as long as we preserve ESP and jump back to 0040141D afterwards, We can do whatever we want.

YEEEEEEEHAAAWWWW!!! I'M THE KING OF THE WOOORLD!!!

Well.. within limits.

sigh*

III. Hands-on experience

I assume you understand the basic principle now. The next step is seeing this technique in action. Get the example target first. The zip file contains these files:

- The original exe, with 3 different protections, called InjectMe.exe
- The exe with a built-in keygen for type 1 protection, called InjectMe1.exe
- The exe with a built-in keygen for type 2 protection, called InjectMe2.exe
- The exe with a built-in keygen for type 3 protection, called InjectMe3.exe
- A file called InjectMe3.exe.txt used for type 3 protection

This program was made to be without (m)any complications.. You will not find many targets in 'the real world' this easy, but I found it would be the best way to illustrate this technique. Best to learn it clean, and then start dealing with complications.

As said, I made a program with 3 different protections..

They range from perfect for our needs (type 1), to not well suited for it (type 3). I'll show how to use this technique on all the protections.. Starting with the easiest one, going up to number 3.

A. Protection 1 - Plain and simple

Lets take a look at the first protection (open it up in wdasm or run through a debugger like Softice). Its code starts at 401069h. (I'm sorry, but this section will have a lot of deadlistings. I normally dislike essays that lean on them too much, but I saw no proper alternative in this case)

As you (hopefully) can see, it first calls GetDlgItemTextA to get the Name you entered out of the editbox.

```
:00401069 6A14      push 00000014                   ;push 20 (14h)
:0040106B 8D45EC    lea eax, dword ptr [ebp-14]     ;load address of buffer into eax
:0040106E 50        push eax                        ;push the address of the buffer
:0040106F 6A64      push 00000064                   ;push 100 (64h)
:00401071 FF7508    push [ebp+08]                   ;push dword at[ebp+08]
                                                    ;(in the wndproc, thats where
                                                     the hWnd is stored)
* Reference To: USER32.GetDlgItemTextA, Ord:0102h   ;Now that the parameters are
                                                     all pushed
                                                    ;neatly onto the stack,
                                                     the program can call
:00401074 E85B020000   Call 004012D4                ;the GetDlgItemTextA function
                                                     to make it all happen.
```

Use win32.hlp for this kind of stuff, if you don't already know them from memory. Here's the definition of GetDlgItemTextA:

```
UINT GetDlgItemText(
    HWND hDlg,              // handle of dialog box
    int nIDDlgItem,         // identifier of control
    LPTSTR lpString,        // address of buffer for text
    int nMaxCount           // maximum size of string
);
```

After that code, whatever was typed into the editbox, is now in memory at location [ebp-14]. Run the program through a debugger and check this yourself. GetDlgItemTextA returns the amount of characters it has gotten in eax. The program uses this for a quick check, and fills in the name 'lamer' if nothing was entered:

```
:00401079 85C0    test eax, eax       ;no characters gotten?
:0040107B 7473    je 004010F0         ;go and fill in 'lamer' into the editbox
```

The program now completed its first step in the registration routine. Getting the user's name. What's next? That varies from program to program.. They all basically do the exact same stuff.. But the way they do it, and the order is different occasionally.

In this case, the next part is:

```
:0040107D 8D55EC      lea edx, dword ptr [ebp-14]  ;load address where
                                                    name is stored into eax
:00401080 52          push edx                     ;push that address
:00401081 E8FD010000  call 00401283                ;call an internal function
```

Basically, this is a function that has as its parameter our name.

This could be almost anything.. It could be a routine that converts the name to uppercase, it might be a hashing algorithm, it could even be a simple function to return the length of the name. There's no way to know without looking in the function, and studying its code in a deadlisting...

Urgh, do we really NEED to know what it does?

However...

Yes?

For this technique I'm about to show you, you won't actually need to know what the program is doing up to where it compares the two codes.

Wow, I actually didn't expect that.

So let's just step over it in a debugger, and peek at what it returns just for the sake of being able to make an educated guess. :) It seems to leave the string unchanged. (to which it had access too, dont forget the ADDRESS of it was pushed, so it could have modified the contents of that as well) It returns (in EAX of course) a mysterious number... Could perhaps be the hashed username. Again, this is pure speculation. It might just as well some random number that is never used again.

Lets just keep going. Next we see this:

```
:00401086 50     push eax   ;The mysterious number gets pushed to the stack

* Possible StringData Ref from Data Obj ->"%lu"
                                  |
:00401087 6809304000  push 00403009   ;a pointer to the string "%lu" is pushed
:0040108C 8D45E3      lea eax, dword ptr [ebp-20]
                      ;load the address of ebp-20 into eax
:0040108F 50          push eax                    ;push that address

* Reference To: USER32.wsprintfA, Ord:02A5h
                                  |
:00401090 E827020000  Call 004012BC               ;call wsprintf
```

To understand what this does, you will need to understand the wsprintf function. Wsprintf is a function used for making it very easy to print out numbers, characters and strings, all in one text. To do this, it uses something known as a format-control string. In short, it is a string that has special notations for letting you print out numbers as well (converting them to ascii automatically for you). For example, "%i" means the first parameter should be assumed to be a signed integer, and printed out in decimal. Here's the official definition of the function:

```
int wsprintf(
    LPTSTR lpOut,           // pointer to buffer for output
    LPCTSTR lpFmt,          // pointer to format-control string
    ...                     // optional arguments
);
```

Say for example, I want to print out a value called num as a decimal and a hexadecimal number. I would use a call similar to this:

```
wsprintf(lpBuffer,"decimal: %i , hexadecimal: %x",num,num);
```

You get the idea right? If not, you should read up on some win32 coding, there are plenty of resources available. Whats important here, is that this function call is basically:

```
wsprintf(lpBuffer,"%lu",value);
```

With the buffer located at [EBP-20]. In short, after this call the 32bit integer will be printed out as text at [EBP-20]. I guess its gonna do something with that return value after all.. :)

Back to the deadlisting:

```
:00401095 6A14       push 00000014                  ;nMaxCount
:00401097 8D45EC     lea eax, dword ptr [ebp-14]
:0040109A 50         push eax                       ;lpString
:0040109B 6A65       push 00000065                  ;nIDDlgItem
:0040109D FF7508     push [ebp+08]                  ;hDlg

* Reference To: USER32.GetDlgItemTextA, Ord:0102h
                                   |
:004010A0 E82F020000              Call 004012D4
```

I guess I've already explained this function, so I'll just quickly translate to C source to clarify:

```
GetDlgItemTextA(hWnd,101,lpBuffer,20);
```

So, the text that was in the editbox who's ID was 101 is now stored at [EBP-14] (with a maximum size of 20 characters). A quick check using:

```
:004010A5 85C0       test eax, eax
:004010A7 745B       je 00401104
```

verifies that some text was actually gotten from editbox 101, and if there is the program can continue at 4010A9h (the next instruction). The program now has our name. It also has the serial number we entered. It now has all the elements it needs to check if our serial name is correct. Lets see where it goes from here..

```
:004010A9 8D45E3       lea eax, dword ptr [ebp-20]
:004010AC 50           push eax
:004010AD 8D45EC       lea eax, dword ptr [ebp-14]
:004010B0 50           push eax

* Reference To: KERNEL32.lstrcmpA, Ord:02D6h
                                   |
:004010B1 E83C020000   Call 004012F2
```

Looking up what this function does in win32.hlp gives us the following definition:

The lstrcmp function compares two character strings. The comparison is case sensitive. If the strings are equal, the return value is zero.

Lets look at the parameters it gets first. If you'll remember, EBP-14 holds our entered registration number, and EBP-20 is where the username hash was stored. It just compares if the two are equal.

Oh Come on. No real protection is that simple!

The same cracking method applies to larger, more chaotic protections.. This is a simple example to show how the technique itself works. After the lstrcmp is 'ye olde eax test' (You've probably seen it a million times):

```
:004010B6 85C0            test eax, eax         ;eax=0? (are the strings equal?)
:004010B8 0F8519000000    jne 004010D7          ;if not, show BAD CRACKER messagebox
```

Finally, we've reached the point where we finished analysis of this protection.. Now we can start the actual cracking of the target. After this, we can use the regular methods again..

- Patching: NOP the JNE so it will never jump.
- Fishing: Look at the lstrcmp to see what the serial for our name was, and use that.

Usually my favorite, as it takes least work and you still registered the program the right way.. By getting a valid serial.

- Keygenning: Analyse the call at:
  `:00401081 E8FD010000     call 00401283   ;call an internal function`

to make a key generator (most good release groups would do this) Now, lets combine all three at once... Patching the .exe so we can have both the ease of serialfishing and the flexibility of a keygenerator. Lets start this example from the serial-fishing point of view, as it is most similar. To get a valid serial for our name, we would use a debugger and go all the way to the lstrcmp, and simply look at the text at EBP-20 to see what our serial should have been. Now lets use the programs structure against itself. The JNE is a definite weakness for example:

```
:004010B6 85C0           test eax, eax  ;if eax is not 0, you entered the wrong code
:004010B8 0F8519000000   jne 004010D7   ;<- an easy to manipulate jump
```

Good, lets look at where some free space is inside the .exe file.

> *NOTE:*
> *I wont overwrite the 'bad cracker' messagebox this time, like i did in my previous essay. I've realised there isnt always enough space, which generally leads to bad cracking habits. You well end up messing around too much inside the .exe possibly damaging it in the process. I've decided to explain it properly this time, so you won't run into trouble.*

Time to open up the file in HIEW for a look in the PE header (F8). Pressing (F6) brings up the object table with its properties. You can use any program you want to to view the PE header.. I just normally use HIEW. Regardless what you use, you will find the object table to be as follows:

```
# Name VirtSize  RVA PhysSize Offset Flag
1 .text 000002F8 00001000 00000400 00000400 60000020
2 .rdata 000001A6 00002000 00000200 00000800 40000040
3 .data 0000005B 00003000 00000200 00000A00 C0000040
4 .rsrc 00000230 00004000 00000400 00000C00 C0000040
```

A PE-file (Portable Executable) consists of several objects. Each has its own location, size and flags. One is for the data (and has read/write flags), the other for the code (with the executable flag).. Etc. This shouldnt be a PE document, if you need info on the PE file format, there are lots of essays released already. PE.pdf is in the RE section of my site, in case you need to read up on some things. The '.text' object is where the rest of the code is too, so its a nice place to put our keygenerator.. You can put the keygenerator into any object you want to, dont get me wrong. However since we have to pick one anyway, this one already has the executable flag that is needed to execute the code.. We might as well use it. IF it has enough room that is.

Lets check it out:

```
# Name VirtSize  RVA PhysSize Offset Flag
1 .text 000002F8 00001000 00000400 00000400 60000020
```

It has a VirtSize of 2F8h , which is the size of the actual code.. And it has a PhysSize of 400h, which is the code filled up to the nearest multiple of 200h (or the file alignment for the nitpickers ;) That gives us a total of 400h-2F8h=108h bytes of unused padding inside the '.text' object. (264 bytes in decimal) You gotta love the PE format for leaving us reversers with all that space. Let's also have a quick look at where we can begin to edit.. The Offset (the start of the object in file) is 400h.. The original code takes up 2F8h bytes, we can start at offset 400h+2F8h=6F8h in file. Lets round that off to a nice (and easy to remember) 700h. Thats the location where our keygenerator will be placed. We also need one more thing, we need to know the Virtual Address of that location. Luckily, its all perfectly easy to find in the PE header. The Image Base (the address at which the program itself is loaded into memory) of the program is 400000h. The RVA (Virtual Address Relative to the Image Base) of the '.text' object is 1000h. Another simple calculation brings us to conclude that the '.text' object will be at 400000h+1000h=401000h in memory. The keygen will thus be located at 401000h+300h=401300h (we rounded the 2F8h up to 300h remember?) We now have everything we need:

```
File Offset Space Virtual Address
700h 100h bytes 401300h
```

Let's figure out what we want to put inside the program.

```
:004010B6 85C0           test eax, eax ;if eax is not 0,
                                        you entered the wrong code
:004010B8 0F8519000000   jne 004010D7  ;<- an easy to manipulate jump
```

This is where we left off. You may also recall that the serial we need is stored at EBP-20 as a string. (If you dont believe this go to this location in softice and type 'd ebp-20' to be sure what is stored there ;) What remains is making the code that will put the string at ebp-20 into the registration code editbox. If you've done some win32 coding, you will probably already know several ways to do this.. I'll show you a simple one. If we look at the imported functions in this program, we can see the function SetDlgItemTextA. Looks interesting, right? It should, the definition is:

```
BOOL SetDlgItemText(
    HWND hDlg,              // handle of dialog box
    int nIDDlgItem,         // identifier of control
    LPCTSTR lpString        // text to set
);
```

Great, buddy.. and where am I supposed to find all these parameters?!

To find the rest of the parameters, let's look at the call to GetDlgItemTextA to get our registration number. We need to do the opposite of it now, which of course, takes mostly the same parameters..

```
:00401095 6A14        push 00000014                 ;nMaxCount
:00401097 8D45EC      lea eax, dword ptr [ebp-14]
:0040109A 50          push eax                      ;lpString
:0040109B 6A65        push 00000065                 ;nIDDlgItem
:0040109D FF7508      push [ebp+08]                 ;hDlg

* Reference To: USER32.GetDlgItemTextA, Ord:0102h
                                  |
:004010A0 E82F020000              Call 004012D4
```

There, you see? The hDlg and the nIDDlgItem are already stored there.. Life is sweet. Lets put some temporary code down with the things we know:

```
lea eax, dword ptr [ebp-20] ;load address of the string into eax
push eax                    ;push lpString   (points to the right serial)
push 65                     ;push nIDDlgItem (the ID of the
                             registration code editbox)
push [ebp+8]                ;push hDlg       (the handle of the dialog)
call SetDlgItemTextA        ;<-- We dont know where this is yet...
```

Let's quickly look up where SetDlgItemTextA is, so we can call it and move on to the patching. Open up the program in W32Dasm, and look at the imported functions. Locate USER32.SetDlgItemTextA and double-click on it. It will take you to a call to that function.. The program already knows how to call it, lets just follow its example. One of the calls to SetDlgItemTextA is:

```
* Reference To: USER32.SetDlgItemTextA, Ord:0228h
                                       |
:004010FA E8E7010000                   Call 004012E6
```

Ok, the function is located at Virtual Address 4012E6

Neat. Still no good to us in a hexeditor though, for that we need to find out its location in the .exe file.. Go to 4012E6 in W32Dasm. There are several ways to do this, one is by standing on the call and pressing the 'call' button. The we should see this:

```
* Reference To: USER32.SetDlgItemTextA, Ord:0228h
                                       |
:004012E6 FF2524204000                 Jmp dword ptr [00402024]
```

You may be wondering why this is a jump, and why you are not in the SetDlgItemTextA function. Well, this is known as a jump table. The address of the REAL function is stored by windows at [402024], which is where this jumps to. Why this method is used, is not relevant now, this essay is big enough as it is. What IS relevant however, is that we want the file offset of this place, instead of the virtual address 4012E6. If it isnt selected already, select the line by doubleclicking on it. You will now see on the bottom of your screen, in the W32Dasm status bar:

```
Line:580 Pg 12 and 13 of 15 Code Data @:004012E6
@Offset 000006E6 in File:InjectMe.exe
```

That means the Virtual Address is 4012E6 and the File Offset is 6E6 Now we have all we need.. Lets take one last look at the total keygenerator we're going to inject:

```
lea eax, dword ptr [ebp-20]  ;load address of the string into eax
push eax                     ;push lpString   (points to the right serial)
push 65                      ;push nIDDlgItem (the ID of the registration
                              code editbox)
push [ebp+8]                 ;push hDlg       (the handle of the dialog)
call 6E6                     ;Call SetDlgItemTextA
```

Open up InjectMe.exe in HIEW, go to ASM mode at location .401300 or 700 in the file. (depending on whether you're viewing the locations as virtual addresses or file offsets, respectively. You can toggle between the two modes with Alt+F1). Once there, press F3 followed by F2 and enter the code.

Keep in mind, the syntax of HIEW is slightly different than that of W32Dasm, so the code will need to be entered like this:

```
lea eax, d,[ebp-20]
push eax
push 65
push d,[ebp+8]
call 6E6
```

Press F9 to make the changes permanent when you're done. Now that we have our keygenerator in place, all that remains is making sure it gets called.

```
:004010B6 85C0            test eax, eax  ;if eax is not 0,
                                          you entered the wrong code
:004010B8 0F8519000000    jne 004010D7   ;<- an easy to manipulate jump
```

Thats the jump we're going to 'hook'. We need that to jump to our keygenerator code, which we stored at offset 700h in the .exe file. Back to HIEW, we go to edit the ASM at .4010B8 / 4B8 and make it do:

```
jne 700
```

Now it should jump to our keygenerator when a wrong serial is entered...

NOTE:
Do not execute the program at this point. It is unstable!

We need to do one last thing. We need to jump back and resume the program after we've run our keygenerator. Lets see, what would have happened if we hadnt intervened with the normal course of the program... It would have jumped to 4010D7, which is File Offset 4D7 (You may want to check this yourself, go ahead. I'm in no hurry ;) So let's just jump back there and pretend nothing has happened. Go to HIEW and edit the ASM at the end of our keygen at .40130E / 70E :

```
jmp 4D7
```

Ok, lets see what would happen now IF we were to run the program.

We enter the correct serial -¿ Everything goes like before The protection isnt fiddled with We enter the wrong serial -¿ The program jumps to our keygenerator -¿ The keygenerator enters the correct serial in the registration box -¿ It jumps back to the original code -¿ the original code displays a 'Bad Cracker' messagebox Seems like we left nothing out, lets do a test run.. And start the now modified InjectMe.exe Try entering your name again, and press the 'Type 1' button. Just like you would using a keygenerator.

WHOA! I CANT BELIEVE IT! IT REALLY WORKS!!!!

It should be working like predicted. We have now successfully installed a keygen right into this program. And WITHOUT taking a lot of time to reverse the hashing algorithm. We've effectively used the program against itself.

Let's make one small change for the perfectionists out there. The Messagebox is still showing, lets remove that. Two ways to do this are:

- NOP-ing out the MessageBox, as if it were a nagscreen
- Jumping back after our keygen, not to 4010D7.. But to right AFTER the MessageBox

Do not underestimate NOP-ing in situations like this. For instance.. If some important code is executed BEFORE the messagebox, we couldnt just jump past all of it, because then we'd miss that code too.. We'd simply NOP away the messagebox and be done with it.

However, since simple NOP-ing is overexplained already anyway in thousands of tutorials around the web, I will choose the second path. Lets take a look at what happens at location 4010D7 in W32Dasm:

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004010B8(C)
|
:004010D7 6A00           push 00000000              ;uType  (MB_OK)

* Possible StringData Ref from Data Obj ->"Bad Cracker"
                                  |
:004010D9 6835304000     push 00403035              ;lpCaption

* Possible StringData Ref from Data Obj ->"Try again"
                                  |
:004010DE 682B304000     push 0040302B              ;lpText
:004010E3 FF7508         push [ebp+08]              ;hWnd

* Reference To: USER32.MessageBoxA, Ord:01BBh
                                  |
:004010E6 E8EF010000     Call 004012DA              ;Call MessageBox
:004010EB E96C010000     jmp 0040125C
```

I've already commented the pushed parameters there, according to win32.hlp:

```
int MessageBox(
    HWND hWnd,          // handle of owner window
    LPCTSTR lpText,     // address of text in message box
    LPCTSTR lpCaption,  // address of title of message box
    UINT uType          // style of message box
);
```

So, this code is basically:

```
MessageBox(hWnd,"Try Again","Bad Cracker",MB_OK);
```

Yup, thats the one thats been bugging us. So, if we skip JUST the MessageBox, we end up jumping back to .4010EB / 4EB

In this case it seems rather silly to go there, because there's a jmp there right away as well, to 40125C. But I've decided to teach you good reversing habits, so we will simply jump to .4010EB /4EB In HIEW's ASM mode, go back to 70E again, and edit the code (F3, F2) there to read:

```
jmp 4EB
```

Start InjectMe.exe again, YEA!!! WE DID IT! It looks perfect now. This type of protection is DEFEATED! Reversers - Protections : 1 - 0

If something went wrong or you dont understand, you can take a look at my version called InjectMe1.exe which is enclosed in the zip file.

B. Protection 2 - Common

Needless to say, I won't start from scratch describing this protection.. It is very similar to the first one, with some small changes to make this slightly more difficult. Its code starts at 401121h.

```
:00401121 6A14        push 00000014                ;nMaxCount
:00401123 8D45EC      lea eax, dword ptr [ebp-14]
:00401126 50          push eax                     ;lpString
:00401127 6A64        push 00000064                ;nIDDlgItem
:00401129 FF7508      push [ebp+08]                ;hDlg

* Reference To: USER32.GetDlgItemTextA, Ord:0102h
                                   |
:0040112C E8A3010000  Call 004012D4                ;Call GetDlgItemTextA
:00401131 85C0        test eax, eax                ;Something entered?
:00401133 7461        je 00401196                  ;No? Fill in "Lamer"
:00401135 8D55EC      lea edx, dword ptr [ebp-14]  ;
:00401138 52          push edx                     ;lpName
:00401139 E845010000  call 00401283                ;Hashing function
:0040113E 8945DC      mov dword ptr [ebp-24], eax  ;Store hash at [ebp-24]
```

Now that we've got that out of the way, we can get to the parts that ARE different.

Say the program gets the serial again.. But this time, not as a string. It only stores it as a number, never as a string we could easily use and display.. The Windows API can do this. It has the function GetDlgItemInt which, unlike the rest of its family, does NOT return a string anywhere. It just assumes the value is a number, converts it for you internally and returns it in EAX.

Protection 2 uses this approach. Like I said, it is slightly harder. But should still cause no problems. Lets look at the next bit of code, which you should see uses the approach i just mentioned:

```
:00401141 6A00          push 00000000    ;bSigned
:00401143 6857304000    push 00403057    ;lpTranslated
:00401148 6A65          push 00000065    ;nIDDlgItem
:0040114A FF7508        push [ebp+08]    ;hDlg

* Reference To: USER32.GetDlgItemInt, Ord:0101h
                                  |
:0040114D E87C010000    Call 004012CE    ;Call GetDlgItemInt
:00401152 833D5730400000   cmp dword ptr [00403057], 00000000
                           ;bSigned = false?
:00401159 744F          je 004011AA      ;Yes? Fill in 0
```

A quick explanation of this might be in order.. I'll make it brief, dont worry. Basically, all you will need to understand is the GetDlgItemInt function and what it does. The Win32 API reference is really indispensable for any kind of decent reversing. But, as before, I will give you its definition (you better be downloading it in the background if you still dont have it ;):

```
UINT GetDlgItemInt(
    HWND hDlg,              // handle to dialog box
    int nIDDlgItem,         // control identifier
    BOOL *lpTranslated,     // points to variable to receive
                               success/failure indicator
    BOOL bSigned            // specifies whether value is signed or unsigned
);
```

The results of this bit of code (all of it, not just the call), is as follows:

- If there was a valid number entered, EAX will have it
- If there was no valid number entered, the code will jump to 4011AAh to enter 0 as the registration number

Now we've reached the core of this type of protection, a simple compare:

```
:0040115B 3945DC        cmp dword ptr [ebp-24], eax
                        ;is hash equal to the registration code?
:0040115E 0F8519000000  jne 0040117D
                        ;no? Jump to "Bad Cracker" MessageBox
```

From a coders perspective: Now they are simply compared as numbers, without going through the mess of creating a buffer and calling wsprintfA and lstrcmp..

> *SIDENOTE:*
> *That explains why this method is a lot more common than protection 1. Its simply easier as well as more logical for the coder. It makes little sense to got through all that trouble of converting to strings. (Of course, in this protection the registration number still has to be converted to a number before comparing. But this is all invisible to the coder because it is done by the GetDlgItemInt function)*

Knowing how to inject a keygen for protection 1 basically means we know how to do this one.. Instead of a SetDlgItemTextA, we now just use SetDlgItemInt. (Which is already imported too ;) I won't explain how to do this, since we've already done basically the same thing in the previous chapter. Instead, I'll pretend this is harder than it really is. I'll pretend SetDlgItemInt isnt imported.

We now have two options:

- Convert the number to a string and then use SetDlgItemTextA
- Import SetDlgItemInt and use that

I'll go for the first option in this case, since wsprintfA is already imported (if we'd have to import that ourselves, we might just as well have imported SetDlgItemInt straight away) Don't worry, I'll demonstrate the 2nd method in the 3rd protection (a worst case scenario). Let's think about what we're going to do again. We're going to insert the keygen at this point:

```
:0040115B 3945DC       cmp dword ptr [ebp-24], eax
                       ;is hash equal to the registration code?
:0040115E 0F8519000000 jne 0040117D
                       ;no? Jump to "Bad Cracker" MessageBox
```

This jne will then point to our keygen code, as before. The correct serial will be stored at [ebp-24] in number form. A quick summary of what our keygen will do:

```
push RightSerial       ;dword ptr[ebp-24]
push lpFormatString    ;the address of "%lu"
push lpBuffer          ;the address of some empty space..
call wsprintfA         ;conversion to string
push lpBuffer          ;the address of the right serial now in string form
push controlID         ;the registration number editbox ID
push hDlg              ;hDlg of main dialog
call SetDlgItemTextA   ;set the text
jmp  BadMessageBox     ;continue the original jump we took
```

Filling in what we know already gives us:

```
push dword ptr[ebp-24]   ;we just learned this one
push 00403009            ;from the old wsprintfA remember? ;)
push lpBuffer            ;we'll have to find some empty space
call wsprintfa           ;we need to file offset of it, which we dont yet have
push lpBuffer
push 65                  ;from the GetDlgItemInt for example
push dword ptr[ebp+8]    ;the hDlg is used in lots of functions
call 6E6                 ;we learned this function's offset in protection 1
jmp  BadMessageBox       ;we dont know its file offset yet
```

Nice, only a few small things to check, and we can finish this.

NextLine:
We already know the virtual address of BadMessageBox, it is (needless to say I hope): 40117D. To know its offset we select the line in wdasm, and look at the status bar. Its offset is 57Dh. Thats one param done.

wsprintfA:
Again, we know its virtual address. We can see that by looking at a call to it. Remember in protection 1:

```
* Reference To: USER32.wsprintfA, Ord:02A5h
                                  |
:00401090 E827020000              Call 004012BC   ;call wsprintf
```

Thus, its Virtual Address is 4012BC. Again, we select the line at 4012BC and look at the file offset. It is 6BCh.

lpBuffer: This is the tricky part of this protection.. You need a quick storage place to store your string. Let's be neat about it, and keep code and data seperated. (So we wont have to fiddle with the PE object flags). So, we will look at the '.data' object, where the program stores its data too.

```
# Name VirtSize  RVA PhysSize Offset Flag
3 .data 0000005B 00003000 00000200 00000A00 C0000040
```

Alright, seems there is 200h-5Bh=1A5h bytes left over in there. Should be enough right? Lets dump our buffer somewhere in there.. Let's say we take a nice round number, and pick 100h inside the .data object... That will be at Virtual Address (You should know how to do this math now): 400000 + 3000 + 100 = 403100

Told you, a nice round number. We have 100h bytes there left over.. Thats 256 characters, and thats WAY more than we need for a simple number. (highest number is 2^{32} = 4294967296 , which is 10 characters + a terminating '0'.. never forget that one.) Ok, now we have enough to complete our ASM source (already converted to HIEW syntax):

```
push d,[ebp-24]
push 403009
push 403100
call 6BC
push 403100
push 65
push d,[ebp+8]
call 6E6
jmp  57D
```

You can edit that in now at offset 700h (why change old habits?) The syntax should be correct i think, so HIEW should accept it. All that remains is having to make our keygen actually get called. To do this, we look back at the conditional jump at 40115Eh (offset 55E) and change that to a nice

```
JNE 700
```

to jump to our keygen in case of a wrong serial. Same as before. Make the change now, and press F9 again to update the file. When you're done, try running the program and using protection 2. You should see similar results as with the previous chapter.

Whoa! Niiice...

Lets remove the ugly MessageBox again, and change the last JMP in our keygen to

```
JMP 591
```

(If you don't understand why, I recommend rereading this bit of the previous chapter again) We're all done here. Another protection failed, and used against itself... Reversers - Protections : 2-0 (InjectMe2.exe is my .exe file at the end of this chapter, if there is something not entirely clear, it might clarify things for you)

C. Protection 3 - Worst Case

As the title says, this will be the worst case scenario. That means, we will not use any of the easy parts, and pretend they're all not there. No use of SetDlgItemInt, no SetDlgItemTextA, and a few more things we might have taken for granted now. You will see what I mean. The reason for this chapter is simple. This way you'll know what to do no matter how the target is constructed. A normal program will not have all these complications at once probably, but it is common that one of them will occur.. The average protection is a combination of this chapter and the previous one.

The code for this protection starts at 4011C7h

You can look at the disassembly for yourself, and you will notice the code to get our name and serial are the exact same as protection 2.. The hashing of the username is the same.. It's stored at [EBP-24], just like before. Everything is EXACTLY the same.. Up to the point where the two serials are compared, at 401201:

```
:00401201 3945DC  cmp dword ptr [ebp-24], eax
                  ;is hash equal to the registration code?
:00401204 7516    jne 0040121C
                  ;no? Jump to "Bad Cracker" MessageBox
```

Are you kidding me? Thats exactly the same as before!

If you look closely at the JNE.. You will see a vital difference with the previous ones. I will put them next to eachother to make it more obvious:

```
:0040115E 0F8519000000            jne 0040117D
          ^^^^^^^^^^^^
          6 bytes

:00401204 7516                    jne 0040121C
          ^^^^
          2 bytes!
```

How can that be? They're both JNE instructions...

There are different types of jumps. Some are short jumps (255 bytes maximum), some are far jumps(2^{32} bytes maximum) Some are relative from the current location.. Some are constant. etc, etc.. You get the idea. This is our first problem in the protection. It is a short, relative jump. To illustrate:

```
:00401204 7516      jne 0040121C  ;jump if not equal
```

75 = short relative jump

16 = SIGNED (-128 to +127 bytes) offset from the next instruction which is at 401204 + 2 (=size of the JNE) = 401206

(401206 + 16 = 40121C, as wdasm already translated for us)

Now think about why this might cause a problem. Right. We can not jump to our keygen code at 401300, because it is out of range. The furthest it could have jumped is 401206h + 127 = 401285h (Note that 127 is in decimal, while the address is in hexadecimal) We will need to find another way of dealing with this... I've thought for a moment what the neatest technique would be to teach you. (As in most reversing matters, there are many ways to do things. You are only limited by your own skill and imagination) The simplest way that occurs to me (though it still seems a little sloppy), is overwriting the "Bad Cracker"-MessageBox with a far jump to our keygen code. We aren't going to use the MessageBox anymore after all, right? So, to summarize, we will still use the old JNE to jump to the "Bad Cracker"-MessageBox. But once there, the code will now

only be a far JMP to our keygen code. (note that it is now and unconditional JMP, since the program has already determined we have entered a wrong serial number) We now can not jump back at the end of our keygen and still show the MessageBox. Instead, we will jump back to the code right after it, just like we did before.. This time 'right after the MessageBox' is Virtual Address 401230h. (File Offset 630) I'll assume you know by now how I got these values.

Now we know how to modify the program to jump back and forth to our keygen, let's focus again on the keygen itself. We will have to somehow display the value that's stored at [EBP-24], same as last time. Showing you the same thing again would be rather pointless I'd say, so let's "invent" some more complications.

Let's for example assume that neither SetDlgItemInt and wsprintfA are imported.. Now we're in trouble, since there is no easy way left to convert the number at [EBP-24] into a string to be put in the editbox.

Again, there are a lot of ways to solve this kind of complication. I've decided to just import SetDlgItemInt myself.

Let's do this API importing the proper way, by putting it in the import table of the PE file. The 'other' way would be calling GetModuleHandle and GetProcAddress to find the location of the function. You run into trouble with this method though, because not all programs import both these functions. (This program is a good example, it doesn't import GetProcAddress) I've considered going into detail now about how to change the import section of this executable, but I've decided to stay on-topic this time, because I fear I would be drifting too far away from the matter at hand, which is the keygen Injection. So, we will use a ready-made tool to add the import instead of doing it by hand with a hexeditor. A tool to do this is called iidking and was made by SantMat of Immortal Descendants. Lets download and run it! It will allow you to specify a .dll and a function inside it, which you want to import in your program. It will also generate a text file specifying where you can call the function (In our case "InjectMe3.exe.txt"). In our case, we select the .exe file, enter "User32.dll" as dll name, and "SetDlgItemInt" as function. In the text file it shows us exactly how we can now call the SetDlgItemInt function, so we no longer need to 'look it up' in a disassembly. In this case the files says:

```
SetDlgItemInt: call dword ptr [40506C]
```

> *NOTE:*
> *This program does some serious messing around inside the .exe file. It will still run, but programs like wdasm and hiew will start complaining the file is 'strange'. In other words, you may want to keep an original copy of the .exe for looking things up.*

We can now use the SetDlgItemInt function at 40506Ch. This function is very useful for what we want to do, because it's a combination of two things:

It converts a number to a string
It writes this string to a dialog item (like SetDlgItemTextA)
The parameters it takes are as follows:

```
BOOL SetDlgItemInt(
   HWND hDlg,          // handle of dialog box
   int nIDDlgItem,     // identifier of control
   UINT uValue,        // value to set
   BOOL bSigned        // signed or unsigned indicator
);
```

We already have the first 2 from the other protections. The uValue is the correct serial, which -as noted in the previous chapter- is stored at [ebp-24]. The last parameter we will need is bSigned, which is TRUE if uValue is a signed number, and FALSE if it's unsigned. It is unsigned, so we will push 0 (FALSE). We once again have all we need.

So let's see the code:

```
push 0           ; bSigned
push d,[ebp-24]  ; uValue
push 65          ; nIDDlgItem
push d,[ebp+8]   ; hDlg
call d,[40506C]  ; SetDlgItemInt, freshly imported :)
jmp  630         ; 401230h, remember?
```

Let's put it into the .exe file, at 700h as we did before. Don't forget to add the unconditional JMP at 40121C/61C to make the program actually USE the keygen.

Fuck me.. It actually works.

Even with complications like this it's still fairly simple to inject a keygen, as long as you know the right tricks. I'll end this chapter by saying: Use Keygen Injection. Use it wherever you want. It is quick, it is clean. It is for some odd reason, heavily underused. Hopefully this essay will have explained all there was needed to know to understand this technique, there really isn't much more to it. The rest is just tips tricks.

IV. Final words

These were all fairly simple protections.. Most protections are in essence as simple as the first 2 protections. In the last chapter I've given you two of the most common problems, and handed you two solutions. If you can think of another way to solve them, by all means do so. I change my methods constantly, whenever I think of something new. There are of course also a lot of other (less common) problems which *might* arise when injecting a keygenerator. Don't get discouraged, quite often they are rather easy to solve. If you feel you have tried all you can, come into #Cracking4Newbies and ask a question.

> *NOTE:*
> *Please think what EXACTLY you want before asking, because we can't and won't help you with questions like "I'm stuck. Help." Cracking is a difficult hobby, we can't explain every little thing every time. Thats why we write essays ;)*

I have plans for a next essay with 'tips & tricks of KI'. It will be a short reference featuring problems and solutions encountered by me or other people. It will be published in my Webbithole when it is made and ready.

-[The End]-

Kwazy Webbit

V. Greetings

Greetings go out to:
The oldleetos. You know who you are.
The reversers i know from c4n and other places around the net.
My friends in real life: Why the hell are YOU reading this?! :P
Most importantly: The people who think further than others. The select few who have innovative ideas. Thank you for having your own opinion.

For comments and suggestions, send an e-mail to:
KwazyWebbit(at)hotmail(dot)com