Ghiribizzo's Cracking Tutorial

IDA & Keygens for Newbies

***** BETA VERSION *****

Basic Techniques

I've been asked a few times to write an IDA tutorial. It seems that newbies struggle with IDA even when they can use W32Dasm perfectly well. Here I will cover my Ghost cracking in more detail with a focus on IDA & keygen technique.

PGP and Signed Tutorials

My tutorials and programs should be signed electronically using PGP. PGP 5 supports DSS/Diffie-Hellman keys. These keys are not supported by previous versions of PGP.
You should check the signature to make sure that the tutorial and especially its program files have not been tampered with. All cracks, tutorials and zip files I release will be signed. This will prevent tampering and will hopefully reduce the chances of viral infection.
My signature will also be the only way you can identify me as my email address will often change.

My Web Site: http://Ghiribizzo.home.ml.org
My Email: Ghiribizzo@geocities.com
My Backup Email: Ghiribizzo@hotmail.com

Introduction

I've been asked a few times to write an IDA tutorial. It seems that newbies struggle with IDA even when they can use W32Dasm perfectly well. Here I will cover my Ghost cracking in more detail with a focus on IDA technique.

To teach you how to use IDA, I'm going to assume you know how to use it. This might sound a little strange. I'm going to teach you a *way* of using IDA for cracking, not 'how IDA works'. Anybody should be able to use the IDA help and experiment with IDA to find the features and know the 'mechanics' of using IDA – if not, before trying to become a cracker, you should try to become computer literate!

This tutorial will take the form of a talk-through. I'll talk you through some of the steps I took when cracking Ghost but will leave most of the reversing bits for you to do. Should you get stuck at any point, you can refer to my IDB file for hints.

What's wrong with W32Dasm?

The main problem is that it is dumb and disassembles straight through a file. Normally this wouldn't be too bad if we could manually designate sections as code or data but this feature is unavailable. Even early disassemblers had this feature (e.g. bubble chamber).

What's good about IDA?

Quite a bit. Too much for me to go into now. Suffice to say, there have been some cracking projects where I have been happy to perform entirely in IDA whereas using only w32dasm just wouldn't cut it.

Following on From ghiric30.pdf

The first thing to locate is where your information is taken. You should quickly be able to find the relevant parts of the protection scheme and be able to identify the simple procedures such as the uppercasing and string length routines.

From here on, it's up to your own experience and expertise to figure out exactly what is going on and how to proceed. **– From previous Ghost tutorial**

Well if you read that and didn't find locating the protection scheme and relevant routines simple, then this tutorial may be for you! :-)

Later versions of Ghost have now been released so it may be difficult to find the original exe. So I have included it as part of the ghiric30 zip package.

Ghost.exe	-	target file
Ghost.idb	-	my original idb file
Ghiric32.pdf	-	this tutorial

The first step is to configure IDA so that you can see what is going on. Go into your IDA directory and find ida.cfg. Then find the line which looks like

```
SCREEN_MODE             = 0          // Screen mode to use
                                     // high byte - cols, low byte - rows
                                     // i.e. 0x5020 is 80cols, 32rows
```

and change it to something like this:

```
SCREEN_MODE             = 0x8440     // Screen mode to use
                                     // high byte - cols, low byte - rows
                                     // i.e. 0x5020 is 80cols, 32rows
```

This will make the default IDA screen 132x80 so you can actually see what is going on. I suspect that many newbies have problems because when the screen is used at it's default size, IDA is almost impossible to use effectively.

Now that we have made this simple adjustment, run IDA and choose ghost.exe as the file to disassemble. This will take a while – one of the disadvantages of using IDA. When a cracking project is simple, it is much better to use W32dasm and save time. Experience will help you judge which tool to use.

Now while it is disassembling, open up a dos command window and run ghost to see what it is like. If you're running NT press ignore when it complains about direct disk access and ignore the complaints form ghost too. We get a splash screen and notice that the product is already 'timed out'. When we press a key we are faced with a registration screen. Let's fill out the form with garbage: we are asked to confirm the details are correct then we get and error message box. We already have plenty of information to help us locate the protection.

We simply search for raw text in the file which corresponds to either the messages displayed on the screen or the screen paints. For example, let's search for 'Invalid license/key combination'. We move to the top of the file *ctrl-pageup* and 'search for text in core' *alt-b*. We type in the text we want. Normally, I do a case insensitive search on a substring e.g. **"invalid lic"**. Press enter to start the search.

The first hit we get is the correct one. We see the text in full and a xref to the code we want. Double click on the xref to jump to the section of code (**Tip**: pressing esc will take you back. The escape key is vital for navigation). Notice that the code we want is in fact immediately after the text. I named the whole procedure 'main_bit' as you can see from examining my idb file. Renaming locations is a powerful feature and gives you vital landmarks which will aid you should you leave your project for any length of time.

Notice that IDA nicely marks xrefs to ASCII strings with something like 'push offset aAcceptRegistra'. With these references we can immediately begin to rename locations and deduce the structure of this section of code. Do this now. Rename procedures and locations to more appropriate names and comment simple conditional tests which are relevant (**Hint**: don't bother looking deeper into any of the functions, deduce what you can from this 'level' only). I urge you to make an effort to look and analyse the code instead of just referring to my idb if you want to actually learn something. When you are done, take a look at my idb (**Tip**: you may want to load up a second instance of IDA with my idb in it to make comparisons easier). **Important**: in order to get the most out of this tutorial, you should not refer to my idb file except when I direct you to it as I will ask you to reverse many of the functions yourself.

You will have hopefully labelled as many functions and locations as I have. The call/or/jnz combination leading to good and bad messages should have stuck out like a sore thumb. The procedure which collected data from the fields should have been relatively obvious as there were exactly 6 calls and 6 fields. The rest was also pretty simple. In case you haven't yet gathered, use of IDA requires knowledge of assembly – if you were stuck, read up on assembly and practice before going further. I know of some 'crackers' who just load up w32dasm, look for the text reference and basically invert the first jump before it. I hope I'm dealing with crackers of a higher calibre here, or at least those who aspire to be of a higher calibre.

Note: Sometimes, I go through IDA and label the functions in the section of code I'm interested in s1,s2,s3… and their subfunctions s1s1, s1s2… etc. I do this as the renamed functions are yellow and stand out and so the overall 'layout' of the code is obvious. I label locations l1, l2 etc. So if you find names such as s7l3, you'll know what these are! After they have been re-labelled, I then reverse the various calls. This is one way of using IDA.

OK. Let's take a look at the function I called '*GoodOrBad?*'. It is clear that the rest of the check is contained within and a return of ax=0 implies a false serial.

Within the first few lines of the function we have 3 calls (2 are the same) a comparison and a conditional jump. **Task**: reverse entirely the two different calls and deduce a property the license number

must satisfy and hence deduce which jump must be taken. If you have never done DOS reversing before, I'm sure you'll find the simple little tasks such as these very rewarding and enjoyable.

Now take a look at the code near loc_1C18_18BE. **Task**: deduce the four possibilities for the first digit of the serial number. **Question**: Look at location 18CD. Does the choice of the first digit have any immediate impact on the flow of execution?

Task: analyse the next few lines of code and deduce the two possible values for the second digit. Analyse the code up to 191A and deduce the possible values for the remaining digits of the serial number.

The bulk of the protection lies in the two functions I have named *get_serial_crc* and *get_key_crc*. These two functions basically generate a number from the license number and key and then a check is made to see if the two numbers match. We will make a key generator which takes a given license number in the correct format and generates the key for it. To do this, we must understand what the two functions do.

While writing this tutorial, I decided to take a break from it by analysing the *get_key_crc* routine. I mentioned that it was surprisingly linear in my previous tutorial so I decided to take a few data points to try and find a mathematical approximation to the function. After a few points, I spotted what should have been blatantly obvious :-) Take a look for yourself! The *get_key_crc* function can be reversed by black box technique alone. Run the code, feeding the function various different numbers and what it does should be obvious. The *get_serial_crc* function can be reversed using IDA alone. But to reproduce it, we need to get some data which is more easily obtained using SoftICE. The functions make use of two tables. In my IDB I have indicated at cseg01:3B60 a table look-up (the protection does this in other places but I haven't indicated those). Unfortunately, simply clicking on 1AE1h will not take you to the table as IDA doesn't know which segment it is in. You can find it by either fishing for the first few bytes in SoftICE and then doing a hex search, or by checking likely segments. There are only 2 data segments so this is probably the quicker option. In fact, we find it in the first data segment.

- Press *Ctrl-S* for jump to segment
- Choose a data segment (dseg48 is the first data segment)
- Press *G* for jump to address and enter 1AE1 as the address to jump to
- You should land at the first data table
- Press *U* to make the bytes there unexplored
- Press * to define the bytes as an array
  - Now IDA suggests using 257 bytes as the array size (the maximum size) though if we look at the code, we can see it looks up [bx+di] (where di==1AE1h) so the maximum that can be addressed is 256 bytes. If you want to be careful, you can use the maximum size in the key generator.
  - Also, you can use the 'dup' construct or standard db construct. The 'dup' method is much more compact, but the db method allows you to see the data 'at a glance' so you have an idea of what it is like. It will probably be better to use the db construct during the cracking session and the dup construct when writing the key generator.
- Now you can copy and paste the code directly into your asm source! This is a really handy feature of using IDA for keygens.

The second data table is a bit harder to get at. We can see it referenced at cseg27:030C but sniffing for a few bytes for a search string under NTIce shows that the search string does not appear in the file! Do a search for 'Jean' if you want my guess as to why this is. It doesn't matter anyway, we'll fish the bytes from NTIce. You can try using a dumper, but this might be tricky as it is a console app. The simplest way is to perform a 'd' command from NTIce with the data window closed and then saving the session history using the command in the SoftICE loader. If you do this, you'll need to manually edit the file to delete the unwanted lines, then you'll need to put the data into the correct asm format. You probably should write a short utility to do this, but if you don't want to waste time on that, you can use something like Hex Workshop to search and replace to get the file in the correct format.

Once we have these two pieces of data, all we need to do is study the protection code and replicate the *get_serial_crc* functionality and then write a bit code to generate a key from our own *get_serial_crc* function.

Having taken a look at the code again, we see that the dword at dseg48:3190 (named tword2 in some versions of my IDB file) is located near the check for the 2nd letter. Examining the cross-references we see it near references such as: "*NetBios only available on Pro version*" which seems to confirm my initial guess. However, we can't register the program as Lite from the registration screen because it checks the status of tword2 (now renamed ProVersion?). It should be possible to create a lite version of Ghost, but that's something I don't have time for at the moment. Though if you do look into it, please email me and I'll include it in future updates of my tutorials.

*** BETA VERSION ***

OK. I've got a working version of my keygen now. All I need to do is stick a UI on it, then I'll post it with source code. I've re-installed Acrobat so this tutorial will probably be in PDF format.

I won't pack my IDB file with this 'pre-release version' instead I'll put then IDC file in and you can create my IDB by making your own IDB saving it to another file and running the IDC file onto it.

Intrepid should have uploaded the ghost file to members.xoom.com/tmoo/ghost41b.zip so you can find it there.

If you can't get it, please don't contact me for it as I will only send it when I release the finished version of this document.

Comments on this version of the tutorial to: ghiribizzo@geocities.com. I'll consider your comments when finishing the tutorial.