Codebreakers-Magazine

http://www.reverse-engineering.net
http://codebreakers.anticrack.de
http://codebreakers.reverse-engineering.net

Issue #1 - 2003

The Codebreakers Magazine – Issue #1 / 2003

http://codebreakers.anticrack.de , http://www.reverse-engineering.net

The Codebreakers Magazine – Issue #1 / 2003

http://codebreakers.anticrack.de , http://www.reverse-engineering.net

1 Disclaimer[1]

This magazine DOES NOT stand for pirated software, warez, vrii or crackz.

If you want an application to keep and use, buy it.This is about Reverse Code Engineering (RCE). Here you can learn how software works within the win32 environment.You can learn how the software was written and how to change it.You can learn to circumvent the different "protection" schemes.You can learn how to utilize the tools that the "experts" use.
We will NOT answer to any crack request or do cracks! As we are reverse-engineers we don´t crack at all.

The creator of this magazine or the ISP(s) hosting any content of this magazine take no responsibility for the way you use the information provided in this magazine.These files and anything else in this magazine are here for private purposes only and should not be downloaded or viewed whatsoever! If you are affiliated with any government, or ANTI-Piracy group, MPAA, CCA, Herrn Rechtsanwalt Günter Freiherr von Gravenreuth , Microsoft, BSA or any other related group or persons or were formally a worker of one you cannot enter this web site and download this magazine, cannot access any of its files and you cannot view any of the files. All the objects on this site are private property and are not meant for viewing or any other purposes other then bandwidth space. Do not enter whatsoever! If you enter this site you are not agreeing to these terms and you are violating code 431.322.12 of the Internet Privacy Act signed by Bill Clinton in 1995 and that means that you cannot + threaten our ISP(s) or any person(s) or company storing these files, cannot prosecute any person(s) affiliated with this page which includes family, friends or individuals who run or enter this web site. If you do not agree to these terms then you must close this document now!

The author gives no guarantee, that the described ways, programs and everything on this site are free from trademarks of thirds. The usage of trademarks, copyrights, names of wares (and so on) should not let you think, that these names are free and can be used by everyone.

This work (Codebreakers Magazine) is owned by the author. All usages beyond the frontiers of international and national laws is without the authors license forbidden and will be punished. These terms belong to any kind of copies, translations, microfilming and the saving and editing in electronic systems too.

The author is not responsible for any illegal and lawbreaking misusage and any illegal and lawbreaking usage of all described methods and programs - including the sourcecodes.

This is a private document.
The owner of this document is not responsible for any damages, which could result from the usage of this document or from the usage of the offered programs, sources, articles, algorithms and methods.

We have to make clear, that everybody breaks law, if he attacks systems and/or sponsors the attacking of systems.. We - www.AntiCrack.de - distances ourselfes from socalled WAREZ-Sites, which offer illegal copies. We do not offer illegal copies.

This magazine is thought only for educational proposes and usage.

The owner of this magazine declares this magazine as an artistic work and therefore as art.

If you are software-producer and you do not want that your software is descibed in any article please contact us. We will remove any article, link and download which belongs to you.

[1] Read it…

This also for all used logos. If you don't want that your logo is used by us (because you don't want this or there is an perhaps unknown to us set trademark), so contact us too. We will then remove all trademarks and logos of you.

If you are an author und you should find one of your articles here and you don't want this, please contact us. We will then remove all articles of you.

If you are software-developer and you have interest in an active working-process with us, to protect your software better, then contact us please (info@AntiCrack.de).

All described patches, keygens, reverse-engeneering-methods and all other topics which are to find on these pages (including methods for the topics Hacking and Virii) are thought for private educational usage and should help to protect software and other computer-relevant topics better (for example hacking and the protection from virii). Only if you know how you can be attacked, you can protect yourself.
This is the philosophy of AntiCrack. You should learn from negativ-examples and should make your techniques better.

AntiCrack is in generall not responsible for any illegal usage or misuage. The also includes the description of the reverse-engeneering-methods of commercial protection-systems.

Our ISP is not responsible for any damages, which could result from the usage of our files.

Only if you accept and aggree to the terms above and you do not want to use our informations (including all programs and downloadable files) for "misusage" or for illegal usage you are allowed to read our magazine.

With a law from 12. may 1998 the "Landgericht Hamburg" has decided, that by setting of a link any author is responsible for the content of the linked page. The author can only protect himself - so the LG - if he is distancing himself definetly from these contents. We have linked to other pages in the internet. For all these links: We definetly say, that we don't have any influence to the design and the contents of the linked page. We keep definetly distance from all contents of all linked pages on our pages. This disclaimer belongs to all links on our pages.

AntiCrack is NOT responsible for any abuse of the information we provide. Members of AntiCrack don't hack to destroy any system, to get data of foreign systems or to destroy this data. As a matter of fact, we don't hack at all, since we are reverse engineers. Our only objective is to further our knowledge. If you want to attack a system it should be your own !

1. Online-contents

The author reserves the right not to be responsible for the topicality, correctness, completeness or quality of the information provided. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected. All offers are not-binding and without obligation. Parts of the pages or the complete publication including all offers and information might be extended, changed or partly or completely deleted by the author without separate announcement

2. Referrals and links

The Author is not responsible for any contents linked or referred to from his pages - unless he has full knowlegde of illegal contents and would be able to prevent the visitors of his site from viewing those pages. If any damage occurs by the use of information presented there, only the author of the respective pages might be liable, not the one who has linked to these pages. Furthermore the author is not liable for any postings or messages published by users of discussion boards, guestbooks or mailinglists provided on his page.

3. Copyright

The author intended not to use any copyrighted material for the publication or, if not possible, to indicate the copyright of the respective object. If you find any unindicated object protected by copyright, the copyright could not be determined by the author. In the case of such a unintentional copyright violation the author will remove the object from the publication or at least indicate it with the appropriate copyright after notification.

4. Legal force of this disclaimer

This disclaimer is to be regarded as part of the internet publication which you were referred from. If sections or individual formulations of this text are not legal or correct, the content or validity of the other parts remain uninfluenced by this fact.

2 Welcome

Welcome to the first Issue of “Codebreakers-Magazine”.

In this magazine you will find lots of informations. The main audience for this magazine should be programmers and reverse-engineers but we even try to include some “Off-Topic” articles which could be with some interest for the audience.

As you can see:
This is the first release, so for sure not everything is like it should be. If you have any positive or negative critics feel free to contact as at codebreakers@anticrack.de.

If you have any ideas how we can get better or if you want to contribute an article contact us too.

So this issue seems to be a little big? Well, we try to include as much informations we can, but we tried to make a hard selection on what we publish. Anyway, we produced many pages. This magazine should not be a “collection” of four or five tutorials. It should be like a real magazine you can buy. The main part will be (sure) the Reverse-Engineering part. Additionally we included some interesting freestyle articles which are of interest. As well we have a look at the VX scene which always deals with interesting coding-techniques and anti-tricks.

Let me say thx to all the contributors which helped to realize this magazine. The issues will be released irregulary – this means the next release will be published during the next 2 or 3 months. If everything starts running as we want and we get more feedback, we are (maybe) able to publish every month. But first see how this issue gives feedback…

Zero – Main Author

3 General Community News

3.1 Ourselves and friends

After several hard attacks one year ago at RCE sites located at free hosting services, it seems to be calm at the moment. Mainly the bigger sites are still running and it seems that they resist any attack. Organisations like the SPA[2], the MPAA[3] or BSA[4] are still mainly focusing at the Pirating-Scene and the warez- and crackz-trading sites and it seems that they have no interest at the RCE sites.

Several so called “AntiCrack” sites are more than a joke.

http://anticrack.hypermart.net/ is one of them. They try to tell shareware authors how to shutdown warez sites. If you look at there links section, you can find this:

„Fravia's How to Protect Better
Well, if you really are interested in protecting your proggies, you must read this. It contains "Mark's famous 14 protector's commandments", "Tidbit's 'common sense' rules" and more. Rating: 9/10“

Well, to be honest I do not see any correlation between Fravia and these “Anticrack-sites”. Did they realised to which person they have linked ?

Next funny site is located at http://anticrack.virtualave.net/. What a useless site.

Other sites like http://www.cat-soft.com/warez.htm come with “The Battle Plan”. Did anybody every realised that there is a war ?

http://www.anticrack.org/ has nothing to do with anticrack or RCE.

All bigger sites focusing on RCE are still existing and it seems that this not change so far. Best examples are http://www.anticrack.de with several relocators or http://tsehp.cjb.net with their forum.

So we will see if these organisations try to attach to the anti-RCE process again and start the discussion if RCE is legal or not. At the moment they seem to have accepted that RCE is a legal science.

When even ladies like C. Cifuentes – who works as a professor – mainly focuses at decompiling applications and teach this as university lecture, we can see that RCE is more than simply cracking applications.

Zero

[2] http://www.spa.org
[3] http://www.mpaa.org
[4] http://www.bsa.org/

4 Interview with +Fravia

First let me say thank you for giving us this interview.

CodeBreakers:
When did you started with reverse engineering ? How old are you really ?

+Fravia:
I began reversing software protection schemes on a "sinclair spectrum" in the late seventies. The programs were on tapes, he :-) I am now in the second half of my life.

CodeBreakers:
What is your real occupation ?

+Fravia:
I am a senior bureaucrat, with a well paid life-long contract. My actual task is to concretize some really useful workflow gains from the huge "informatization" waves and investments made in the past 20 years. A very hard task :-)

CodeBreakers:
Hobbies ?

+Fravia:
Sailing, reversing propaganda, learning how to search the web better than anybody else and explaining people how to do it :-)

CodeBreakers:
What kind of books interest you most ? Any favorites ?

+Fravia:
Strategy, rhetoric, history.
There are so many "favorites"... would be a too long list.
At the moment I am re-reading Kurt Tukolsky's masterpieces. If you find them, enjoy!

CodeBreakers:
Your favorite drink ?

+Fravia:
Cocktail: a 'traitor' (you may find the recept somewhere on the web).
Wine: pomerol.
Beer: Ename.

CodeBreakers:
Did the knowledge / learning process of reverse-engineering influenced your life ? And if so, how ?

+Fravia:
It's the other way round: My life influenced reverse-engineering :-P

CodeBreakers:
You have influenced many people with your old RCE website. Actually many people took your webpages as a startup into the RCE scene. What do you think about this ? Ever thought that you could influence people wrong ?

+Fravia:
Not really. I am sure I have done my best to put a lot of people on the correct ethical path.
Reversing is a sine qua non in a world of zombies.
Reversing software is a good start into it.
Having transformed many crackers into software reversers, and some good ones into real reversers, is a nice reward for those years.

CodeBreakers:
if you could invite a personality to dinner who would it be? And which personality will be never invited ? Why ?

+Fravia:
What kind of 'personality' do you mean?
If we remain in the rev-eng scene I would love to dine with somebody like Saltine, or Mark Russinovich, or Bryce Cogswell (the two clever ones behind Filemon and Regmon).
I would never invite bogus reverse engineers like Cristina Cifuentes, though. You can divide people into those that actually do things and those that take the merit. But i have all the time "real life" encounters (mostly dinners) with many rev-eng good ones, like Ilfak Guilfanov, the genius behind Ida.
Some good reversers have also become good friends of mine, like Richard Stallman or Mammon_, and we meet often.

CodeBreakers:
If you were a kid nowadays, would you take on cracking?

+Fravia:
Of course, there are many nice games that I would like to try out without paying. As a matter of fact I would "take on" cracking whatever age I would have :-) But there is a difference between 'take on' and 'concentrate on'. I would NOT make reversing software my main activity, nowadays. Knowing how to search the web gives you MUCH more power (and makes you a much more "dangerous" fellow, whatever your --hopefully worthy-- aims may be).

CodeBreakers:
If you had a time machine, which event would you travel to witness in your own eyes?

+Fravia:
Audwin & Albwin times (longobards). I would visit Chramno's descendants in Aquitania. The end of the roman empire fascinates me. I see so many links with the actual times.

CodeBreakers:
What was the hardest protection you ever reversed ?

+Fravia:
The hardest? Well, that must be seen in the context of my capacities, which have grown over the years. I would say that Paint Shop Pro around 1996/1997 was hard (well such a protection was hard in those times for us at least, ask +Greythorne :-)

Mozpong is also a nice challenge, hard until you understand the trick, try it yourself :-)

The most recent 'somehow hard' one i cracked (for my own use) was poser 5.

CodeBreakers:
Is there any protection you have not been able to crack ?

+Fravia:
No. I have always cracked what I wanted, some time it took long, though. Note that I have never in my life made a serial generator. I do not spread cracks, I (did) spread knowledge on protection schemes, which is something quite useful for protectors, see
http://www.searchlores.org/protec/protec.htm
or
http://www.searchlore.org/protec/protec.htm.

The idea was always to study the different protection schemes, not to steal the software. If you want to steal software, just codebar in the shops. It's quicker, easier, and you get the manuals as well :-)

Yet, to be honest, there were also many protections that got so boring (or so effective, depends which side do you look at them :-) that I was compelled to apply some very 'dirty' cracking approaches. Such 'solutions' were not elegant at all, and some time even uncomplete or buggy. Maybe one could say I was not able to crack those targets.

CodeBreakers:
Which do you think is currently the best protection on the market?

+Fravia:
No idea, really. I stopped many years ago any software protection reversing activity that was not related with my immediate needs. The most difficult schemes usually protect 'niche' small products that normal people do not use.

CodeBreakers:
For a long time your website about RCE was one of the largest and best known ones on the web. Why have you decided to close it and why are you now only focussing on "searchlores.org" and on searching topics?
Tired of reversing ? If yes, why have you retired ?

+Fravia:
Cracking protection schemes for the sake of it is -as such- actually quite boring. Reverse engineering software (which means for instance documenting malwares practices) is slightly more interesting, but still pretty boring.
Both are necessary activities no doubt, but dull... I was simply bored.
 On the other hand searching the web, and learning how to find ANYTHING you may want (http://www.searchlores.org/targets.htm), is MUCH more important (and necessary) nowadays. As a matter of fact I always supposed it (you may remember that i already had a full-fledged 'how to search' section on my site), now I can tell you: I know it for sure.
Moreover:
reversing http protocols and entering databases is a sine qua non in a world
-like ours- of slavemasters and slaves.

CodeBreakers:
You have not been very "unpolitical" when talking about "social" structures. Do you really think that RCE and politics/socials should be combined ? Or should RCE be completely free from these "real-life" problems ?

+Fravia:
"Reversing" encompass much more than software reverse engineering. We live in a world where the 300 most rich individuals have a bigger yearly income than the 3 billions (3000000000) poorest ones. No kidding: 10.000.000 slaves per slavemaster. I do not agree with this. I do not like it. I have always done my best to counter it. Software reverse engineering is a small crumb of knowledge in the darkness where they keep us. Knowing how to search gives much more, as I know now, but that was already a good start.

Note also that nowadays there's no difference between searching the web, reversing software, reversing reality, and/or being politically active.

The moment you engage in any of these activities you will soon find, or bounce against, some unpleasant truths. You may swallow them, you may ignore them, you may try to get some personal gains out of them or you may try to debunk them. You will have to choose your own way. I did.

CodeBreakers:
You have published at your old website this document:
http://www.woodmann.com/fravia/io13.htm

This document is titled with "his real identity" and the text "Born on 30 August 1952 in Oulu..."
Actually this guy on the photo is not you.
Instead this is a photo of JRR Tolkien...
(http://www.mi.uib.no/~respl/tolkien/pic/large/jrrt-color.jpg)

Why this ? Do you like it to lead people to the wrong path ?

+Fravia:
With sites like mine you're a sitting duck. It is trivial to stalk someone that has a big web site, bound to drop hundred of useful "angles" that can be used to identify him. Making some "smoke" is the only way to make things a little more difficult for the stalkers. Those that are there have the purpose to mislead, even if there are also many truths, there. A newer better version can be found on my sites: http://www.searchlores.org/info.htm or http://www.searchlore.org/io13.htm

Note also that with the years my real visage is getting more and more similar to Tolkien's photo :-)

CodeBreakers:
When browsing your old pages, it is clear that you seem to have had a special "connection" to +ORC. +ORC seems to be like a ghost. Does +ORC really exists or is it just a myth/fake like the JRR Tolkien photo created by you?

+Fravia:
+ORC is dead. Died in Egypt. RIP.

CodeBreakers:
There is still the myth that there are some "missing +ORC lessons" existing. Do they really exist?

+Fravia:
Dunnow. I have seen only one in all these years.
Those that got them were bound not to publish them on the web, guess they are no more important now, anyway.

CodeBreakers:
Do you think the +HCU should be reopened ?

+Fravia:
No. Time has changed many things.

CodeBreakers:
Time has changed many things. Some (more) years ago copyprotections have been made with normal name/serial checks.
Today we are confronted with many different kinds of protections:
self-modifying-code, polymorphic code, encryption, running code backwards or anti-debugging tricks. All of them have been cracked but everything seems to be possible nowadays. Where do you think will future protections go ?

+Fravia:
Future protections will be enhanced per law, using systems that will compel users to register every move they do. The trend has already started within windoze xp. That's nothing, they will criminalize debuggers and disassemblers and compel all zombies to give out their Grandma's sexual tastes in order to beg to access a commercially more and more polluted web.

CodeBreakers:
Where do you think will all this world wide web information system go ?
To absolutely security like the developers of quantum cryptography promise or to absolute chaos where no rules exist ?

+Fravia:
To absolute chaos... with its many rules... as usual, yet I am sure we'll be able to circumvent part of them. If you think that reversing software is challenging and dangerous, wait until you begin to reverse "their" plans for the web.

CodeBreakers:
It seems to be a challenge for many people to do "Manually Unpacking" of a target. What do you think of this ?

+Fravia:
I am not a specialist in unpacking but I fail to understand the question:
There are many good essays that explain -even too much- how to do it.
There are more and more (good) tools to automatize completely, or at least in part, such work. Is there such a big challenge?

CodeBreakers:
How should the perfect protection be ?
And do you think that there will be some day a perfect one ?

+Fravia:
You concentrate too much on software "hardcoded" protections.
A better protection will be "web-based" and there is something of that taste already hidden in your computer... only they still use it for crap commercial purposes.
Access for instance my site with a bloated MSIE browser (the browser of choice for almost all zombies) instead than, say, Opera, and have a look at your own 'collar tag':
http://www.searchlores.org/supercookie.htm

Yep, that's you. Bet you did not know you were tagged like that. It would be trivial to port something similar to some kind of 'law-enforced' software protection scheme, if you see what I mean and fear.

CodeBreakers:
You are often talking about "Zen"-Cracking. There are many discussion what this is really. It seems that everybody has his own definition. How do you define "Zen"-Cracking ?

+Fravia:
It's easy: If you ever had a 'satori' experience in your life, you know what it is like. There's no apparent reason for a choice you make, for a path you follow. Not even your past experiences, yet it turns out to be the right choice, the correct path... Good old +ORC said that when the debugger is in the hand of a zen-cracker, it identifies with the man himself, it is no more a tool with no mind of its own. It acquires a soul. The cracker, emptied of all thoughts, all sense of insecurity, all desire to "crack", is not even conscious of using softice any more. Both cracker and softice turn inrto instruments in the hands, as it were, of the unconscious. Probably bullshit... until it happens to you.

CodeBreakers:
Imagine you have the free choice to code a software.
What would this software be ?

+Fravia:
A good simple disassembler, sorta wdasm updated.

CodeBreakers:
And would you protect it ?

+Fravia:
No. Why?

CodeBreakers:
All of your software legal bought ?

+Fravia:
Of course not. You HAVE TO defend yourself from all these beta buggy crap they sell and patch -maybe- later.

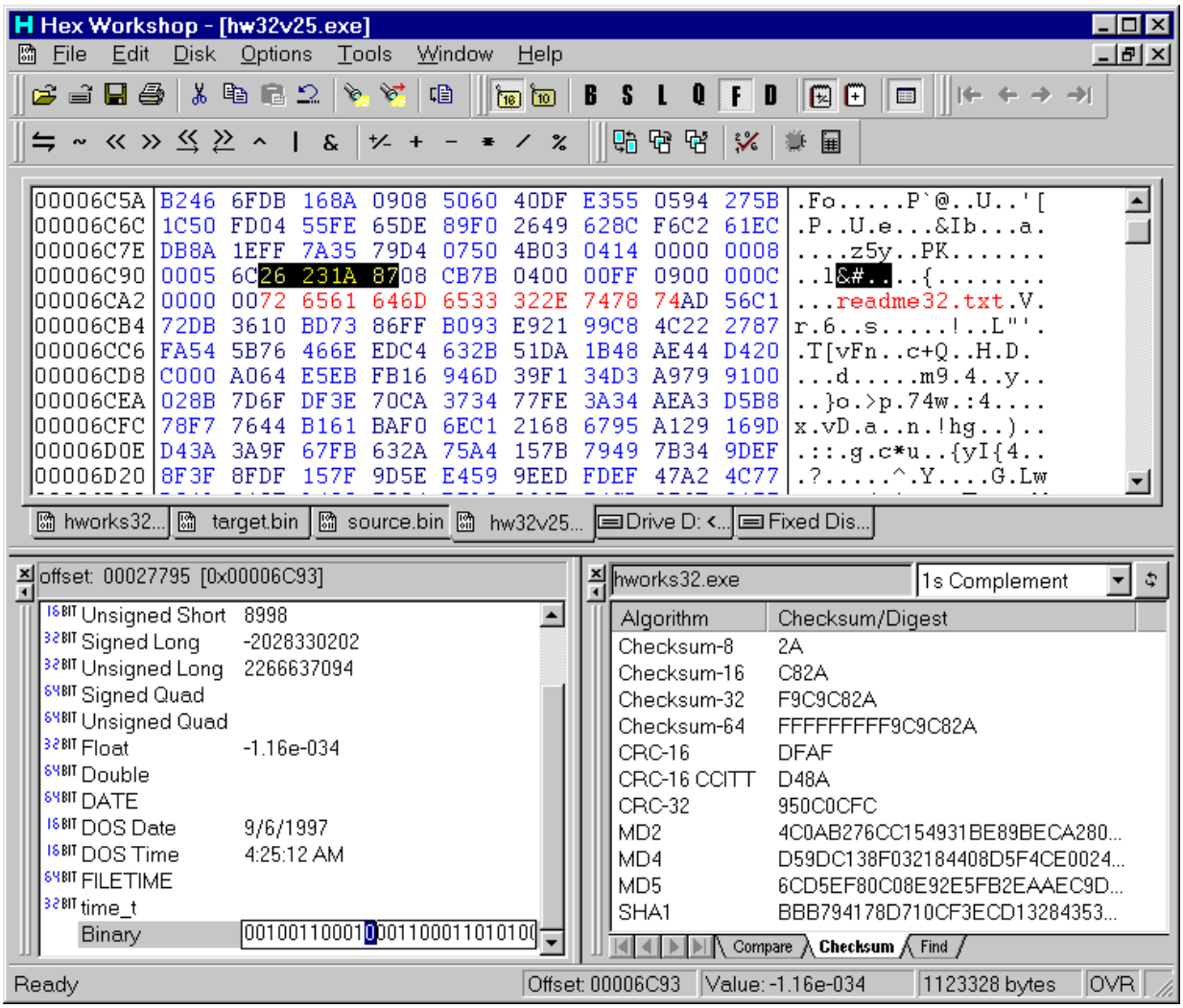
Yet I have bought afterwards most (well, yes, a lot) of the software I have respected and really used. Out of gratitude for the programmers. Will probably happen with poser 5, I may buy it as soon as those clowns sort out its many bugs.

CodeBreakers:
Thanks +Fravia for this interview !

5 Tool-Reviews

5.1 HexWorkShop 4.0

5.1.1 Official Statements

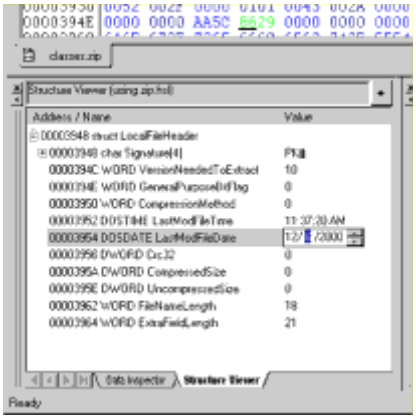


The Hex Workshop Hex Editor is a set of hexadecimal development tools for Microsoft Windows, combining advanced binary editing with the ease and flexibility of a word processor. With Hex Workshop you can edit, cut, copy, paste, insert, and delete hex, print customizable hex dumps, and export to RTF or HTML for publishing. Additionally you can goto, find, replace, compare, calculate checksums and character distributions within a sector or file.

BreakPoint Software is proud to annouce Hex Workshop v4.0. With over 13 new major features and several feature enhancements, Hex Workshop is by far the premium Hex Editor available on the market today. Hex Editing has never been simplier!

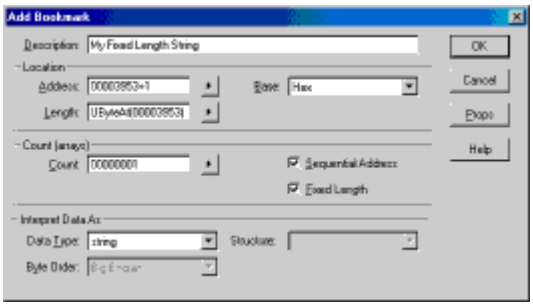
Please read through the key changes made in Hex Workshop v4.0 to learn why Hex Workshop v4.0 will be your hex editor of choice.

- New Structure Viewer
- New "Smart" Bookmarks
- New Color Mapping Feature
- New Custom Character Maps
- New Set Floor and Set Ceiling Operations
- New ASCII Text Manipulation Operations
- New Block Bit Shifting Operations
- New Redo Command
- New Copy Sectors Feature
- New Automatic Version Feature
- Enhanced Export Capabilities
- Enhanced Resynchronizing Compare
- Enhanced Find All in File

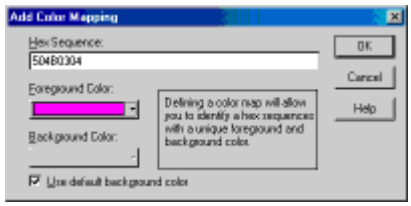


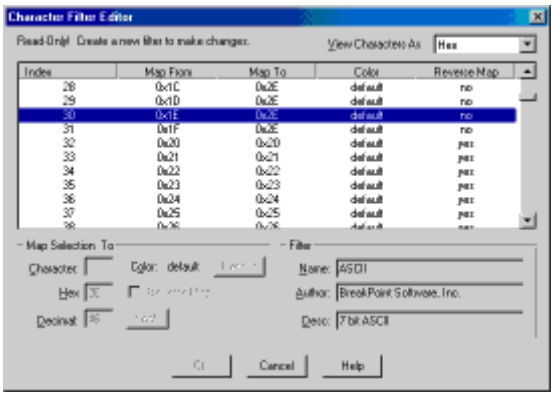
New Structure Viewer allows you to view and edit data in the most convenient and intuitive way. This helps turn the art of hex editing into a simple straightforward task. Structures are defined in a text file and closely resemble the C/C++-style struct definition. Many atomic data types are supported along with the ability to nest structures.

New Bookmarks Feature helps you pick apart and label a data file for a single use or to share with friends and co-workers. When defining a bookmark, you can set the bookmark's data type and then edit the bookmark's data directly in the bookmark viewer. You can even use a simple macro language when defining bookmark addresses and lengths. This allows for powerful "smart" bookmarks that are portable across like files.



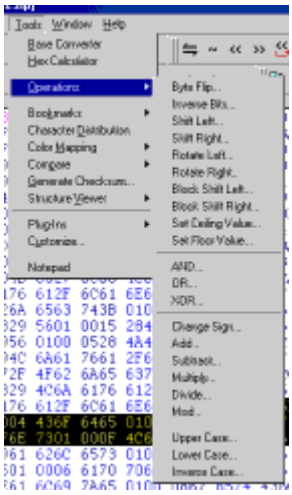
New Color Mapping Feature helps you visually find patterns, add visual cues, and de-emphasize uninteresting data. Users can set the foreground and background colors used to display data for any hexadecimal string.





New Custom Character Filters allow you to define your own mapping between a byte of data and the text character chosen to represent it. The character filter is also consulted when entering characters into the text area of Hex Workshop. Mappings can be either unidirectional or bi-directional.

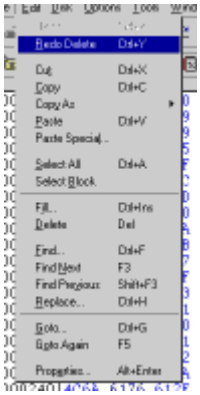
New Set Floor and Set Ceiling Operations allow you to set the lowest value or highest value within a selected range or throughout an entire file. Signed and unsigned 8 bit, 16 bit, 32 bit, and 64 bit integers are supported along with 32 bit and 64 bit floating point values.

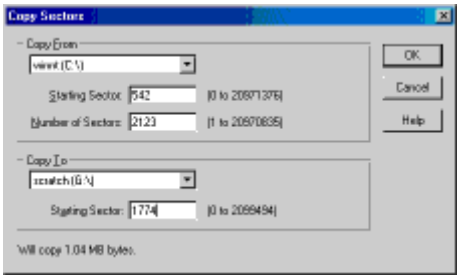


New Convert to Lower Case, Convert to Upper Case, and Switch Case Operations help you manipulate ASCII text within Hex Workshop. You can modify the case of ASCII text with a selection or throughout an entire file. Operations include converting all ASCII text to upper case, converting all ASCII text to lower case, or swapping upper case characters to lower case characters and lower case characters to upper case characters.

New Block Bit Shifting Operations allow you to shift bits across byte, word, dword, and quad boundaries. This flavor of shifting and rotate operations carries bits across data type binaries - An time saver if you find yourself faced with a large block of data all shifted by a few bits.

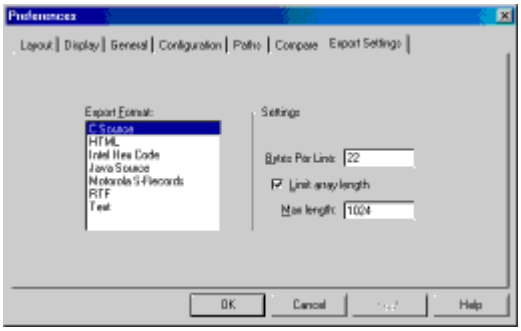
New Redo Last Command allows you to Redo your last n-Undos. By allowing you to specify the size of the undo/redo buffer under Preferences, the number of undo/redo operations are limited only by the amount of RAM set aside.





New Copy Sectors Feature allows you to copy sectors between different drives and drive partitions. This feature is incredible useful when attempting salvage data from corrupt disk drives, manually rebuilding drives and partitions, and cloning drives.

New Automatic Version Checking helps users to stay on top of latest release of Hex Workshop. Users can choose to check for updates manually or allow Hex Workshop to automatically check for updates once a month. No private or sensitive information is exchanged during version upgrade checks.



Enhanced Export Capabilities allow users to set the maximum array size for C or Java export. If a user tries to export data larger then the maximum size defined, Hex Workshop will output multiple chunks of data into separate arrays. This feature was added to address the a maximum array size limitation in some compilers.

Enhanced Resynchronizing Compare adds color to the Hex Workshop display area for compared files. Users can now visually see matches, insertions, deletions, and replacements in user-definable foreground and background colors.

Enhanced Find All in File adds color to Hex Workshop by display all instances of the last search string in a user defined foreground and background color. This allows you to easy and quickly scroll through a file and focus on your search results.

5.2 OllyDbg 1.08b

5.2.1 Official Statement

OllyDbg is a 32-bit assembler level analysing debugger for Microsoft® Windows®. Emphasis on **binary code analysis** makes it particularly useful in cases where source is unavailable. OllyDbg is a shareware, but you can download and use it **for free**. Special highlights are:

- **Intuitive user interface, no cryptical commands**
- **Code analysis - traces registers, recognizes procedures, loops, API calls, switches, tables, constants and strings**
- **Object file scanning - locates routines from object files and libraries**
- **Allows for user-defined labels, comments and function descriptions**
- **Understands debugging information in Borland® format**
- **Writes patches back to executable file and updates fixups**
- **Open architecture - allows for third-party plugins**
- **No installation - no trash in registry or system directories**

- Debugs multithread applications
- Attaches to running programs
- Configurable disassembler, supports both MASM and IDEAL formats
- MMX, 3DNow! and SSE data types and instructions, including Athlon extensions
- Full UNICODE support
- Dynamically recognizes ASCII and UNICODE strings - also in Delphi format!
- Recognizes complex code constructs, like call to jump to procedure
- Decodes calls to more than 1900 standard API and 400 C functions
- Gives context-sensitive help on API functions from external help file
- Sets conditional, logging, memory and hardware breakpoints
- Traces program execution, logs arguments of known functions
- Shows fixups
- Dynamically traces stack frames
- Searches for imprecise commands and masked binary sequences
- Searches whole allocated memory
- Finds references to constant or address range
- Examines and modifies memory, sets breakpoints and pauses program on-the-fly
- Assembles commands into the shortest binary form
- Starts from the floppy disk

and much, much more!

Why 1.08b

Two days after I have uploaded 1.08, a nasty new bug was reported: Assembler was unable to compile PUSH const. This error was a result of another last-minute bugfix. Version 1.08a corrected this frequently used command.

Next day, another red alert came: run trace saved invalid values of registers EAX and ECX. Due to importance of run trace in program analysis, I was forced to replace 1.08a with 1.08b. Another small correction removes possible GPF in heap window. Sorry...

5.2.2 Our Statement

Olly is still the best alternative for SoftIce. There are several positives which can be mentioned. First there is no installation needed. Just copying it to any directory you want and running it is one of the best features of it.
The author of OllyDbg has done several bugfixes and coding additions, so this tool is starting to rule the scene.

5.2.3 Quick Start

5.2.3.1 Frequently used menu functions

Function	Window	Menu command	Shortcut
Edit memory as binary, ASCII or UNICODE string	Disassembler, Stack Dump	Binary|Edit	Ctrl+E
Undo changes	Disassembler, Dump Registers	Undo selection Undo	Alt+BkSp
Run application	Main	Debug|Run	F9
Run to selection	Disassembler	Breakpoint|Run to selection	F4
Execute till return	Main	Debug|Execute till return	Ctrl+F9
Execute till user code	Main	Debug|Execute till user code	Alt+F9
Set/reset INT3 breakpoint	Disassembler Names, Source	Breakpoint|Toggle Toggle breakpoint	F2
Set/edit conditional INT3 breakpoint	Disassembler Names, Source	Breakpoint|Conditional Conditional breakpoint	Shift+F2
Set/edit conditional logging breakpoint (logs into the Log window)	Disassembler Names, Source	Breakpoint|Conditional log Conditional log breakpoint	Shift+F4
Temporarily disable/restore INT3 breakpoint	Breakpoints	Disable Enable	Space
Set memory breakpoint (only one is allowed)	Disassembler, Dump	Breakpoint|Memory, on access Breakpoint|Memory, on write	
Remove memory breakpoint	Disassembler, Dump	Breakpoint|Remove memory breakpoint	
Set hardware breakpoint (ME/NT/2000 only)	Disassembler, Dump	Breakpoint|Hardware (select type and size!)	
Remove hardware breakpoint	Main	Debug|Hardware breakpoints	
Set single-short break on access to memory block (NT/2000 only)	Memory	Set break-on-access	F2
Set break on module, thread, debug string	Options	Events	
Set new origin	Disassembler	New origin here	
Display list of all symbolic names	Disassembler, Dump Modules	Search for|Name (label) View names	Ctrl+N

Context-sensitive help (requires external help file!)	Disassembler, Names	Help on symbolic name	Ctrl+F1
Find all references in code to selected address range	Disassembler Dump	Find references to|Command Find references	Ctrl+R
Find all references in code to the constant	Disassembler	Find references to|Constant Search for|All constants	
Search whole allocated memory	Memory	Search Search next	Ctrl+L
Go to address or value of expression	Disassembler Dump	Go to|Expression Go to expression	Ctrl+G
Go to previous address/run trace item	Disassembler	Go to|Previous	Minus
Go to next address/run trace item	Disassembler	Go to|Next	Plus
Go to previous procedure	Disassembler	Go to|Previous procedure	Ctrl+Minus
Go to next procedure	Disassembler	Go to|Next procedure	Ctrl+Plus
View executable file	Disassembler, Dump, Modules	View|Executable file	
Copy changes to executable file	Disassembler	Copy to executable file	
Analyse executable code	Disassembler	Analysis|Analyse code	Ctrl+A
Scan object files and libraries	Disassembler	Scan object files	Ctrl+O
View resources	Modules, Memory	View all resources View resource strings	
Suspend/resume thread	Threads	Suspend Resume	
Display relative addresses	Disassembler, Dump, Stack	Doubleclick address	
Copy	Most of windows	Copy to clipboard	Ctrl+C

5.2.3.2 Frequently used global shortcuts

Ctrl+F2	Restart program
Alt+F2	Close program
F3	Open new program
F5	Maximize/restore active window
Alt+F5	Make OllyDbg topmost
F7	Step into (entering functions)
Ctrl+F7	Animate into (entering functions)
F8	Step over (executing function calls at once)
Ctrl+F8	Animate over (executing function calls at once)
F9	Run
Shift+F9	Pass exception to standard handler and run
Ctrl+F9	Execute till return
Alt+F9	Execute till user code
Ctrl+F11	Trace into
F12	Pause
Ctrl+F12	Trace over
Alt+B	Open Breakpoints window
Alt+C	Open CPU window
Alt+E	Open Modules window
Alt+L	Open Log window
Alt+M	Open Memory window
Alt+O	Open Options dialog
Ctrl+T	Set condition to pause Run trace
Alt+X	Close OllyDbg

5.2.3.3 Frequently used Disasembler shortcuts

F2	Toggle breakpoint
Shift+F2	Set conditional breakpoint
F4	Run to selection
Alt+F7	Go to previous reference
Alt+F8	Go to next reference
Ctrl+A	Analyse code
Ctrl+B	Start binary search
Ctrl+C	Copy selection to clipboard
Ctrl+E	Edit selection in binary format
Ctrl+F	Search for a command
Ctrl+G	Follow expression
Ctrl+J	Show list of jumps to selected line
Ctrl+K	View call tree
Ctrl+L	Repeat last search
Ctrl+N	Open list of labels (names)
Ctrl+O	Scan object files
Ctrl+R	Find references to selected command
Ctrl+S	Search for a sequence of commands
Asterisk (*)	Origin
Enter	Follow jump or call
Plus (+)	Go to next location/next run trace item
Minus (-)	Go to previous location/previous run trace item
Space ( )	Assemble
Colon (:)	Add label
Semicolon (;)	Add comment

5.3 DriverStudio 2.7

5.3.1 Official Statement[5]

DriverStudio is an integrated set of software tools for developing, debugging, tuning, testing and deploying Windows device drivers. DriverStudio brings kernel mode programming and debugging up to the same state-of-the-art as application programming.
The SoftICE Driver Suite is a suite of the core device driver tools that accelerate the development and debugging of Windows device drivers.
DriverStudio now has full support for Windows XP and SoftICE DriverSuite has remote debugging capabilities.

New in DriverStudio Version 2.7 and SoftICE Driver Suite Version 2.7

One of the most innovative new features in Driver Studio is its host/target architecture. This means that developers can debug, test and tune drivers located on remote (target) machines from their development (host) machine. The host controls the target machine, and is connected using a serial cable or a TCP/IP connection through a LAN, WAN or the Internet.
DriverStudio 2.7 also provides full support for Windows XP, addressing the demand for new device drivers required by rapidly changing technology. As part of the device driver development suite, the following products have been enhanced:

DriverWorks and DriverNetworks

- NDIS 5.1 Support
- Enhanced support for WDM filter driver development.
- Libraries now support C++ exception handling in the kernel.

SoftICE

- Displays the number of logical processors assigned to a physical processor on Hyper-Threading enabled platforms.
- Allows the user to display available Model Specific Registers (MSRs), dump a specified MSR or range of MSRs.
- Improved Windows XP support.

BoundsChecker Driver Edition

- Enhanced memory and resource tracking collects multiple memory leaks, providing detailed information on each leak.
- Allows the user to monitor if a driver starts a thread but does not terminate the thread before the driver unloads.

TrueTime Driver Edition

- Supports USB 1.0 and 1.1, NDIS 5.1 (including Miniports) and IEEE1394 (Firewire) protocols.
- Users can measure and monitor the performance of display drivers, print drivers.

Driver Workbench

- Reports multiple memory leaks with detailed information on each leak, as detected by a corresponding enhancement in the BoundsChecker Driver Edition. This helps the user to quickly locate memory leaks and other resource issues.

Licensing

- Both DriverStudio and SoftICE Driver Suite are licensed on a named user basis. Under the new host/target architecture, there are separate installation procedures for the host and target machine. Host installations require a license; target installations do not. One licensed developer can debug an unlimited number of target machines.

[5] Collected from http://www.numega.com

5.3.2 Inside SoftICE Driver Suite

Couple the best Windows device driver development tools with the hands-down best Windows kernel-level debugger, and you've got a combination that will see you safely through the gnarliest driver development projects the world might throw at you.

SoftICE Driver Suite is composed of the SoftICE debugger, accompanied by Compuware's core Windows device driver development tools: VtoolsD, DriverWorks and DriverNetworks. With SoftICE Driver Suite you can create and debug drivers for literally every Windows operating system under the sun: Windows 3.1, 9x, Millennium Edition, NT, 2000 and XP.

5.3.2.1 Components

- SoftICE
- VtoolsD
- DriverWorks
- DriverNetworks

5.3.2.2 SoftICE

The 800 Pound Gorilla of Debuggers

SoftICE is THE premier Windows debugger. An award-winning, kernel-mode debugger, SoftICE gives the device driver developer full, system-wide insight into the Windows environment. With SoftICE, you can track calls and events down to the lowest level of the operating system.

Because drivers run in kernel mode, it is difficult to access the insides of device drivers while they're executing. Traditional debugging tools run at the Windows application level, and can't observe and report actions and events occurring at the driver level. SoftICE is specifically designed to run between the operating system and the hardware, and can see interactions between drivers, calling routines and kernel services a capability that is necessarily absent from application-level debuggers.

SoftICE accommodates a variety debugging configurations. In a multi-machine arrangement, SoftICE can debug a target machine across a serial line, as well as an IP connection (ideal for those situations where you would prefer SoftICE not interfere with the target machine's display). Or, SoftICE can run on a single PC, eliminating the cumbersome requirement for a system to monitor the behavior of the target.

Solving problems with Windows device drivers can be challenging. Because ordinary debugging tools share Windows resources, they have limited access to the Windows kernel. When Windows crashes, they crash, too.

Crashing device drivers can cause "blue screens." Without SoftICE, figuring out the cause of a blue screen can take weeks. With SoftICE, you can peer into the OS kernel and examine instructions, events, calls, etc.

SoftICE has a text command-driven interface. The SoftICE screen is a resizable, movable window that displays a user-configurable selection of Windows internal information.

On-Demand Debugging

The SoftICE debug screen is text-based and doesn't use or rely on Windows APIs to display information, making it possible to get debugging information even if Windows is affected. With SoftICE, you control the entire system the entire time especially during a failure, when valuable system state information is still available. A user-definable hot key allows you to pop-up the SoftICE screen and "freeze" the OS. You can then explore code, stack, memory, registers, and whatever else you need to track down driver problems.

For years, SoftICE has been the last word in serious, system-level debugging. SoftICE provides driver developers with:

- access and trace of difficult driver problems right through to the hardware level
- blue screen-immune debugging
- multiple-process and thread debugging
- single-machine debugging
- multiple-machine debugging across a serial or network connection
- on-demand debugging of code running at any level in the operating system.

It's time to get serious! You can be a better device driver developer; you can spend less time searching for driver errors; you can deliver higher quality drivers. You can with SoftICE.

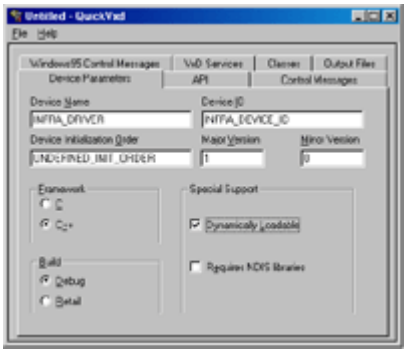
5.3.2.3 VtoolsD

Painless VxD Development

VtoolsD is a comprehensive platform for rapidly producing reliable VxDs. There's no need to maintain separate and older development tools or fill your files and bookshelves with extensive information on VxD development. With VtoolsD, you have the same high-quality development environment as those available to desktop application developers. VtoolsD matches desktop development tools in increased productivity, ease of use and quality of output.

VtoolsD helps you create VxDs for the Windows 3.x, Windows 9.x and Windows Me operating systems. The toolkit includes the QuickVxD wizard, supporting libraries, comprehensive help and documentation, as well as a number of example drivers (the examples are accompanied by complete source code). VtoolsD can be used in conjunction with DriverNetworks™ (see below) to construct TDI client drivers.

QuickVxD



QuickVxD Wizard

QuickVxD is a forms-based wizard that uses a fill-in-the-blank user interface for quickly and easily creating a VxD's source code skeleton. You enter the information that describes your VxD device driver, and QuickVxD does the rest. The entry parameters are grouped using a tabbed layout manager. Once you've entered all the information for your device driver, simply push a button and QuickVxD generates the following files:

- an include file, which contains definitions of symbolic constants
- a code module, which contains skeletons of the functions within the VxD that handle control messages, provide VxD services, and provide application entry points
- a make file, which shepherds the process of compiling the VxD. All the required compiler parameters are defined in the makefile.

The source code is copiously documented, and is infused with "TODO" comments that indicate the routines you flesh out with driver-specific code.

Libraries

The VtoolsD libraries are comprehensive, providing interfaces for every service offered by the Virtual Machine Manager and all of the standard Microsoft VxDs. Interfaces in the libraries support both C and C++. Also included is a broad subset of the ANSI C run time functions, rewritten for VxD development.

VtoolsD provides debug and retail versions of the libraries. The debug version includes diagnostic code that can detect various errors. The retail version is suitable for compiling into the final VxD. Best of all, the source code is included.

Help

VtoolsD is accompanied by extensive online help. Context sensitive help in the QuickVxD wizard makes it a snap to learn and use. Also provided is an exhaustive reference to the class libraries, as well as descriptions for the example VxDs. The online help includes numerous technical notes and tips that help you improve your VxD code, as well as avoid the traps and snares of VxD programming

5.3.2.4 DriverWorks

Better Drivers, Faster

DriverWorks automates much of the detail work in designing and building new device drivers for Windows-based systems. It provides an underlying driver architecture for interfacing hardware and other devices to the operating system. DriverWorks also incorporates difficult-to-find information about device and operating system characteristics, harnessing that information so that you can use it to quickly build efficient drivers.

DriverWorks supports the design and development of device drivers for Windows 95, Windows 98, Windows Millennium Edition, Windows NT, Windows 2000 and Windows XP. Finally, device driver developers have a development environment comparable to environments available to desktop application developers in productivity, ease of use and quality of output. DriverWorks is a central part of the DriverStudio suite.

DriverWorks is not a single application. It is a collection of cooperating components, orchestrated to work together to provide an environment that accelerates Windows device driver development. Those components include the DriverWizard, an extensive library of classes, documentation and examples.

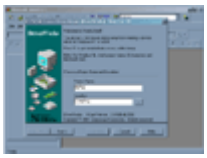
DriverWizard

The DriverWorks DriverWizard automates the process of creating a Windows device driver. The DriverWizard is a step-by-step guide that leads you from conception to reality. It produces ready-to-compile source code that serves as a scaffolding on which you can erect the details of your specific driver.

DriverWizard is integrated with Microsoft Visual Studio. Launch the DriverWizard from the Visual Studio toolbar, and the wizard prompts you through the creation of your device driver, beginning with your driver project's name and target directory.

Navigation buttons let you move forward and backward in the process, freely altering and adjusting the project's parameters.

Each step in the DriverWizard is accompanied by context-sensitive help, so there's no possibility of you getting lost.



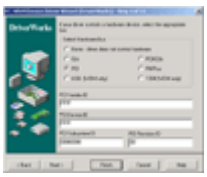
DriverWizard starts a project

The DriverWizard can generate driver source for Windows NT drivers and WDM drivers.

The DriverWizard "understands" the popular buses; USB, PCI, 1394, ISA and more. Select the bus your driver will interface with, and the DriverWizard adjusts its prompts accordingly.

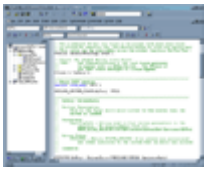
At any step in the process, you can select "FINISH" and source code will be generated. (Those parameters in unreached steps in the DriverWizard are set to defaults.)

When you're done, DriverWizard emits ready to compile source code, all bundled into a Visual Studio project. And not only is the source ready to compile, but the resulting executable is ready-to-run, too.



DriverWizard understands many driver types

DriverWizard creates a skeleton driver; now all you have to do is fill in the blanks. We've made that job easy, too. Liberally sprinkled throughout the source code, you'll find "TODO" placeholders, showing you exactly where your driver-specific code has to go, and telling you exactly what that code has to do.



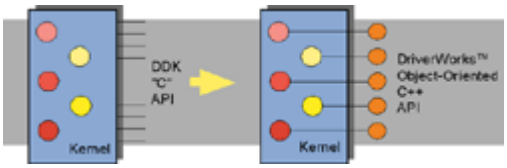
Profusely commented code

DriverWizard provides the precise kick start that your driver project needs. It removes most of the starting friction you face whenever you tackle creating a new device driver. Let DriverWizard handle the drudge-work of writing the source code for those common parts of the device driver that you would rather not write. You concentrate on the specifics of your design.

Library

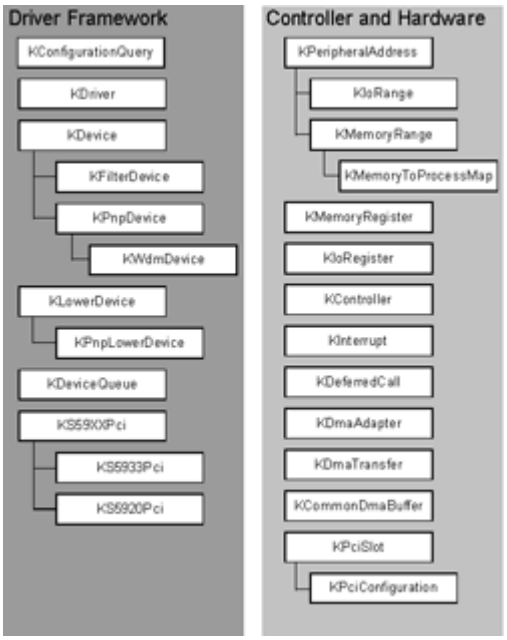
The structure of the Windows kernel is not a homogeneous soup of functions. Though not constructed along strict object-oriented lines, it is nevertheless composed of interoperating components.

Microsoft's DDK paints a thin, flat C-language API "veneer" over the Kernel's not-so-flat architecture. Albeit comprehensive, the API is nevertheless complicated. Being a flat, functional-language API, the DDK misses the opportunity to bring organization to the already well-structured landscape of kernel objects.



DDK compared to DriverWorks

The DriverWorks library takes a different approach. Rather than ignore and mask the kernel's abstractions, it exposes them. The underlying object structure isn't whitewashed over by a flat API, it is enhanced through a C++ API.



A portion of the DriverWorks class library

The DriverWorks library reduces the complexity of the maze of DDK functions. Naturally, code generated by the DriverWizard employs the classes and objects implemented within the DriverWorks library. The resulting code is more expressive than would be possible using merely the C API provided by the DDK. Source code generated using library calls also tends to be shorter than equivalent code using only the DDK, and most important more readable.

Documentation and Examples

Good documentation is key to the effective use of a tool as complex as DriverWorks. Good documentation must play several roles, and the authors of the DriverWorks documentation have worked hard to see that it fulfills them all:

1. It must provide an easy introduction to the toolset. That first rise in the learning curve is the steepest; the documentation has to work with you as you begin your climb. Consequently, DriverWorks arrives with a "Getting Started" manual. In addition, the DriverWizard offers the easiest leg up on the driver development process you'll find anywhere; each DriverWizard step is accompanied by context-sensitive help, and many of the help topics are conveniently linked to related topics in the larger DriverWorks online help system.
2. It must provide "how to" information. There's no use spending your time re-inventing the wheel. The DriverWorks documentation contains an extensive "How To" section. Common tasks and programming topics you're sure to face in your development cycle are described in detail. We've figured out how to do it so you don't have to. Also, the best way to learn something is by studying a pre-built example of it. DriverWorks is accompanied by an extensive collection of sample driver source code. You can tear the code apart to see how it's done, or modify it to fit your particular driver's requirements.
3. It must serve as an effective reference source. When you're deep in the bowels of device driver development, and you can't remember the name of that one function or method you need to take the next step, the last thing you want to spend your time doing is wandering the trackless wastes of confused documentation. You want to get your information, and get on with the job. We understand that, and we've built DriverWorks documentation with speed and efficiency in mind. You can search by keyword, or you can navigate via topics. Furthermore, the documentation is extensively inter-hyperlinked. While reading one topic, you can quickly jump to related information.

5.3.2.5 DriverNetworks

Instant Network Driver Expertise

Writing device drivers for network interface cards (NICs) using the Network Driver Interface Specification (NDIS) presents significant challenges. Not only are you faced with an evolving technology, but there is a lack of technically-nourishing documentation on the intricacies of NDIS. The work involved is particularly arduous for novice network driver developers, who must learn the details of a kernel-mode program at the same time they're trying to learn NDIS.

With the inclusion of DriverNetworks, DriverStudio offers a full development and debugging solution for NDIS and TDI (Transport Driver Interface) client network drivers. DriverNetworks provides an object-oriented framework that brings the same level of organization to network drivers that DriverWorks brings to Windows drivers in general.

DriverNetworks supports the design and development of device drivers for Windows 95/98, Windows Millennium, Windows NT, Windows 2000 and Windows XP — even 64-bit XP. At last, network driver builders have a development environment that rivals desktop environments in productivity, ease of use and quality of output.

With DriverNetworks, you can build a wide range of network drivers, including NDIS Miniport, NDIS Intermediate, NDIS Protocol , NDIS Transport, and TDI Client drivers.

Like DriverWorks, DriverNetworks is a collection of interoperating components: the Network Driver Wizard, an extensive class library, and documentation and example drivers.

Network Driver Wizard

DriverNetworks Network Driver Wizard carries on the tradition of excellent driver-development sorcery begun with VtoolsD QuickVxD and transmitted to DriverWorks Driver Wizard. The Network Driver Wizard can be integrated with Visual Studio, or run standalone. The Network Driver Wizard targets any Windows OS that supports NDIS; the Wizard can even generate 64-bit configurations, which can be built against the Windows XP DDK (IA64).

The Network Driver Wizard can generate all the source and project files you need to create your driver in Microsoft Visual Studio. Just like the DriverWorks Driver Wizard, the Network Driver Wizard walks you through the process step by step, prompting you at each point for information that defines the characteristics of your network device driver.

The Network Driver Wizard generates all the necessary source, .DSP and .INF files for you . It jump-starts an NDIS Miniport, Intermediate or Protocol driver (select which sort of driver, and the Wizard adjusts itself accordingly).

Network Driver Wizard

Libraries

The DriverNetworks class libraries bring object-based organization to network drivers. Classes in the libraries can be divided into the following categories:

- NDIS classes - includes classes that provide the frameworks for NDIS Miniport, NDIS Intermediate and NDIS Protocol drivers.
- Notify Object classes - COM server DLLs that are used by the Network class installer and Network control panel applications. The Notify Object framework provides wrapper classes around the COM interfaces to simplify writing a Notify Object.
- Transport Driver classes - These are a set of loosely-coupled model classes which can be used (and are used by the Network Driver Wizard) as part of a transport driver because of the complexity and variety of design choices for a transport driver, the Transport Driver classes model concepts, rather than provide base classes or templates.
- TDI Client (DriverSocket) classes - Implement DriverSockets, a C++ framework for kernel-mode drivers that require TCP/IP connectivity.
- Utility classes - Include classes for synchronized containers, allocation from an alternate heap, a class that wrappers the system-defined NDIS_STRING, and other useful classes.

Part of the DriverNetworks framework

The DriverNetworks class framework for NDIS drivers resides between your driver code and the system NDIS interface. Similarly, the DriverNetworks framework for TDI Clients (DriverSockets) resides between your driver code and the system TDI Interface. The frameworks delegate system callbacks into classes you derive from adapter-base classes supplied from DriverNetworks.

Documentation and Examples

DriverNetworks documentation and supporting collateral consists of extensive online documentation, as well as context-sensitive help within the Network Driver Wizard that links into the online documentation at appropriate points. DriverNetworks help files include plenty of "how to" topics as well as comprehensive reference information for the library classes.

Finally, DriverNetworks fully-functioning source-code examples include samples of NDIS Miniport, Filter and Intermediate drivers. In addition, sample DriverSockets drivers are included for Windows NT, WDM and Windows 9x.

Development Utilities

We've got even more weaponry for the battle-weary driver developer. Included with DriverStudio is a squad of command-line tools that target specific development problems.

- Monitor - Used to display debug messages and to start and stop VxDs and NT4-style drivers.
- SetDDKgo - Sets up the device driver build environment. Sets all those esoteric environment variables for the proper DDK build, so you don't have to worry about them.
- SrcToDsp - Used to convert drivers that are built from the BUILD.EXE tool (which uses a SOURCES files) to a .DSP that can be used in Visual Studio.
- DspToDsp - Used to convert older .DSP project files to be compatible with the XP DDK.
- ExDriverInstaller - Installs WDM drivers on Windows 2000 and XP. This tool lets you sidestep the device manager and the "Add New Hardware" wizard.
- SymLink - A simple but useful utility that displays a list of the symbolic links defined in the system.

DriverStudio includes other utilities not listed here. For example, there is a utility for adding user-defined BoundsChecker events (See BoundsChecker).

5.3.2.6 SoftICE Driver Suite

It's What You Need

SoftICE Driver Suite is a one-two development and debugging punch. It leads off with the best Windows driver construction tools in the business — VtoolsD, DriverWorks, and DriverNetworks — then follows through with the champion Windows kernel debugger, SoftICE. The combination is unbeatable; you'll be knocking out your device driver projects in record time.

5.3.3 Our Statement

SoftIce and Driverstudio is still the best tool for debugging. The price of about 1000$ is very high but you have to decide if this is a fair price.

Actually there are many discussions at several boards[6] about SoftIce specific topics. One of the most mentioned cases are the problems with several graficscards and Windoze XP. Most problems had been fixed since version 2.5 of the DriverStudio product, but still version 2.7 has many problems with XP or gfx hardware. Especially the GeForce gfx card seems to cause several problems.

Additionaly it seems that SoftIce have sometimes problems after pressing F12 or similar. This seems to be more a problem of the target OS than of the product itself.

For small targets it is still usefull to use OllyDbg instead of this “Gorilla of Debugger”. For advanced reversing techniques or difficult protectionschemes SoftIce still rules.

In combination with several other tools which can hide SoftIce activity this tool can nearly do everything.

[6] http://www.woodmann.com/upload

6 Crackme of the Issue - CrackMe tutorial for Cruehead's Crueme 1.0 - By Merkuur / CiA (Using Genetic Algorithms for retrieving serials)

6.1 Tutorial

Hi reversers,

In this tutorial I will explain the method I used to find valid codes for Cruehead's nice crackme. I think I was the 5th cracker to actually complete it. Cruehead released the source code for his crackme after that, which is just as well - I won't have to copy a lot of asm into this text :).

Put a breakpoint on hmemcpy and run the program, entering any code.
Sure enough, you break in. Step around, and you will eventually get to the place were the check on your code is performed, namely, at 00401B73. Single-step inside this code, and watch what happens... You will quickly see that the entered password is at ebx, and that the code performs operation on the data at ebx, ebx+4, and ebx+8. So the entered code must be 12 chars long. You should then analyze the code, understand what it does. I've summed it up for you below:

- **2 values are computed from the 3 ints at ebx, ebx+4 and ebx+8. These values are stored in edi and esi. I've converted the asm code performing the computation to C, from the 3 int values p1, p2, p3. If you understand asm better than C then you'll be at home with the disassembled listing.**

```
unsigned int eax, i, j;
unsigned int length=12; /* the code length */
esi=0; edi=0;           /* resulting values in edi, esi */
for (i=0;i<0xFF;i++) {
   for (j=0;j<0xFF;j++) {
      eax = (p1+p3+0x0012AB20)*length;
      eax = eax ^ (p2+0x048FF4EA);
        eax = eax ^ (p3-p1-0xBC309A);
        edi = edi + (eax | 0x029359E2);
        edi = edi & 0x15263748;

        eax = (p1-p3-0x127FB9)/length;
        eax = eax ^ (p2-0x048FF4EA);
        eax = eax ^ (p1+p3+ 0xBC0533);
        esi= esi+ (eax & 0x029359E2);
        esi = esi | 0x596A7B8C;
      }
      edi = edi + 0x911;
      esi = esi - 0x911;
   }
```

```
- the following code is executed:
     :00401C51 add edi, EFFDE3AF
     :00401C57 jne 00401C8E         ; bad code !
     :00401C59 add esi, A4948D23
     :00401C5F jne 00401C8E         ; bad code !
     :... good code !
```

so edi+EFFDE3AF and esi+A4948D23 must be 0, which means that we must have:
edi = 10021c51, esi = 5b6b72DD

Allright... Now the tough part is, what must the p1, p2, p3 values be if we want to satisfy the final check ? it is not practical to work back from edi = 10021c51 and esi = 5b6b72DD, first because there are 255x255 operations to revert, and, more importantly, because the operations involved are not reversible (edi = edi & 0x15263748 for instance...). Therefore... Bruteforce is the way to go...

Unfortunately the key we want to bruteforce is a 3-int thing... even when limited to printable chars, it's still something like a 84 bits key...
Trying to test all possible values will take ages !!! So what I did is, try a key (like ABCDEFGHIJKL), write down edi and esi. Change one bit in the key, and repeat the process... What I found is, changing a single bit in the key doesn't change the resulting edi and esi much. So the idea is, starting from a random key, try to change bits in it so that the resulting edi and esi values get "closer" (in terms of number of matching bits) to the desired values (10021c51 and 5b6b72DD). Actually the number of matching bits in the resulting edi and esi values can be thought as an "evaluation of the interest" of the initial key. The more matching bits, the closer you are from a valid key. Whenever you have found a "evaluation of interest" function, (also called a "survival criterion") you can use a GENETIC ALGORITHM in order to find a key that minimizes this function.

A genetic algorithm is an iterative process that mimicks the natural evolution process ("only the best ones survive"). For this crackme we would consider our 12 chars key to be a living creature, made of 12 genes. Starting from a random set of creatures, we compute the "survival criterion" of each. Then, we iterate the following:

- **"Natural selection" process: keep some of the best scoring creatures, and get rid of the others.**
- **"Reproduction" process: generate new creatures, by mixing the genes from the best ones. Hopefully this will result in new ones that will have an even better score.**
- **"Mutation" process: in order to allow the apparition of new interesting genes that were not present in the initial population, randomly modify some genes in a few creatures.**

Repeat this sequence for several generations of creatures, and eventually, there will be one that has the best possible selection criterion. In the case of the crackme, it is a key that generates the wanted values for edi/esi.

You will find a C program that performs all these operations in this package. On my P100 it finds valid codes in a few minutes. If someone ever writes an assembler version please send it to me, I would love to see it.

Happy reversing,

Merkuur

6.2 Source

```
/*  Genetic bruteforcer for crueme.exe
  0                                                                        NELEMS
  <--- SURVIVORS--->  <--- REPRODUCTION OF SURVIVORS ---> <--- MUTATORS --->

some solutions:
445340531008
752968361941
043907667011
DCIEDHNKGALJ
BZONJSGDHJZE
AQGMPDPNCQD
agstjdpiosau
qtocggrvelho

some working calls:
crue 10 5 5 <range> seems to work pretty well
crue 10 5 5 09
crue 10 10 10 09
crue 3 20 50 09
crue 10 10 10 AQ
crue 3 10 10 AZ
crue 3 20 30 AQ
crue 10 5 5 az

details:
crue 3 10 10 AZ
IZOZZPGDHUDE   score: 5, 23 attempts
BZOZJPGJHJZE   score: 3, 83 attempts
BZOZJCGDHJZE   score: 2, 183 attempts
BZOZJSGJHJZE   score: 1, 203 attempts
BZONJSGDHJZE   score: 0, 903 attempts

crue 3 20 50 09
922428202948   score: 5, 73 attempts
752964341041   score: 4, 213 attempts
752968341941   score: 1, 353 attempts
752968361941   score: 0, 563 attempts

*/
```

```
typedef struct {
  unsigned char v[12];
  unsigned int score;
} elt;


int NELTS, SURVIVORS, MUTATORS, BYTES_START, BYTES_RANGE;
static elt *SOUP;
static unsigned int attempts=0;
/*-------------------------------------------------------*/
/* this is the translation to C of the crypting function */
/*-------------------------------------------------------*/
static void GetCrypt(unsigned int p1, unsigned int p2, unsigned int p3, unsigned
int *esi, unsigned int *edi) {
  unsigned int eax, i, j;
  unsigned int length=12;
  attempts++;
  *esi=0; *edi=0;
  for (i=0;i<0xFF;i++) {
    for (j=0;j<0xFF;j++) {
       eax = (p1+p3+0x0012AB20)*length;
       eax = eax ^ (p2+0x048FF4EA);
       eax = eax ^ (p3-p1-0xBC309A);
       *edi = *edi + (eax | 0x029359E2);
       *edi = *edi & 0x15263748;

       eax = (p1-p3-0x127FB9)/length;
       eax = eax ^ (p2-0x048FF4EA);
       eax = eax ^ (p1+p3+ 0xBC0533);
       *esi= *esi+ (eax & 0x029359E2);
       *esi = *esi | 0x596A7B8C;
    }
    *edi = *edi + 0x911;
    *esi = *esi - 0x911;
  }
}
/*-------------------------------------------------------*/
/* count the number of non-zero bits in a 32 bits number */
/*-------------------------------------------------------*/
static int CountBits(unsigned int v) {
  int res=0;
  while (v!=0) {
    if (v&1 != 0) res++;
    v=v>>1;
  }
  return(res);
}
```

```
/*-------------------------------------*/
/* Compare 2 elements  (48 bits value) */
/*-------------------------------------*/
static int SameElt(elt *E1, elt*E2) {
  if (*((int*)(E1->v))==*((int*)(E2->v))) {
    if (*((int*)(4+E1->v))==*((int*)(4+E2->v))) {
      if (*((int*)(8+E1->v))==*((int*)(8+E2->v))) {
       return(1);
      }
    }
  }
  return(0);
}
/* is this element already in the list ? */
static int IsDupElt(elt *E1) {
  int i;
  elt *E2=SOUP;
  for (i=0;i<NELTS-1;i++) {
    if (E1 != E2) {
      if (SameElt(E1, E2)) {
       return(1);
      }
    }
    E2++;
  }
  return(0);
}
/*-------------------------------------------------------------------------*/
/* compute the score of a given element - it's the number of non-zero bits */
/* in the result of the crypting function - how 'far' in number of bits,  */
/* esi and edi are from the desired values                                 */
/*-------------------------------------------------------------------------*/
static int GetScore(elt *E) {
  unsigned int esi, edi;
  unsigned int p1, p2, p3;
  int res=0;
  p1=*((int*)(E->v));
  p2=*((int*)(4+E->v));
  p3=*((int*)(8+E->v));
  GetCrypt(p1, p2, p3, &esi, &edi);
  esi=esi^0x5b6b72DD; edi=edi^0x10021c51;
  res+=CountBits(esi);
  res+=CountBits(edi);
  return(res);
}
/*----------------------*/
/* print out an element */
/*----------------------*/
static void print_elt(elt *E) {
  int i;
  for(i=0;i<12;i++) {
    printf("%c", E->v[i]);
  }
  printf("   score: %d", E->score);
}
```

```
/*-------------------------------------------------------------------*/
/* init an element, with a random 12 bytes value                     */
/* allowed bytes start at BYTES_START, within BYTES_RANGE range. */
/*-------------------------------------------------------------------*/
static void init_elt(elt *E) {
  unsigned char i;
  do {
    for(i=0;i<12;i++) {
      E->v[i]=BYTES_START+(rand()%(BYTES_RANGE));
    }
  } while (IsDupElt(E)==1);
  E->score=GetScore(E);
}
static void init_elts() {
  int i;
  SOUP=(elt*)malloc(NELTS*sizeof(elt));
  for(i=0;i<NELTS;i++) {
    init_elt(&(SOUP[i]));
  }
}
/*-------------------------------------------------------------------*/
/* sort elements according to survival criterium (their 'score') */
/*-------------------------------------------------------------------*/
static int CmpFunc(const void  *_e1, const void  *_e2) {
  elt* e1=(elt*)_e1;
  elt* e2=(elt*)_e2;
  return(e1->score-e2->score);
}
static void SortSoup() {
  int i;
  qsort(SOUP, NELTS, sizeof(elt), CmpFunc);
}
```

```
/*-------------------------------------------------------------------------------
-------------*/
/* the actual reproduction process. Keep the best ones, reproduce them, and mutate
some of them */
/*-------------------------------------------------------------------------------
-------------*/
static void Mutate() {
  int i, j, idx1, idx2, bit, mask;
  int count;
  /* we keep only the SURVIVORS first ones for reproduction */
  /*printf("reproducing\n");*/
  for(i=SURVIVORS;i<NELTS-MUTATORS;i++) {
    count=0;
    do {
      count++;
      for(j=0;j<12;j++) {
       /* pick a random byte in a random survivor item */
       idx1=rand()%SURVIVORS;
       SOUP[i].v[j]=SOUP[idx1].v[j];
      }
   } while (IsDupElt(&(SOUP[i]))==1 && count!=SURVIVORS);
   if (count==SURVIVORS) {
     init_elt(&(SOUP[i]));
   } else {
     SOUP[i].score=GetScore(&(SOUP[i]));
   }
  }
  /*printf("mutating\n");*/
  /* last MUTATORS ones are mutations of the best ones - a single byte is randomly
changed */
  for(i=NELTS-MUTATORS;i<NELTS; i++) {
    count=0;
    SOUP[i]=SOUP[NELTS-i-1];
    do {
      count++;
      idx2=rand()%12;
      SOUP[i].v[idx2]=BYTES_START +(rand()%(BYTES_RANGE));
    } while (IsDupElt(&(SOUP[i]))==1 && count !=MUTATORS);
    if (count==MUTATORS) {
      init_elt(&(SOUP[i]));
    } else {
      SOUP[i].score=GetScore(&(SOUP[i]));
    }
  }
}
/* show we're still alive and computing :) */
static void PrintProgress() {
  static int prev_score=-1;
  if (prev_score!=SOUP[0].score) {
    prev_score=SOUP[0].score;
    print_elt(&(SOUP[0]));
    printf(", %d attempts\n", attempts);
  }
}


static void check(unsigned int p1, unsigned int p2, unsigned int p3) {
  unsigned int esi, edi;
  GetCrypt(p1, p2, p3, &esi, &edi);
  printf("%08X %08X %08X -> %08X | %08X\n", p1, p2, p3, edi, esi);
}
```

```
int main(int argc, char **argv) {
  int CHILDS;
  /* parse cmd line */
  if (argc!=5) {
     printf("crue.exe <survivors> <childs> <mutations> <c1cn>\n");
     printf("example: crue 10 5 5 09\n");
     exit();
  }
  SURVIVORS=atoi(argv[1]);
  CHILDS=atoi(argv[2]);
  MUTATORS=atoi(argv[3]);
  BYTES_START=argv[4][0];
  BYTES_RANGE=argv[4][1];
  if (BYTES_RANGE<BYTES_START) {
       BYTES_RANGE=BYTES_START;
       BYTES_START=argv[4][1];
  }
  BYTES_RANGE-=BYTES_START; BYTES_RANGE++;
  NELTS=SURVIVORS+CHILDS+MUTATORS;
  init_elts();
  do {
    SortSoup();
    PrintProgress();
    Mutate();
  } while (SOUP[0].score!=0);
  printf("\n");
}
```

7 Tool of the Issue – Cryptool[7] - Demonstration and Reference Program for Cryptography

7.1 Official Statement

What is CrypTool?

CrypTool is a freeware program with a graphical user interface which enables you to apply and analyse cryptographic mechanisms.

CrypTool contains an exhaustive online help, which can be understood without deep knowledge in cryptography.

CrypTool is completely available in English and German.

CrypTool has implemented almost all state-of-the-art crypto functions and allows you to learn about and use modern and classic cryptography.

The methods available include both classic methods (e.g. the Caesar encryption algorithm) and modern cryptosystems (for example, the RSA and AES algorithms, as well as algorithms based on elliptic curves).

Automatic analysis tools for obtaining the key, starting from a knowledge of the encrypted document and any additional information (the unencrypted document or language of the document), are provided for the classic encryption algorithms.

To support your own analysis of documents, CrypTool can display a histogram of the document, determine the statistics for any N-grams, and calculate entropy and autocorrelation.

CrypTool can be used to calculate hash values for a document and to generate or analyse random numbers.
CrypTool also provides facilities for compressing and decompressing files. This enables to analyse the effects of file compression in advance of actual encryption of a given document.

The RSA cryptosystem is covered comprehensively and implemented for different codings. The RSA key is generated from prime numbers created by the user. Key generation, encryption and decryption can be followed step-by-step.

In the development of CrypTool much care has been taken to ensure that context sensitive help is available at any point by pressing the F1 key (we assume that the user is able to use typical Windows applications). The extensive online help explains all basic cryptographic terms, contains a short list of references from the area of cryptography and provides a timetable with a historic overview.

The examples (tutorials) in the online help will make it easy for you to get up to speed.

---

[7] http://www.cryptool.com/

In the script (provided as a PDF file) you will find other, somewhat mathematically oriented information on cryptographic methods, on prime numbers and on elementary number theory.

Also included is the story The Dialogue of the Sisters by Dr. Carsten Elsner (translated by Klaus Esslinger). Within this fantasy story a variant of the RSA cryptosystem is used.

CrypTool was developed jointly by companies and universities to provide an adequate teaching aid for modern courses and as part of an end user awareness programme aimed at raising the IT security awareness of staff and helping them to gain a deeper understanding of the concept of security.

Another objective was to make clearer the cryptographic techniques employed in Deutsche Bank. Thus it is possible with CrypTool, as a reliable reference implementation of the various encryption algorithms, to test the encryption used in other programs. With this tool it is therefore possible to check whether claims about encryption are founded or not.

The hash functions, the symmetric and asymmetric encryption algorithms and the key management facilities are based on the Secude toolkit (version 5.4.15) developed by Secude Ltd.

The enclosed Secude library is restricted through the ticket file to a key length of 768 bits for the asymmetric algorithms and smart card support is disabled. If you have a Secude lib licence you will also be able to use longer key lengths for the asymmetric methods with the aid of your ticket file.

In order to demonstrate the RSA cryptosystem and to factorise integers CrypTool uses the long integer arithmetic of the Miracl library (version 4.4.3) from Shamus Software Ltd. We restricted the integer bit length up to 1024.

Please note the terms of licence and the information regarding liability for any damage arising from the use of CrypTool.

Release of the source code

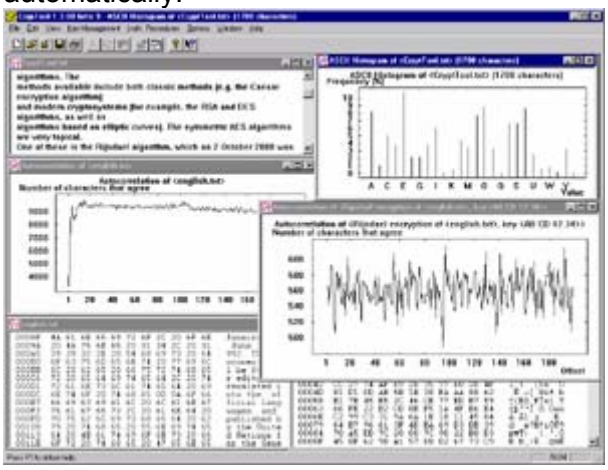
The sponsors of this product, Deutsche Bank AG, Secude Ltd., FZI Karlsruhe and the Universities of Darmstadt, Siegen and Karlsruhe will make the source code of the CrypTool application (but not of the Secude lib) available free of charge for training purposes with GNU-like (www.gnu.org) conditions.

Starting with release 1.4.00, the departement "IT Security" at TU Darmstadt, lead by Prof. Dr. Claudia Eckert, will take over the maintenance of CrypTool and coordinate its further development as Open Source Software. The new maintainer commits to make sure that

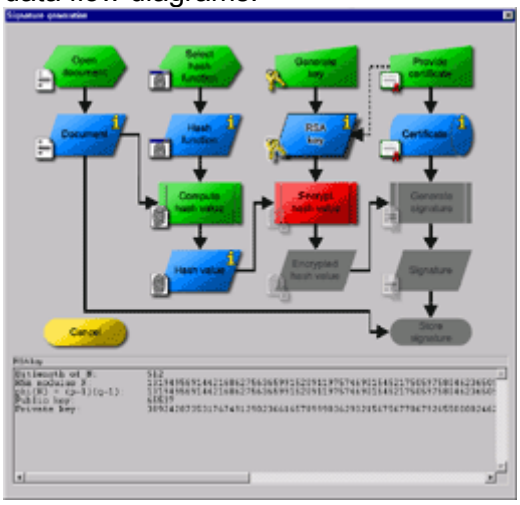
- the source code remains consistent,
- competent contact persons are available to support other developers,
- the names of authors and sponsors remain visible during run time of CrypTool and in the source code in a way comparable to release 1.3.03 and that
- all upcoming versions are maintained bi-lingually in English and German.

7.2 Screenshots

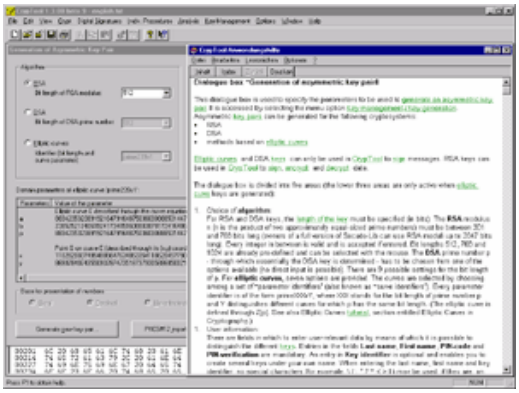
A number of different text analysis procedures are available in CrypTool. These will bring out the weaknesses of simple encryption algorithms, while some of the algorithms can be broken automatically.



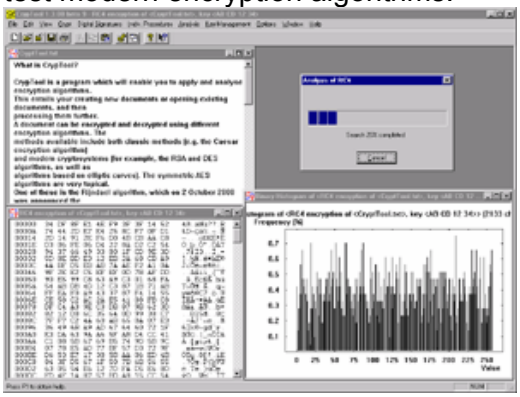
CrypTool reveals the inner working of digital signature and hybrid encryption with interactive data flow diagrams.



The capabilities of CrypTool are actively supported through the comprehensive Help facilities.



Thanks to the use of the Secude library, CrypTool offers the possibility of getting to know and test modern encryption algorithms.



8 Stupidity of the Issue – by ManKind

Well, for sure this stupity is not made by ManKind – he just provided us this example of stupied wannabies protectionists…

This author has done wrong a few things:

1) showing the directory and file name of the keyfile
2) it shows what the last three lines should contain, thus giving hints to crackers

I haven't tried to build a correct keyfile myself, probably the contents of the keyfile before the last 3 lines is also significant, so the author don't worry about giving us that hint.

However this is really something he shouldn't do.

Any decent buyer wouldn't have problem registering the app with the keyfile and they don't need to edit the keyfile and compare the last 3 lines.

This, even many crackme coders have already known and therefore shouldn't be repeated anymore.

ManKind

9 Essay of the Issue – by crUsAdEr

AsProtect - A reverse engineering approach

This tutorial aims to discuss more about internal working of AsProtect mainly, more than just unpacking it. So if you just want to unpack it and don't want to waste your time on reverse engineering, forget the second part of this tutorial!

TOOLS used :
IDA 4.15
Soft Ice on Win2k
LordPE
Revirgin (for unpacking only)
WinHex (for unpacking only)

Targets : ReGet Deluxe 3.0 beta (build 117) (but I think any program protected with the same version of AsProtect will do)
Included : asprotect.dll

1. Unpacking AsProtect
This is boring after a while, here I summarise some steps you should take when unpacking AsProtected program
(please read Spl/j's tutorial on commview, not much has changed since!)

1. - Run the program
2. - Run WinHex and open the program memory space, search for AsProtect signature byte "61 FF E0"
3. - If you are on win 98, run super bpm. (for win2k user, use Solomon's trick of "bpx 80464C50" to prevent AsProtect from clearing your bpm).
4. - Close the program, put a bpx on GetVersion and run the program again, sice should break, press F12 and you should be in the AsProtected code, trace with F8 and take note of where the results of this GetVersion is stored in memory. AsProtect store this results and used it later to emulate the API. Other emulated API are GetModuleHandleA, GetCurrentProcess, GetCurrentProcessId and GetCommandLineA. So trace and watch where they are stored. Write down these memory addresses. *Tips : trace with F8*
5. - Do a bpm on the address of "61 FF E0" you found earlier
6. - Let the program runs and sice will break again at "popad ; jmp eax " where eax is you OEP. Dump here.
7. - Run revirgin and let it resolve IAT (read the revirgin manual on how to do this)
8. - There are a few missing APIs which are either emulated (remember the address we wrote down earlier?) or redirected. You need to fix these manually and they should be dealt with case by case. *(again, read Spl/j's tutorial)*
9. - Sometimes AsProtect dips inside the program code before hitting OEP to trick /tracex and set various flags, encrypt/decrypt some codes but these should be dealt with individually.

Phew, done with the boring part, now if you are interested in how AsProtect really works and willing to work on your own to learn the art of reverse engineering then read on. The next few sections aims to discuss how AsProtect decrypts the program, how APIs are really emulated, how API is mangled, how dippings are done etc... and of course how your bpm are cleared!

2. AsProtect library

Yep, you are right! AsProtect has a dll that is used to perform all of its tasks of decrypting and loading the target. This dll is decrypted at runtime so this section will discuss how to obtain this dll from memory, rebuild it and use it to study AsProtect. Use IDA to disassemble the program, you should be able to over come most of the obfuscation code (grab IDA tutorials or read the IDA manual). My approach is disassemble and debug bit by bit in parallel.

Trace with softice from the beginning of the protected program, you will soon be brought AsProtect code in the last section of the program, F8 a few step and then you will be in the first decryption loop :

```
0067B083                    ; ---------------------------------------------------
----------
0067B083
0067B083                    loc_67B083:                         ; CODE XREF: 0067B06F□p
0067B083 8A DD                 mov     bl, ch
0067B085 5E                    pop     esi                      ; esi := 67B074
0067B086 8A C3                 mov     al, bl
0067B088 81 C6 C2 07 00 00     add     esi, 7C2h                ; esi := 67B836
0067B08E 56                    push    esi
0067B08F 8A E2                 mov     ah, dl
0067B091 5B                    pop     ebx                      ; ebx := 67B836
0067B092 68 B5 01 00 00        push    1B5h
0067B097 59                    pop     ecx                      ; ecx := 1B5
0067B097                                                        ; Number of time loop
perform
0067B097                                                        ; or number of dwords to
be decrypted
0067B098
0067B098                    loop1:                              ; CODE XREF: 0067B154□j
0067B098 FF 36                 push    dword ptr [esi]
0067B09A 66 B8 8B 7F           mov     ax, 7F8Bh
0067B09E 5A                    pop     edx                      ; edx := [esi]
0067B09F 0F 89 0C 00 00 00     jns     loc_67B0B1
0067B09F                    ; ---------------------------------------------------
----------
0067B0A5 0F                    db  0Fh ;
<----useless code---->
0067B0B0 00                    db    0 ;
0067B0B1                    ; ---------------------------------------------------
----------
0067B0B1
0067B0B1                    loc_67B0B1:                         ; CODE XREF: 0067B09F□j
0067B0B1 81 C2 51 CF 9C 42     add     edx, 429CCF51h           ; add edx
0067B0B7 66 BF 0A 05           mov     di, 50Ah
0067B0BB 81 F2 B6 23 95 0D     xor     edx, 0D9523B6h           ; xor edx
0067B0C1 80 CB 2D              or      bl, 2Dh
0067B0C4 81 EA B7 D8 25 0E     sub     edx, 0E25D8B7h           ; sub edx
0067B0CA 68 B0 D5 A1 60        push    60A1D5B0h
0067B0CF E8 14 00 00 00        call    sub_67B0E8
0067B0CF                    ; ---------------------------------------------------
----------
0067B0D4 DC                    db 0DCh ; _
<----useless code---->
0067B0E7 5B                    db  5Bh ; [
0067B0E8
0067B0E8                    ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
```

```
0067B0E8
0067B0E8
0067B0E8                    sub_67B0E8 proc near         ; CODE XREF: 0067B0CF□p
0067B0E8 66 BF A4 D9           mov     di, 0D9A4h
0067B0EC 58                    pop     eax              ; eax := 67B0D4
0067B0ED 5F                    pop     edi
0067B0EE 89 16                 mov     [esi], edx       ; store back edx into [esi]
0067B0F0 80 F7 27              xor     bh, 27h
0067B0F3 81 EE 7D 99 CB 4F     sub     esi, 4FCB997Dh
0067B0F9 66 8B D9              mov     bx, cx
0067B0FC 81 C6 79 99 CB 4F     add     esi, 4FCB9979h     ; sub esi, 4
0067B102 66 B8 35 7B           mov     ax, 7B35h
0067B106 49                    dec     ecx                ; decrease counter
0067B107 0F 85 22 00 00 00     jnz     continue_decrypt
0067B10D 0F 8A 06 00 00 00     jp      loc_67B119
0067B10D                    ; ---------------------------------------------------------
--------
0067B113 81                    db  81h ; ü
<----useless code---->
0067B118 74                    db  74h ; t
0067B119                    ; ---------------------------------------------------------
--------
0067B119
0067B119                    loc_67B119:                  ; CODE XREF:
sub_67B0E8+25□j
0067B119 E9 48 00 00 00                                  ; 0067B190□j
0067B119                       jmp     loc_67B166
0067B119                    ; ---------------------------------------------------------
--------
0067B11E A5                    db 0A5h ; Ñ
<----useless code---->
0067B12E 15                    db  15h ;
0067B12F                    ; ---------------------------------------------------------
--------
0067B12F
0067B12F                    continue_decrypt:            ; CODE XREF:
sub_67B0E8+1F□j
0067B12F E8 0D 00 00 00        call    loc_67B141
0067B12F                    ; ---------------------------------------------------------
--------
0067B134 91                    db  91h ; æ
<----useless code---->
0067B140 85                    db  85h ; à
0067B140                    sub_67B0E8 endp
0067B140
0067B141                    ; ---------------------------------------------------------
--------
0067B141
0067B141                    loc_67B141:                  ; CODE XREF:
sub_67B0E8+47□p
0067B141 E9 0D 00 00 00        jmp     loc_67B153
0067B141                    ; ---------------------------------------------------------
--------
0067B146 01                    db    1 ;
<----useless code---->
0067B152 F5                    db 0F5h ; )
0067B153                    ; ---------------------------------------------------------
--------
0067B153
0067B153                    loc_67B153:                  ; CODE XREF: 0067B141□j
0067B153 5F                    pop     edi              ; edi := 67D134
0067B154 E9 3F FF FF FF        jmp     loop1
```

Lots of obfuscation code but I list here once so that hopefully you will get accustomed to them. There are more to come!

I hope the dead listing with comments are good enough to understand, but as you can see AsProtect is decrypting something and that some thing happen to be the VERY next block of codes. Do trace through it once or twice and you will get the hang of it. This is important as it will help you to understand AsProtect structure better as you go on. As you can see, the key is to use IDA to disassemble at the right place and ignore obfuscation code. Also, just a personal opinion, comment like mad, I commented everything I see eventhouugh sometimes I don't know what they are, I simply rename those offset "some_shit", the next time you see "some_shit" you'll know that this variable has been accessed before and it helps....

Once you have understand how this loop works, bpx on the exit of the loop and you will soon see the next loop with the same algorithm but different key and size used to decrypt the block after itself. This decryption is repeated a few times, (I think 4) and then a block of data is copy to high memory and decrypt there (our dll). It is also quite interesting to watch how AsProtect search for its import; namely GetProcAddress, GetModuleHandleA, LoadLibraryA, VirtualAlloc and VirtualFree by scanning the export directory of kernel32.dll instead of using the pre-loaded import IAT.

Once, the dll is loaded into some high memory, I made a dump, attach it to the end of the program, adjust the sections header so that the virtual address is the same. At the first glance, it looks like just a data block with some code on it but once I start tracing this code I see something fishy. Here comes the OS loader :
(I remove the relocation codes as they are too long to list here. Only Import loading and OEP calculation is listed)

```
00A4A488                    loc_A4A488:
00A4A488                       mov     esi, dword ptr ss:unk_442A61[ebp] ; esi =
[A4A11D]
00A4A48E 8B 95 D8 30 44 00     mov     edx, dword ptr ss:unk_4430D8[ebp] ; add image
base
00A4A494 03 F2                 add     esi, edx                 ; esi now points to
Import Directory
00A4A496
00A4A496                    load_next_dl_import:
00A4A496 8B 46 0C              mov     eax, [esi+0Ch]           ; get dll name offset
00A4A499 85 C0                 test    eax, eax
00A4A49B 0F 84 0A 01 00 00     jz      finish_import_loading    ; eax := [A4A121]
00A4A4A1 03 C2                 add     eax, edx                 ; add image base
00A4A4A3 8B D8                 mov     ebx, eax
00A4A4A5 50                    push    eax
00A4A4A6 FF 95 EC 31 44 00     call    dword ptr ss:unk_4431EC[ebp] ;
GetModuleHandleA
00A4A4AC 85 C0                 test    eax, eax
00A4A4AE 75 07                 jnz     short library_loaded
00A4A4B0 53                    push    ebx
00A4A4B1 FF 95 F0 31 44 00     call    dword ptr ss:unk_4431F0[ebp] ; LoadLibraryA
00A4A4B7
00A4A4B7                    library_loaded:
00A4A4B7 89 85 4D 29 44 00     mov     dword ptr ss:unk_44294D[ebp], eax
00A4A4BD C7 85 51 29 44 00+    mov     dword ptr ss:unk_442951[ebp], 0 ; initialise
Import Counter
00A4A4C7
00A4A4C7                    next_first_thunk_entry:
00A4A4C7 8B 95 D8 30 44 00     mov     edx, dword ptr ss:unk_4430D8[ebp] ; get image
base
00A4A4CD 8B 06                 mov     eax, [esi]               ; check
Original_First_Thunk
00A4A4CF 85 C0                 test    eax, eax
00A4A4D1 75 03                 jnz     short original_first_thunk_found ;
00A4A4D3 8B 46 10              mov     eax, [esi+10h]           ; get first thunk
offset
00A4A4D6
00A4A4D6                    original_first_thunk_found:             ; CODE XREF:
00A4A4D1□j
```

```
00A4A4D6 03 C2                add     eax, edx                ;  add image base
00A4A4D8 03 85 51 29 44 00    add     eax, dword ptr ss:unk_442951[ebp] ; add
counter
00A4A4DE 8B 18                mov     ebx, [eax]              ; get Import ASCII
00A4A4E0 8B 7E 10             mov     edi, [esi+10h]
00A4A4E3 03 FA                add     edi, edx                ; edi => first thunk
00A4A4E5 03 BD 51 29 44 00    add     edi, dword ptr ss:unk_442951[ebp] ; add
counter
00A4A4EB 85 DB                test    ebx, ebx
00A4A4ED 0F 84 A2 00 00 00    jz      dll_done
00A4A4F3 F7 C3 00 00 00 80    test    ebx, 80000000h          ; import by ordinal?
00A4A4F9 75 04                jnz     short loc_A4A4FF
00A4A4FB 03 DA                add     ebx, edx                ; add image base
00A4A4FD 43                   inc     ebx
00A4A4FE 43                   inc     ebx                     ; add 2 to point to
API ASCII
00A4A4FF
00A4A4FF                   loc_A4A4FF:                            ; CODE XREF:
00A4A4F9□j
00A4A4FF 53                   push    ebx
00A4A500 81 E3 FF FF FF 7F    and     ebx, 7FFFFFFFh
00A4A506 53                   push    ebx
00A4A507 FF B5 4D 29 44 00    push    dword ptr ss:unk_44294D[ebp] ; modulehandle
00A4A50D FF 95 E8 31 44 00    call    dword ptr ss:unk_4431E8[ebp] ; get Proc
address
00A4A513 85 C0                test    eax, eax
00A4A515 5B                   pop     ebx
00A4A516 75 6F                jnz     short API_add_found
00A4A518 F7 C3 00 00 00 80    test    ebx, 80000000h          ; import by ordinal ?
00A4A51E 75 19                jnz     short loc_A4A539
00A4A520 57                   push    edi
00A4A521 8B 46 0C             mov     eax, [esi+0Ch]
00A4A524 03 85 D8 30 44 00    add     eax, dword ptr ss:unk_4430D8[ebp]
00A4A52A 50                   push    eax
00A4A52B 53                   push    ebx
00A4A52C 8D 85 53 31 44 00    lea     eax, unk_443153[ebp]
00A4A532 50                   push    eax
00A4A533 57                   push    edi
00A4A534 E9 99 00 00 00       jmp     loc_A4A5D2
00A4A539                   ; ---------------------------------------------------
------
00A4A539
00A4A539                   loc_A4A539:                            ; CODE XREF:
00A4A51E□j
00A4A539 81 E3 FF FF FF 7F    and     ebx, 7FFFFFFFh
00A4A53F 8B 85 DC 30 44 00    mov     eax, dword ptr ss:unk_4430DC[ebp]
00A4A545 39 85 4D 29 44 00    cmp     dword ptr ss:unk_44294D[ebp], eax
00A4A54B 75 24                jnz     short loc_A4A571
00A4A54D 57                   push    edi
00A4A54E 8B D3                mov     edx, ebx
00A4A550 4A                   dec     edx
00A4A551 C1 E2 02             shl     edx, 2
00A4A554 8B 9D 4D 29 44 00    mov     ebx, dword ptr ss:unk_44294D[ebp]
00A4A55A 8B 7B 3C             mov     edi, [ebx+3Ch]
00A4A55D 8B 7C 3B 78          mov     edi, [ebx+edi+78h]
00A4A561 03 5C 3B 1C          add     ebx, [ebx+edi+1Ch]
00A4A565 8B 04 13             mov     eax, [ebx+edx]
00A4A568 03 85 4D 29 44 00    add     eax, dword ptr ss:unk_44294D[ebp]
00A4A56E 5F                   pop     edi
00A4A56F EB 16                jmp     short API_add_found
00A4A571                   ;---------------------------------------------------
-------
00A4A571
00A4A571                   loc_A4A571:                            ; CODE XREF:
00A4A54B□j
00A4A571 57                   push    edi
00A4A572 8B 46 0C             mov     eax, [esi+0Ch]
00A4A575 03 85 D8 30 44 00    add     eax, dword ptr ss:unk_4430D8[ebp]
00A4A57B 50                   push    eax
00A4A57C 53                   push    ebx
```

```
00A4A57D 8D 85 A4 31 44 00    lea     eax, unk_4431A4[ebp]
00A4A583 50                   push    eax
00A4A584 57                   push    edi
00A4A585 EB 4B                jmp     short loc_A4A5D2
00A4A587                    ; ---------------------------------------------------------
-----------
00A4A587
00A4A587                    API_add_found:                          ; CODE XREF:
00A4A516□j
00A4A587 89 07                                                      ; 00A4A56F□j
00A4A587                       mov     [edi], eax               ; update first thunk
00A4A589 83 85 51 29 44 00+    add     dword ptr ss:unk_442951[ebp], 4
00A4A590 E9 32 FF FF FF        jmp     next_first_thunk_entry   ; get image base
00A4A595                    ; ---------------------------------------------------------
------------
00A4A595
00A4A595                    dll_done:                               ; CODE XREF:
00A4A4ED□j
00A4A595 89 06                 mov     [esi], eax               ; clear Import
Directory Entry
00A4A597 89 46 0C              mov     [esi+0Ch], eax
00A4A59A 89 46 10              mov     [esi+10h], eax
00A4A59D 83 C6 14              add     esi, 14h                 ; next Import
Directories Entry
00A4A5A0 8B 95 D8 30 44 00     mov     edx, dword ptr ss:unk_4430D8[ebp]
00A4A5A6 E9 EB FE FF FF        jmp     load_next_dl_import      ; get dll name offset
00A4A5AB                    ; ---------------------------------------------------------
------------
00A4A5AB
00A4A5AB                    finish_import_loading:                  ; CODE XREF:
00A4A49B□j
00A4A5AB 8B 85 65 2A 44 00     mov     eax, dword ptr ss:unk_442A65[ebp] ; eax :=
[A4A121]
00A4A5B1 50                    push    eax
00A4A5B2 03 85 D8 30 44 00     add     eax, dword ptr ss:unk_4430D8[ebp] ; add image
base
00A4A5B8 5B                    pop     ebx
00A4A5B9 0B DB                 or      ebx, ebx
00A4A5BB 89 85 11 2F 44 00     mov     dword ptr ss:unk_442F11[ebp], eax ; update
instruction at A4A5CC
00A4A5C1 61                    popa
00A4A5C2 75 08                 jnz     short OEP_OK
00A4A5C4 B8 01 00 00 00        mov     eax, 1
00A4A5C9 C2 0C 00              retn    0Ch
00A4A5CC                    ; ---------------------------------------------------------
------------
00A4A5CC                    OEP_OK:                                 ; CODE XREF:
00A4A5C2□j
00A4A5CC 68 00 00 00 00        push    0                        ; this will be
changed to push OEP
00A4A5D1 C3                    retn                             ; go to OEP of the
Delphi prog
00A4A5D2                    ; ---------------------------------------------------------
-------------
```

I know it is long dead listing, but do read through it, at least the comments as I think I wrote sufficient comments for readers to have an idea on what is happening. I have removed most of the codes (the decompression and relocation codes). Basically, it decompresses the main block and then perform what seems to me a relocation of image file, then import loading and finally getting OEP to jump to it. Hence I suspect this was either a dll or exe file (packed with AsPack?) or something. Hence I decided to dump this image (before all this reloc and IAT loading is done), it was hard to build the file as the header is completely ripped off and I took some time before getting it right so that IDA will disassemble them. First I notice that it has 6 sections (most if not all Delphi app has 6 sections) and then I remember AsProtect likes Delphi very much! Thus I open some Delphi exe and ripped the header completely, paste into this dump, fix the section info, fix the reloc and import info. (If you are not familiar with PE header, read some excellent PE tutorials by Iczellion and others at krobar's site).

Finally, I got IDA to disassemble the file nicely and hence discover it was a console dll written in Delphi and IDA (great tool!! Really helpful here) actually can apply its FLIRT feature to detect lots of Delphi function and that saves me lots of time. I must say if IDA has not been able to disassemble the dll properly, this tutorial would not have been done so easily. here I attach the dll here so you can disassemble and study them yourself. You should also try to dump and rebuild on from any AsProtect program as a practice.

Yep, once you obtain the dll, you will realise that all of AsProtect mysteries are inside that dll... its seh clearing debug registers, its crc check, IAT mangling etc... Here come the fun!!!

3. AsProtect.dll seh tricks

Here is how a typical seh is set up in AsProtect and you will know that seh is used not less than 30 times in this dll. It used to stop newbies like me from tracing AsProtect code but not anymore once you understand what is going on. Before you read this, please read some essential information about seh (I suggest Jeremy Gordon's excellent paper on seh).

```
004106B2 E8 49 C6 FF FF      call    Clear_API_emu_code      ; upper limit
004106B7 E8 25 00 00 00      call    set_seh_3
004106BC
004106BC                  seh_3_handler:                          ;
004106BC 8B 44 24 0C         mov     eax, [esp+0Ch]          ; get context to eax
004106C0 83 80 B8 00 00 00+  add     dword ptr [eax+0B8h], 2 ; add context.eip by
2
004106C7 51                  push    ecx
004106C8 31 C9               xor     ecx, ecx
004106CA 89 48 04            mov     [eax+4], ecx            ; clear debug
register 0
004106CD 89 48 08            mov     [eax+8], ecx            ; clear debug
register 1
004106D0 89 48 0C            mov     [eax+0Ch], ecx          ; clear debug
register 2
004106D3 89 48 10            mov     [eax+10h], ecx          ; clear debug
register 3
004106D6 C7 40 18 55 01 00+  mov     dword ptr [eax+18h], 155h ; context.dr7 :=
155
004106DD 59                  pop     ecx
004106DE 31 C0               xor     eax, eax                ; exception handled,
continue
004106E0 C3                  retn
004106E0                  set_seh_1 endp ; sp =  4
004106E1
004106E1                  ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
004106E1
004106E1                  set_seh_3 proc near                     ; CODE XREF:
set_seh_1+25□p
004106E1 31 C0               xor     eax, eax
004106E3 64 FF 30            push    dword ptr fs:[eax]      ; set up seh 3
004106E6 64 89 20            mov     fs:[eax], esp
004106E9 31 00               xor     [eax], eax              ; cause seh 3
004106E9                                                     ; this seh clear
debug register
004106EB 64 8F 05 00 00 00+  pop     large dword ptr fs:0    ; remove seh struc
004106F2 58                  pop     eax
004106F3 E8 CC 1F FF FF      call    @System@Randomize$qqrv  ;
System::Randomize(void)
```

The seh is set up in a slightly different way which makes it harder to detect but with IDA everything becomes very clear!

```
004106B7            call    set_seh_3             ; this is equivalent to push
handler and move eip to 4106E1 same as

                                                                              ;
push 4106BC     (our seh handler)

004106E1                xor     eax, eax
004106E3               push    dword ptr fs:[eax]          ; set up seh 3
004106E6                mov     fs:[eax], esp
004106E9                xor     [eax], eax                          ; cause seh 3
004106E9                                                                    ;
this seh clear debug register
004106EB               pop     large dword ptr fs:0    ; remove seh struc
004106F2               pop     eax
```

Thus when we trace over 4106E9, the seh is triggered and our context is retrieved to eax, where eip is adjusted by increasing it by 2 to point to the next working instruction, also debug registers are cleared the same way. (this is posted by R!sc before)

This is interesting indeed! For example when you are tracing and you are at 4106B7, if you trace into with F8 you will meet the faulty instruction at 4106E9 and be lost in kernel seh code! if you trace over with F10, sice will place a break point at the next instruction which happen to be the seh handler and then you will soon meet the "ret" at 4106E0 and again you will be lost in seh kernel code! Now that we know how this whole seh scheme works, we can trace anywhere we want, simply do a "r eip eip+2" at the faulty instruction and seh handler will be skipped altogether!

Analysing the dll is slightly harder as it is a long, full-blown dll with a hell lot of seh and Delphi bloated codes but IDA eases the jobs quite a bit. You should try to analyse this slowly, always trace over a call and guess what it does first before actually stepping inside it.

4. AsProtect.dll internal

OK, I am not going to discuss the whole dll here, that would take too long and is pointless to paste long Delphi codes. I will just roughly describe its structure. Here is the skeleton of the dll

```
00410C32 68 4C E7 40 00   push    offset self_hash
00410C37 68 5C 0D 41 00   push    offset Dips_DriveHash_Date_Registry_CodeDecrypt
00410C3C 68 90 02 41 00   push    offset nothing_important ; notthing really
important here
00410C3C                                                  ; just some file header
checking
00410C41 68 44 FF 40 00   push    offset complete_API_mangling
00410C46 68 1C F9 40 00   push    offset Fully_decrypt_file ; decrypt the file
fully,
00410C46                                                   ; IAT is untouched..
virgin!!!!!
00410C46                                                   ; import ASCII stripped
but
00410C46                                                   ; hint are still there
00410C4B 68 10 F3 40 00   push    offset Decrypt_file_first_time
00410C50 68 2C 06 41 00   push    offset emu_API_n_file_hash
00410C55 C3               retn
```

Yep, so AsProtect will execute the API_emulation routine and then on return, it will jump to the next routine Decrypt_file_first_time and then so on and so on. API emulation is pretty easy to understand, especially if you have unpack a few AsProtected programs. Also AsProtect use open the exe and hash the whole file to check for CRC. Also, you should read the thread by me, Mike and Dakien at Fravia's Board (crypto forum, "stream cipher??" and "Tutorial: finding encryption code") on encryption routines used. You will find that in fact all the main decryption routines are similar to that one throughout this dll with MD5 heavily used!

The third routine decrypt the exe fully and perform relocation on the code section if required. Here you will see how the mysterious "ret" at 401014 call (that appears in all AsProtect programs) is made, I guess to test if the code section is executable.

It should be noted that after the 3rd routine, we can actually dump the file and obtain a nice clean dump with IAT first thunk intact, import ASCII are stripped together with dll names but the hints are still there so we can actually rebuild the program from there!

Disassembling the API mangling routine we'll see exactly how decrypt import information, classify imports into different categories (6 if I am not wrong) and have different treatment for each of them. Again, you have unpacked AsProtect then you will know these routines or at least have an ideas of how Imports are treated differently by AsProtect. I will not post the full code here as the offset given above should be enough for you to find where the routines are.

```
00410033 A1 A4 49 41 00     mov     eax, ds:Encrypted_data_offset
00410038 8B 55 F0           mov     edx, [ebp-10h]
0041003B 89 42 04           mov     [edx+4], eax
0041003E 8B 15 94 32 41 00  mov     edx, ds:sign_13_import_data
00410044 8B 45 F0           mov     eax, [ebp-10h]
00410047 E8 A4 D0 FF FF     call    Decrypt_n_Find          ; eax points to
mem_struc
00410047                                                    ; mem_struc+4 : original
data
00410047                                                    ; mem_struc+8 :
decrypted data
00410047                                                    ; it also traverse the
decrypted data
00410047                                                    ; block to search for
data block with
00410047         ; signature in edx
0041004C 8B D8              mov     ebx, eax
0041004E 85 DB              test    ebx, ebx
00410050 74 1B              jz      short import_data_not_found
00410052 8B 43 04           mov     eax, [ebx+4]
00410055 E8 3A 25 FF FF     call    @System@@GetMem$qqrv    ; size in eax
0041005A 89 45 FC           mov     [ebp-4], eax            ; [ebp-4] points to
import data
```

The call at 410047 decrypt a block data in the exe containing all the information about the protected program, like import data, dips to be performed, various hash results, decryption keys and assign each of these data with a signature tag. In this case the signature tag is named "sign_13_import_data" by me. The procedure search for this signature in edx and output eax pointing to the desired block of data. Hence, rename this procedure and you will that it is called everywhere in the dll!

Look into the second last routine, we'll see how pre-OEP dipping is done. Using the call Decrypt_n_Find above, it searches for data blocks with Dip_signatures in the large chunk of decrypted data. Each of these data block contains a dipping address in the main program code section and the signature represents different kind of duties these dippings perform. Here are examples of the dips :

```
00411133 83 3D A4 45 41 00+   cmp     ds:dip_B_address, 0
0041113A 74 0B                jz      short loc_411147   ; dip_B : redirect
TApplication$Initialise
0041113C 68 00 CE 40 00       push    offset @Forms@TApplication@Initialize$qqrv ;
00411141 FF 15 A4 45 41 00    call    ds:dip_B_address
00411147
00411147                  loc_411147:                     ; CODE XREF:
set_seh_2A+2C2□j
00411147 C7 45 C0 F7 27 00+   mov     dword ptr [ebp-40h], 27F7h
0041114E 8D 85 B5 D7 FF FF    lea     eax, [ebp-284Bh]
```

OR here :

```
004111C9 83 3D 84 45 41 00+   cmp     ds:Dip_3_address, 0
004111D0 74 16                jz      short loc_4111E8
004111D2 68 54 CD 40 00       push    offset nullsub_3
004111D7 FF 15 84 45 41 00    call    ds:Dip_3_address      ; dip_3 : simple return
004111DD 68 54 CD 40 00       push    offset nullsub_3
004111E2 FF 15 88 45 41 00    call    ds:dip_4_address      ; dip_4 : simple return
004111E8
004111E8                  loc_4111E8:                     ; CODE XREF:
set_seh_2A+358□j
004111E8 83 7D C0 00          cmp     dword ptr [ebp-40h], 0
004111EC 75 4C                jnz     short registry_data_found_already
```

As you can see, there are different type of dips, ranging from @Forms@TApplication@Initialize$qqrv (which probably only applicable to Delphi apps), Decrypting parts of code section to a simple "ret". I think there are about 11 types of dips but some of them are very similar. ReGet only used 2 dips to decrypt some parts of the code section so I was not able to debug much of this D-D business, mainly analysing the dead listing. As you can see from the name of routine, it used you hard disk information as hash key, store the hash in registry, access system date etc... these dippings can get really wild :>

Finally, the last routine listed above are the self hash routine that check for error in its own dll code, so that if you place a breakpoint somewhere there, its opcodes will be replaced by "CC" and AsProtect will be able to detect it and exit. Here is one example :

```
0040E74C 68 54 C3 84 15     push    1584C354h         ; [esp] value
0040E751 68 AC 0F 00 00     push    0FACh  ; end of area to be hashed???
0040E756 68 9C D7 00 00     push    0D79Ch  ; start area to be hashed???
0040E75B 68 00 90 01 00     push    19000h  ; RVA of rsrc section (hash data)
0040E760 FF 35 14 40 41 00  push    ds:hInstance ; base image of dll
0040E766 E8 31 E8 FF FF     call    self_hash
0040E76B 31 04 24           xor     [esp], eax          ; test hash ([esp] =
1584C354h)
0040E76E 8B 05 14 40 41 00  mov     eax, ds:hInstance
0040E774 01 04 24           add     [esp], eax          ; if hash is wrong, ret
goes to wrong place
0040E774                                                ; seh is trigger and
program quit
0040E777 C3                 retn                        ; go to 4117D4 if correct
```

I am not quite sure about how exactly parameters are used in the Hash routine, too lazy to really trace into the routine in details to figure everything out, but I do have a rough idea of what is going on. As you can see, the hash result is used to decide where the program will go to next so if you are tracing and the hash is wrong, you would not notice it at all and will continue tracing until caught in seh and the program exits!

The above routine is repeated 1 more time to test the other half of the dll to make sure that no bpx escape its grasp ... but now we know how to defeat it :)... Once the hash check are OK, AsProtect proceeds to another memory area (loaded and decrypted by the dll) to perform the final task of calculating OEP and the famous "popad ; jmp eax"! I did not bother tracing and dumping this routine as it looks like a long boring nonlinear MD5 ... nah it wasn't MD5 but I don't see that I can learn much from it. AsProtect is more or less fully reversed.

5. API Mangling, a closer analysis

Okie, this section is going to discuss how AsProtect redirect API in more details so that hopefully you will be able to analyse other protectors the same way. After AsProtect decrypt the import data block, it has a Delphi-like structure with first dword is the signature (remmeber?), next dword is length, then followed by blocks of each library. The library block start with position of the first thunk offset, then dll name, then followed by each import entries. The import entries consist of first byte as group classification, second byte is length of the entry then import ASCII (encrypted of course).

The first byte is 01 then there will be no mangling, 03 is GetProcAddress, 04 is import by ordinal, 05 is redirect API and 06 is emulated API!!! You can find this whole API mangling routine starting from 41011A.

I would like to discuss about class 05 :> (which is quite interesting how AsProtect scan first few instruction of API, copy them to the redirected API location etc..)...

```
0040FD57                    check_next_instruction:  ; CODE XREF: Mangle_IAT+76□j
0040FD57 E8 84 FF FF FF       call  get_instruction_Table
0040FD5C 8B D8                mov   ebx, eax
0040FD5E C6 44 24 0C 00       mov   [esp+10h+copied_flag], 0 ; clear flag
0040FD63
0040FD63                    check_instruction_start_byte: ; CODE XREF:
Mangle_IAT+6F□j
0040FD63 0F B6 33             movzx esi, byte ptr [ebx]   ; first byte to esi
0040FD66 8D 43 01             lea   eax, [ebx+1]          ; the second byte is
stored in edi
0040FD66                                                  ; for later usage
0040FD69 0F B6 38             movzx edi, byte ptr [eax]
0040FD6C 8D 53 02             lea   edx, [ebx+2]          ; third byte onwards
0040FD6F 8B CE                mov   ecx, esi              ; first byte
0040FD71 8B C5                mov   eax, ebp              ; original API address
0040FD73 E8 E8 C6 FF FF       call  CompareBinary         ; compare binary string in
eax and edx
0040FD73                                                  ; with length ecx
0040FD78 84 C0                test  al, al
0040FD7A 74 18                jz    short not_equal
0040FD7C 8B CF                mov   ecx, edi              ; second byte
0040FD7E 8B D5                mov   edx, ebp              ; original API adddress
0040FD80 8B 44 24 08          mov   eax, [esp+10h+new_mem_pointer] ; newly allocated
memory address
0040FD84 E8 7F 47 FF FF       call  Move_memory           ; copy a few bytes from
original API
0040FD84                                                  ; adddress to the newly
allocated
0040FD84                                                  ; memory to redirect the
API
0040FD89 01 7C 24 08          add   [esp+10h+new_mem_pointer], edi
0040FD8D 03 EF                add   ebp, edi              ; calculate the position
for
0040FD8D                                                  ; the redirected API to
jump back
0040FD8D                                                  ; to the original API
0040FD8F C6 44 24 0C 01       mov   [esp+10h+copied_flag], 1 ; set the API
redirected flag
0040FD94
0040FD94                    not_equal:                    ; CODE XREF:
Mangle_IAT+46□j
0040FD94 83 C6 02             add   esi, 2
0040FD97 03 DE                add   ebx, esi              ; point ebx to next API
redirection
0040FD97                                                  ; data block
0040FD99 80 7C 24 0C 00       cmp   [esp+10h+copied_flag], 0
0040FD9E 75 05                jnz   short loc_40FDA5      ; is API already
redirected?
```

```
0040FD9E                                                ; jump if yes
0040FDA0 80 3B 00            cmp   byte ptr [ebx], 0    ; the end of the API
redirection
0040FDA0                                                ; block??
0040FDA3 75 BE               jnz   short check_instruction_start_byte ;
0040FDA5
0040FDA5                   loc_40FDA5:                  ; CODE XREF:
Mangle_IAT+6A□j
0040FDA5 80 7C 24 0C 00      cmp   [esp+10h+copied_flag], 0
0040FDAA 75 AB               jnz   short check_next_instruction
```

Basically, the routine above check the first few bytes of each instruction in the original API routine, compared with a pre-stored table of instructions and decide how it should copy the routine over to the redirected API. The first call of the routine at 40FD57 "call get_instruction_table" simply points eax to the beginning of the instruction table. The loop goes on until an "unknown" instruction is found, that is an instruction is not defined in the pre-stored table. The pre-stored table looks something like this

```
0040FCFB 01                  db    1 ;                  ; no. of bytes to compare
0040FCFC 01                  db    1 ;                  ; number of bytes to copy
0040FCFD                   ; -------------------------------------------------------
---------
0040FCFD 57                  push  edi
0040FCFD                   ; -------------------------------------------------------
---------
0040FCFE 01                  db    1 ;                  ; no. of bytes to compare
0040FCFF 02                  db    2 ;                  ; number of bytes to copy
0040FD00 6A                  db  6Ah ; j                ; 6Axx  ==> push  xx
0040FD01 01                  db    1 ;                  ; no. of bytes to compare
0040FD02 05                  db    5 ;                  ; number of bytes to copy
0040FD03 68                  db  68h ; h                ; 68xxxxxxxx ==> push
xxxxxxxx
0040FD04 02                  db    2 ;                  ; no. of bytes to compare
0040FD05 03                  db    3 ;                  ; number of bytes to copy
0040FD06 FF                  db 0FFh ;                  ; FF75xx ==> push dword
ptr [ebp+xx]
0040FD07 75                  db  75h ; u
```

Look back at the loop above again, you will understand how it scan through the table to find the right instruction, for example when the routine scan through the table, at 40FD04, the first byte is 02 so that means it compares the first 2 bytes of the current instruction in the original API with 2 bytes starting from 40FD06, if same then the instruction is "push dword ptr [ebp+xx]" hence copy the next 3 bytes (next instruction) over to the redirected routine. This is how it can copy the full instructions without using a disassembler.

After leeching as much as possible from the original API, AsProtect sets about to create the final jump to bring the redirected API to the original API... this is again mundane calculation of number of byte copied etc... Only the last interesting bit is it uses a Random number to decide which kind of call back to use, "push xxxxxxxx ret" or a long jump!

```
0040FDDA B8 02 00 00 00      mov   eax, 2
0040FDDF E8 D0 29 FF FF      call  @System@@RandInt$qqrv ; System __linkproc__
RandInt(void)
0040FDE4 83 E8 01            sub   eax, 1                ; randomize between 0 and
1 :>??
0040FDE4                                                 ; 2 options f returning to
the original
0040FDE4                                                 ; API?? a push ret or a
long jump?
0040FDE7 73 1A               jnb   short push_return
0040FDE9 8B 44 24 08         mov   eax, [esp+10h+new_mem_pointer]
0040FDED C6 00 E9            mov   byte ptr [eax], 0E9h  ; setting up the long jump
E9xxxxxxxx
0040FDF0 83 C5 05            add   ebp, 5
0040FDF3 8B 04 24            mov   eax, [esp+10h+API_address]
0040FDF6 2B C5               sub   eax, ebp              ; calculate the relative
distance
0040FDF6                                                 ; to put in after E9
0040FDF8 03 F0               add   esi, eax
0040FDFA 8B 44 24 08         mov   eax, [esp+10h+new_mem_pointer]
0040FDFE 40                  inc   eax
0040FDFF 89 30               mov   [eax], esi            ; update the redirected
API witht the
0040FDFF                                                 ; distance found
0040FE01 EB 1B               jmp   short done
0040FE03                   ; ---------------------------------------------------
------------
0040FE03
0040FE03                   push_return:                  ; CODE XREF:
Mangle_IAT+B3□j
0040FE03 8B 44 24 08         mov   eax, [esp+10h+new_mem_pointer]
0040FE07 C6 00 68            mov   byte ptr [eax], 68h   ; setting up a push,
68xxxxxxx
0040FE0A 03 34 24            add   esi, [esp+10h+API_address] ; calculate the
return address
0040FE0D 8B 44 24 08         mov   eax, [esp+10h+new_mem_pointer]
0040FE11 40                  inc   eax
0040FE12 89 30               mov   [eax], esi            ; update the redirected
API
0040FE14 8B 44 24 08         mov   eax, [esp+10h+new_mem_pointer]
0040FE18 83 C0 05            add   eax, 5
0040FE1B C6 00 C3            mov   byte ptr [eax], 0C3h  ; finally put a "ret", C3
0040FE1B                                                 ; to go to the address
pushed above
0040FE1E
0040FE1E                   done:                         ; CODE XREF:
Mangle_IAT+CD□j
0040FE1E 8B C7               mov   eax, edi
```

Rather interesting! I hope everything is clear. There is one more routine that deals with type 06, emulated APIs and it is left as a practise for readers to locate and analyze this routine... it is not that simple though!

6. Finally

I hope you have learnt something from this long tutorial. This is my first so there are bound to be mistakes, PLEASE contact me and help me correct them (I can be found at RCE board most of the time). I hope that you will be able to use these info to unpack AsProtect better, or to inline patch it, to remove CRC check and all...

This tutorial would not have been possible without the following people : Spl/j, evaluator, Solomon, SpeKKel, FoxThree, sv (for helping me unpacking my first AsProtected program), Tsehp (for revirgin), Clandestiny (for answering seh stuff), Daemon (for anti-debugging/tracing stuff on his site) and last but not least R!sc (the old genius) for his excellent tutorials on unpacking.... I am sorry if I miss someone out, but you know that I am always grateful for your help, it is the thought that counts heh :>

Special thanks to Kayaker for some analysis on API mangling and of course, give me more work to do :<... so if you find this essay too long, blame him :>>

Last Edited : 30 April 2002

Cheers,

crUsAdEr

10 Source of the Issue – 61 Byte Mandelbrot-Generator

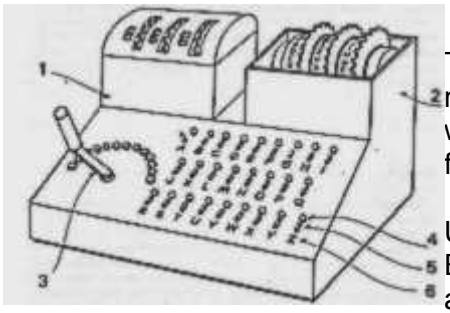
This is a small implementation of a mandelbrot generator. I've found this gem a some time ago in a swedish fido-net meeting as a UUencoded file. All comments have been inserted by me (John Eckerdal). I have tried to give some information about what the program acutally calculates. This information might however be incorrect.

```
;
; mandelbrot plotter, 61 bytes - Tenie Remmel
;
; assumes on startup:
;   ax = 0000h
;   di = fffeh
;

        IDEAL
        MODEL TINY
        P386
        CODESEG
        ORG 100h

start:
        push    0A000h          ; set segment
        pop     es
        mov     al,013h         ; set video mode
        int     10h
        stosw
        mov     cl,200          ; screen height
outer_loop:
        mov     si,320          ; screen width
inner_loop:
        mov     bp,79           ; #iterations
        xor     bx,bx           ; zero re-part
        xor     dx,dx           ; zero im-part

complex_loop:
        push    dx
        mov     ax,bx
        sub     ax,dx
        add     dx,bx
        imul    dx              ; u:=re^2-im^2;
        mov     al,ah           ; u:=u/100h
        mov     ah,dl
        pop     dx
        xchg    bx,ax
        sub     bx,si           ; new_re:=u-width;
        imul    dx
        shld    dx,ax,9         ; 2*re*im
        sub     dx,cx           ; new_im:=2*rm*im-height
        test    dh,dh           ; if j>=256 plot pixel
        jg      short plot_color
        dec     bp              ; next iteration
        jne     short complex_loop

plot_color:
        xchg    bp,ax
        stosb                   ; plot pixel
        dec     si
        jne     short inner_loop
        loop    short outer_loop

        ret                     ; end program

end     start
```

11 Crypto of the Issue – The German Enigma and Polish Decrypting Effort (1930-1939)

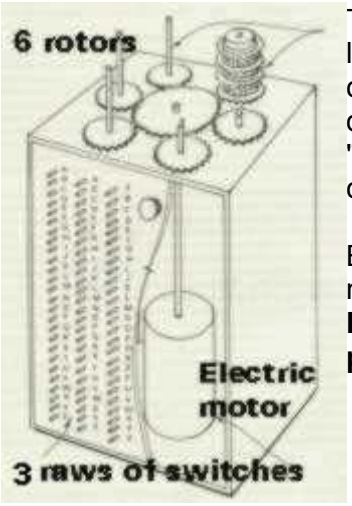
11.1 Historical Efforts

Early in the decrypting effort, Polish cryptologists devised a simple method to solve the patch panel interconnect, which **consisted of two paper strips**. One strip had six holes representing the six panel interconnect the Germans used initially. This paper strip was overlaid on another paper strip that had Enigma machine rotors encrypting series. When the top overlay and the bottom paper strip aligned with the same letters, this became the solution for the patch panel interconnect. This method worked well for a time until the Germans started changing the number of patch panel interconnects and the method lost its value. Fortunately, the Polish cryptologists came up with much better method that consisted of **an assembly of two Enigma machines**, which was able to go quickly through many combinations generated by the three rotors and patch panel of the Enigma machine. There were 6 patch panel combinations and 26 (for each letter of the alphabet) times 26 times 26 rotor combinations, giving a total of 105,456 combinations. In less than 20 minutes they were able to find solution for the Enigma set up and to decrypt messages.



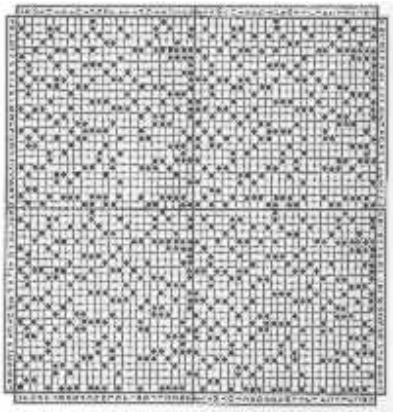
They called this machine **"Cyclometer"** i.e. machine that measured the Enigma machine cycles. In few years they were able to set up a library for over 80,000 typical set ups for Enigma.

Unfortunately, in November 1937, the Germans changed Enigma again. Within less than a year, the Poles were again able to break the code and by mid-1938 they reached the peak of their operation. Suddenly, in mid-September 1938 a number of German messages could not be decrypted! Polish intelligence agents in Germany found that the Germans were changing the initial rotor positions not once a day but with every message so the existing methods were no longer adequate. M.Rejewski started working on the problem at once and came up with new, faster and more powerful approach. It consisted of **six Enigma machines connected together** and driven by a single motor.



They called it **"Bomba"**, a term that was used by the French and later by the British at Bletchley Park. Using "Bomba", all combinations could now be examined in two hours. Since fast decryption was of utmost importance a system consisting of six "Bombas" was designed and production was given to AVA, the company previously used.

By November 1938 the system was operating and German messages were again being decrypted. In addition to the "Bomba", **H. Zygalski developed a completely new method using perforated paper sheets**.



Each sheet had 51 by 51 squares and about 1000 holes arranged in a pattern. Twenty six sheets, each for each rotor position, were required. As the sheets were superimposed and adjusted on each other, holes shone through giving possible solutions. Six sets of these were required for finding possible Enigma settings. The advantage of the perforated sheets over the "Bomba" was that they were not affected by the patch panel setting.

By December 1938, the Germans again introduced new changes which lengthened the encryption sequence and the existing system, though still sufficient, took a long time to obtain a solution. The Polish team was now reaching the limit of their resources and it was time for the start of the second World War.

In July 1939, the Germans added rotors 4 and 5 to their machines. This information was gained through Polish intelligence in Germany and cryptologists soon confirmed this by analysis of the new messages. On the basis of new messages and errors committed by German operators, the Polish cryptologists were able to deduce the interconnections in the new rotors. This was possible because the Germans did not, for some time, use new encryption methods and consequently some messages were still decryptable but took about 10 times longer! The only solution was to build a more powerful machine consisting of **60 "Bombas"** or **60 sets of perforated sheets**. This change of pace, shortage of resources and the imminence of war caused the **Polish Chief of Staff, Gen. W. Stachewicz, to decide to pass all the information gained by the cryptologists, to the Allies, i.e. the French and the British.**

This had to be done under the greatest security as German spies were everywhere. A conference was set for **25-27 July to which the French and the British crypto specialists were invited**. The conference participants representing France, Britain and Poland were as follows - France Colonel G. Bertrand, radio intelligence Captain H. Braquenie,intelligence Britain Colonel S. Menzies, Chief of British Intelligence **A. Denniston**, Chief of G.C.C.S. **A. Knox**, a cryptologist member of G.C.C.S. Poland Colonel G. Langer, Chief of Cryptology Major M. Ciezki, Chief o German section (BS4) M. Rejewski, cryptologist J. Rozycki, cryptologist H. Zygalski, cryptologist A. Palluth, mechanical engineer from AVA C. Betlewski, mechanical engineer from AVA The conference was held in the Pyry Forest near Warsaw to assure utmost privacy. H. Zygalski presented the Polish made Enigma and explained how it worked. He also explained how they broke the code by mathematical analysis, coupled with information provided by the French intelligence officer Capt. G. Bertrand. The next presentation was by M. Rejewski who demonstrated the six Enigma machines ("Bomba") which they were using to break the code. On the second day it was explained that the Germans were now using 5 rotors Enigmas and that since the decrypting process took 10 times longer the existing system was no longer practical. Further, the Germans had introduced a patch panel with 7 to 10 interconnects which further complicated the problem. H. Zygalski presented his alternative method, using perforated sheets, but the multiplicity of sheets made it slow and cumbersome.

Col. G. Langer, the Polish Intelligence Section Chief, then addressed the French and British delegations stating that Poland expected to be attacked by the Germans very soon and that the Polish government had decided to pass all the information, drawings, perforated sheets and Enigmas to the Allies.

Poland did not have sufficient resources and time to continue with the effort. **Within two weeks the French received two Enigmas via diplomatic channels. On August 16th Capt. G. Bertrand, accompanied by Commander W. Dunderdale the British Intelligence Resident in Paris, passed to the British a package containing an Enigma machine and papers relating to breaking the code**. Two weeks later, on September 1st 1939, WWII begun.

11.2 The Enigma rotor-type ciphering machine

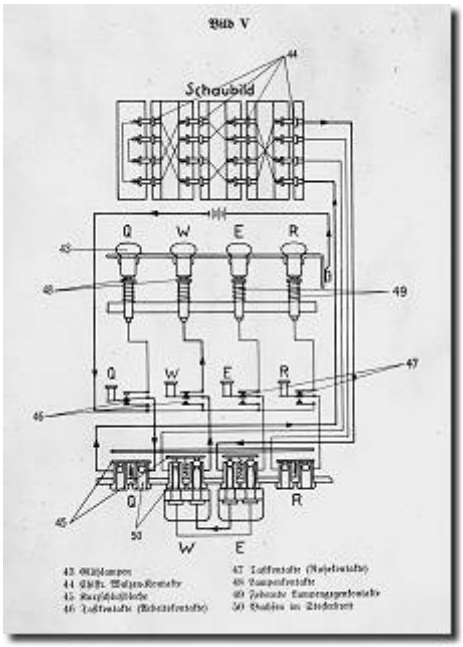
Among the special machines that were invented and further developed over several decades to simplify the routine enciphering and deciphering of code, the Enigma machine is probably the most well known example around the world. It was used during the Second World War to encipher most of the radio messages of the German Armed Forces (Wehrmacht) before these commands were transmitted, and to decipher them after they had been received. It is possible that between 100,000 and 200,000 Enigma machines were built during World War II.

From the time of Enigma onwards, the history of computer technology was strongly influenced by enciphering techniques and the efforts expended in breaking unknown codes. Today, as computers and digital telecommunications are increasingly connected together, enciphering plays a bigger role than ever before. And the Enigma story leads one to suspect that everything in the history of the world might have been changed by the systematic alteration of letters. As our knowledge grows concerning the specific events that make up the historical entity of World War II, our assessment of the role of enciphering and code breaking becomes better balanced and more discriminating. We must remember that the initial plan was to drop the atomic bomb on Germany, and that it was Japan's cities that were decimated only because the war had ended earlier in Europe. It soon becomes clear how many alternatives were possible. Thus, Enigma's role as hero is rather a negative one as far as its technical superiority and importance to the German final victory are concerned. To continue the metaphor: it became the pedestal on which the heroes of code breaking still stand today.

The radio messages enciphered by Enigma machines were deciphered during the war at the British Cipher Bureau, Bletchley Park, despite the new technical refinements that the device gained on so many occasions. Finally, the Allies were able to intercept this strand of German military radio messages, with only a very few exceptions.

11.3 The inventor and the operators

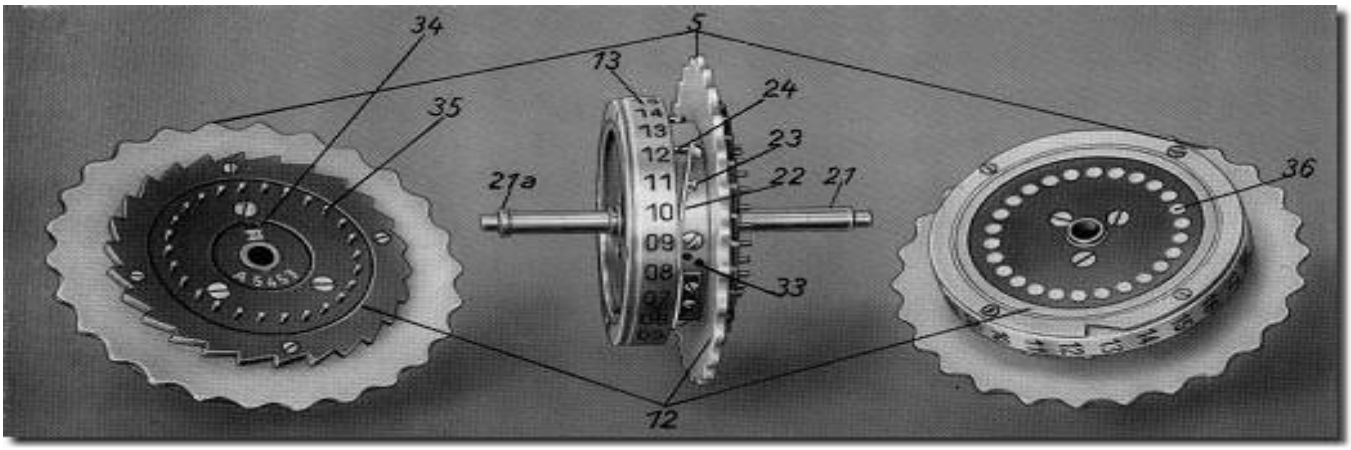
The invention of the working principle used in Enigma dates back to the first years of World War I, when Edward Hugh Hebern (1869-1952), an American building contractor, invented in 1917 a rotating device for poly-alphabetic substitution using independent alphabets. He offered his ciphering machines to the American military. In 1918 the engineer Arthur Scherbius (1878-1929) filed the rotor principle with the patent office. He built a machine that he designated "Enigma" - the Greek word for riddle - in his "Chiffriermaschinen AG" in Berlin before demonstrating it to the public in Bern in 1923, and then in 1924 at the World Postal Congress in Stockholm. The Enigma was not much of a secret in those days. In 1927 Scherbius bought the patents of the Dutchman Hugo Alexander Koch, who had reinvented the rotor principle in 1919. After the death of Scherbius in 1929, Willi Korn was in charge of further technical development of Enigma. In 1919 the rotor principle had also been invented independently in Sweden. In all these countries, government agencies initially showed little interest in the machines. In Germany, the Enigma was taken over by the Reichswehr (German Army of the Reich), and when the massive military build-up began under Adolf Hitler in 1933, the Enigma machine remained part of the programme. Although the Enigma was the encoding machine most often used by German agencies during World War II, it was not the only one. Some strategic messages were enciphered with a number of devices that were even more complex.



Circuit diagram of the Enigma rotor-type ciphering machine

From 1933 onwards the Enigma was constantly developed for use by the army, the navy, and the diplomatic service. Scherbius had already supplemented the rotor system with a plug board in 1928, which made enciphering even more complicated. The working principle is based on a number of simple electric circuits. Each one connects a key of the typewriter-like keyboard with a glow lamp that illuminates a letter in the display panel. Every keystroke causes the illumination of a new letter. Each current path leads to one of the three rotors, always via one front and one rear contact, up to the reversing wheel, or reflector. From there, it leads back through all the rotors and also through the plugs of the plug board. Enciphering takes place through a rather complicated system: the input and output contacts that are positioned opposite to each other are not electrically connected with each other via each rotor but cross-connected according to a certain system. The plug board connections are varied. Every time a key is pressed, the rotors are advanced one position according to a preordained system. Three of these rotors are always selected from a set of five and used in a sequence that is determined anew on each occasion. The initial setting of the rotors is always reset again, too. A setting ring is mounted on each rotor. Whilst in operation during the war, the various settings were changed every eight hours.

11.4 The code breakers



Rotors of the Enigma rotor-type ciphering machine
Setting wheels (5), enciphering rotors (12), number ring (13), axle (21), axle collar (21a), retaining spring (22) button (23), spring-loaded stud (24), marking by means of dots (33), marking by means of Roman numeral (34), sprung contact pins (35), smooth contact surfaces (36)

The message can only be deciphered if the recipient knows all settings of the Enigma machine that is sending. He can then use the 'same' Enigma machine to reverse the ciphering and read the fully deciphered text. The enemy's code breakers had to ascertain the physical design of the machine and its working principle. Using the analysis of the radio messages intercepted, they were forced to calculate the setting configuration anew. This had to be done within a certain period of time in which a military reaction to the deciphered information was still possible.

The Enigma machine in the Deutsches Museum

The collection of the Deutsches Museum includes three Enigma machines: one naval Enigma with four rotors, and two machines from the army, which employ three rotors. Together with other ciphering devices, two of these machines are held by the Computer department (sorry, these informations are only in German). Fairly often, visitors to the museum start conversations purely by chance and identify themselves as users of Enigma during the war. A few years ago, a museum was opened in Bletchley Park, near Milton-Keynes, a city not very far from London. There, the Enigma story is presented. On display are the machines used for breaking its code. The methods used are also explained. A growing circle of historians and expert cryptologists have been dealing with the historic role of Enigma for years. It is not known how many computer simulations of Enigma there are already, but it is certain that this machine will continue to challenge both dilettante and professional programmers in the future.

Enigma rotor-type ciphering machine, version with 3 rotors, wooden housing, no. A 2178 K; ca. 1928

11.5 Simple "Enigma" Machine Encrypt/Decrypt machine - C-Source

```
//  Enigma.C - Simple "Enigma" Machine Encrypt/Decrypt machine
//
//  Myke Predko - 98.02.10
//
//  This code runs a Simple (Three Wheel) version of the WW2 German
//   "Enigma" Machine with a brute force decryption algorthim to
//   demonstrate how fast data can be decrypted by a Pentium Class
//   PC.
//
//

#include "stdio.h"  //  stdio.h In quotes so it will
      //   show up in html.

//  Define Message Strings Along with the Variable Messages
int i, j, k, n;

// Letter Positions:ABCDEFGHIJKLMNOPQRSTUVWXYZ

char Wheel_1[26] = "ZYXWVUTSRQPOMNLKJIHGFEDCBA";  //  Reverse Letters
char Wheel_2[26] = "BADCFEHGJILKNMPORQTSVUXWZY";  //  Translate Up 13
char Wheel_3[26] = "FGHIJABCDEPQRSTKLMNOXYZUVW";  //  Blocks of 5

char far * Msg    = "CANTHISBEDECODED";  //  Message to be encoded/
char far * Encode = "canthisbedecoded";  //   decoded.
                                  //  "Encode" points to space for Msg
//  Define the Wheel Operations
char Encode1( char Input )
{                                 //  Convert the Letter using "i"
  return Wheel_1[( i + Input - 'A' ) -
                                      (( i + Input - 'A' ) / 26 * 26 )];
}  //  End Encode1

char Encode2( char Input )
{                                 //  Convert the Letter using "j"
  return Wheel_2[( j + Input - 'A' ) -
                                    (( j + Input - 'A' ) / 26 * 26 )];

}  //  End Encode2

char Encode3( char Input )
{                                 //  Convert the Letter using "j"

  return Wheel_3[( k + Input - 'A' ) -
                                      (( k + Input - 'A' ) / 26 * 26 )];

}  //  End Encode3



char ReadDecode( Input )
{                                 //  Figure out what the Character is
                                  //   For Input, i and j

char retvalue;                    //  Character to be Returned
int  count;
```

```
  for ( count = 0; Wheel_3[ count ] != Input; count++ );

  if (( count = count - k ) &lt; 0 )
    count += 26;                //  Get a Valid Character Position

  retvalue = count + 'A';       //  Convert to ASCII

  for ( count = 0; Wheel_2[ count ] != retvalue; count++ );

  if (( count = count - j ) &lt; 0 )
    count += 26;

  retvalue = count + 'A';       //  Convert to ASCII

  for ( count = 0; Wheel_1[ count ] != retvalue; count++ );

  if (( count = count - i ) &lt; 0 )
    count += 26;

  return count + 'A';

}  //  End ReadDecode


EnigmaInc()                     //  Rotate the Wheels
{

    if ( ++i &gt; 25 ) {           //  Now, Rotate the Wheels
      i = 0;                    //  "i" is Rolling Over
      if ( ++j &gt; 25 ) {
        j = 0;
        if ( ++k &gt; 25 )
          k = 0;
      }  //  endif
    }  //  endif

}  //  End EnigmaInc


main()
{

int Wheel_1Start = 16;          //  Set the Initial Wheel Positions
int Wheel_2Start = 25;          //   As part of the Code
int Wheel_3Start = 17;

  i = Wheel_1Start;             //  Setup the Checking Parameters
  j = Wheel_2Start;
  k = Wheel_3Start;

  for ( n = 0; n &lt; strlen( Msg ); n++ ) {
    Encode[ n ] = Encode3( Encode2( Encode1( Msg[ n ])));
    EnigmaInc();                //  Get the Encrypted Character
  }  //  endfor

 printf( ""%s" is encoded into "%s"
", Msg, Encode );
```

```
//  "Encode" is loaded with the Encoded Message

  for ( i = 0; i &lt; 26; i++ )    //  Now, Find "i", "j" and "k" to
    for ( j = 0; j &lt; 26; j++ )  //   Decrypt
      for ( k = 0; k &lt; 26; k++ ) {
        Wheel_1Start = i;         //  Save Starting Values for Looping
        Wheel_2Start = j;
        Wheel_3Start = k;
        for ( n = 0; ( n &lt; strlen( Msg )) &amp;amp;&amp;amp;
                     ( Msg[ n ] == ReadDecode( Encode[ n ])); n++ ) {
          EnigmaInc();
        }  // endfor
        if ( n == strlen( Msg ))
          printf(
            "Decode Wheel Positions: 1 -&gt; %i, 2 -&gt; %i, and 3 -&gt; %i
",
                           Wheel_1Start, Wheel_2Start, Wheel_3Start );
      i = Wheel_1Start;         //  Restore the Wheel Values
      j = Wheel_2Start;
      k = Wheel_3Start;
    }  //  endfor

}  //  end Enigma
```

12 Algorithm of the Issue – The "Travelling Salesman" problem[8]

12.1 Introduction

A salesman has to visit *N* cities with given distances d_{ij} between cities *i* and *j*, returning finally to his city of origin. Each city is to be visited only once, and the route is to be made as short as possible. A popular special case is the Euclidean TSP, where the cities are given by their positions (x_i,y_i) in the plane and the distance matrix is given by the Euclidean distance,

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The Travelling Salesman Problem (TSP) is the most prominent problem in combinatorial optimization. It has attracted - and still attracts - researchers from various fields.

Any known algorithm that claims to find the true optimum tour has a running time that grows exponentially with the number *N* of cities. Since TSP is NP-complete this will hold for any future algorithm - at least with high probability. Therefore efforts have concentrated on the development of heuristic algorithms, that do not aim to find the shortest tour but a tour that is reasonable short. Such approximate algorithms incorporate ideas that range from simple to highly sophisticated, and some of them give results that come very close to the optimum solution.

This is an introduction to some these approximate TSP algorithms.

Figure 1 shows a diagram of the Travelling Salesman problem (Hamiltonian path problem). The objective is to find the shortest path from start to end going through all the points only once. This problem is difficult for conventional computers to solve because it is a "non-deterministic polynomial time problem" (NP). NP problems are intractable with deterministic (conventional/serial) computers, but can be solved using non-deterministic (massively parallel) computers. A DNA computer is a type of non-deterministic computer. The Travelling Salesman Problem was chosen because it is known as "NP-complete"; every NP problem can be reduced to a Travelling Salesman problem.

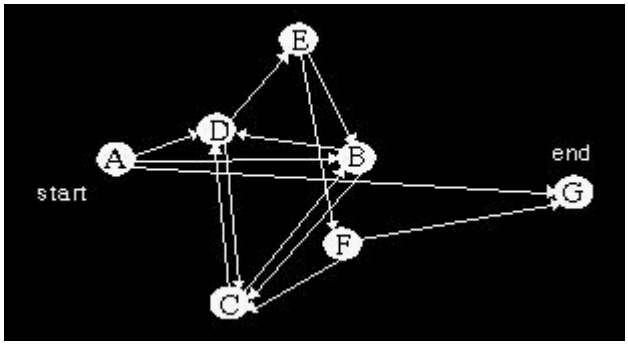


Figure 1 : The Travelling Salesman problem. The goal is to find a path from the start city to the end city going through every city only once

[8] Partly taken from http://www.geocities.com/Eureka/Enterprises/1397/project3.html

12.2 Solving the Problem

The following algorithm solves the Travelling Salesman Problem:

1. Generate random paths through the graph.
2. Keep only those paths that begin with the start city (A) and conclude with the end city (G).
3. If the graph has n cities, keep only those paths with n cities. (n=7)
4. Keep only those paths that enter all other cities at least once.
5. Any remaining paths are solutions.

13 VX-Knowledge for the Reverse-Engineer - Lord Julus' Anti-Debugger & Anti-Emulator Lair

13.1 DISCLAIMER

The following document is a study. It's only purpose is to be used in the virus research only. The author of this article is not responsible for any misuse of the things written in this document. Most of the things published here are already public and they represent what the author gathered across the years. The author is not responsible for the use of any of these information in any kind of virus.
Lord Julus.

13.2 Before F0reW0rd - W0rd ;-)

Due to the fact that I was very anxious to release this, and the fact that while writing it my computer got burned, and that, anyway I was sick and tired of looking at it anymore, I released it in a, let's say for now Version 1.0. As soon as I'll feel again ready to write, I shall come with more ideas and stuff. For now just read this and don't kick me if you find any mistakes I didn't have time to correct... Anyway, during the writing of this I kinda felt a little more on the encryption side, which actually is the basis of a good fight with an AV. You got an unbeatable encryption, you rule! So, don't be frightened by the math involved here: everything is explained. Secondly, also while writing this article I got involved in Win32 programing. This made me leave the mortal's world for a while ;-) and go in higher circles. So, just read along...

13.3 Foreword

Well, my dearest friends and enemies (;-)), here I am again, not really having much to do these days but work and code... Alongside these, I would name, "high" things to do, I still have time to study, analyze and check various stuff around. Since a little while back, I started a campaign of writing anti-anti-viral programs. These would be like memory TSR's bypassers and memory patchers and searchers. Well, I looked deep and now I decided it's time I put it down in words, black on white (or more like white on blue, as I see it now ;-)).

Anyway, for those of you who know jam about what's debugging and emulating, I will try to make a short description here in the foreword on the debug process, emulating process and some other stuffs.

Here come the descriptions of the terms about to be used here (definitions taken out of the Webster, and additional explanations by me):

DEBUG = "To detect and remove defects or errors from smth."
DEBUGGER (in comp. sense) = A tool to debug code

In common language, this 'debug' term has enlarged the specter, no longer meaning only detecting and removing errors, but also simply looking over the code. We'll take a look later at the most common debuggers.

EMULATE = "Try to equal or excel; imitate with effort to equal or surpass"
EMULATION = "Effort or desire to equal or excel with others"
EMULATOR (in comp. sense) = A tool to emulate code

This term also has a different connotation in the computer business. It doesn't really means making a program in order for it to be able to imitate someone else's program. This would be, of course, stupid. If you want to copy the best tool is 'copy`n`paste' ;-)). Anyway, the emulator is a piece of code, usually very complicated, actually much more complicated then the code to be emulated itself, which has the capability to take a program instruction by instruction and imitate what that program would do if it were ran. BUT, the emulator will never allow a program to really do what it should do. It only tries to come to that program's goal and guess it. This comes to the next definition:

HEURISTIC = "stimulating interest as means of furthering investigation"

Actually, the best definition for this term can be found in the polymorphic tutorial written by The Black Baron. It reads: 'heuristic = A set of well defined rules to apply to a problem in the hope of achieving a known result'. Hope you got it...

Anyway, since the beginning of the viral activity, somewhere in 1987, the anti-virus-writers had to use some powerful tools. We all know that it's much harder to build, restore or repair something then to destroy or damage it (take a life example: hit your TV screen with a hammer... it only takes a second... than try to repair it ;-))), and also it's always much easier to prevent something bad but to restore it's damage afterwards.
Just like Confucius said: "Those who do not see the danger coming shall surely suffer from the urge approaching". Therefore, the antiviral community have started to build certain types of tools in order to come in a not so fair fight (thousands of virus writers and a couple of AV guys ;-)).

Mainly, the developed tools are these:

1) TSR blockers/checkers
2) String scanners for memory/files/places on disks
3) Heuristic scanners / code emulators

Let's define them quickly and start the real thing:

TSR Blockers/Checkers

This category of AV utilities is largely used and was made famous by SAFE (one of the most known TSR utilities). Their main purpose is prevention. They do not clean viruses, but stay here in memory and before any operation is done they check... If something strange pops up (like a write-to-disk, an executable change) they have the nasty habit to flush a (usually) red window on the screen warning about the danger. Others are blockers for different viruses. Usually a virus checks whether it's resident or not by calling an interrupt with certain values and waiting for something in return. A TSR blocker will simulate the virus by returning the 'already resident' values. In this way, even if your files are infected, the virus will never go resident.

String scanners

Any virus, like any piece of code actually is made of those tiny, little 0's and 1's called bits, which form those pretty nice 8 bit thingies called bytes, which put in pairs form those really nifty words (and no, I don't think you're stupid ;-)). Anyway, a code has this thing called a signature. A string scanner will search in a file/memory or anywhere else for a set of bytes and will decide whether it's a virus or not. Smart scanners allow smart wildcards, like scan x bytes, then jump over the next 3 bytes, scan other 2, and so on... Anyway, even with the growing popularity of the polymorphic viruses, most of the viruses around can be detected with a signature.

Heuristic Scanners / Code Emulators

Let's imagine you write a new virus... Of course, there is no one who knows any signature for your virus for the simple reason that it's new. Here the heuristic analyzers come around. These 'look' into your code and set some levels of danger for that particular code. For example if a heuristic scanner finds a check for 'load & execute' command, it will probably warn the user. The code emulator does more. It simulates the code execution by putting 'by hand' values into the registers and trying to 'see' what the code does. This method is essential for new viruses and for polymorphic viruses.

13.4 First approach

Ok, now we have defined our 'environment', sort of speak, so let's start talking a little deeper about each of the above AV-types.

The TSR-blocker... Yeah... This one is the easiest type of AV to go around. There are a lot of TSR-blockers out there... If you feel threatened by any one of them, simply disassemble the darn thing an check out the method it uses in order to check. There are several ways they use. The most common is to monitor interrupts 21h, 13h, 76h, 25h, 26h. All these can be overridden by simple tunneling/tracing routines. But, many TSR shits have some anti-tunneling routines that might warn the user about tunneling in progress. But, this is not a matter to speak about in this article.

Another largely used method is the monitoring of the INT 03/01. This means that the TSR checks every command your program does and decides if a couple of commands are dangerous or not. These are usually crap, because they slow down everything. However, this type of TSR blocker gets killed by the Prefetching Queue.

Another method is the monitoring of the INT 08 (the clock interrupt).
Using this interrupt the AV checks various things. A simple CS:IP check type tunneling and you override it's checks. However, be careful to leave it like it would work. You may need to disassemble it and find particular bytes to patch (some of them use INT 08 also to trigger keyboard event, like the pop-up of the options menu; and if the options menu doesn't work, the user may notice).

Anyway, the TSR blockers can be easily killed as you saw. However, the AV guy might create a specific TSR-blocker for your specific virus (for example by making a TSR that returns your virus' `already resident` signature, but this means your virus is really good or you really pissed of some guy...)

In order to kill the string scanner all you need to do is to put in your virus a well random-oriented polymorphic decryptor. In this way you're safe from any kind of string scanning. Right after the polymorphic decryptor has finished it's job, it's time for another decryptor. You have got to create a well balanced set of decryptors, kinda like this:

-the more complicated the poly decryptor is, the less it polymorphic
-the longest the poly decryptor is, the hardest is for the emulator
to go thru (don't forget that there exist code emulators + scan strings;
they can go thru your poly decryptor and scan string the second decryptor)
-the more complicated the second decryptor is, it's break becomes
more difficult for the emulator

So, you need a balance. A well scan-string based AV will not have a very good anti-poly routine. This because the loading of the scan strings and searching for them takes long time. The same for the emulator. In order to create a good poly decryptor, check out the article I wrote on poly decryptors at http://members.tripod.com/~lordjulus.

So, the scan strings go down the drain too...

And finally I reached where I wanted to... The heuristic scanners and the code emulators. These are the most dangerous AV ever and they seem to be written by some smart guys (some of them ;-))... Anyway, the main disadvantage for the code emulator (called CE from here) is it's speed. As it must 'emulate' each instruction, it has to kinda do what the CPU would do, still however using the CPU... Therefore it's slow. Also, a lot of instructions are not emulated by them or some of them are emulated incorrectly. Further I will try to put up a set of methods I recommend you to use inside the second decryptor. In case the CE goes past your poly decryptor, it *must* hang in the second decryptor, otherwise, your virus is disclosed.

13.5 Anti - emulator code

One of the best methods in the fight with the AV code emulator is to find out pieces of code that generate a certain known result. The same result must be retrieved through another method, which also can be or can be not emulated by the AV. Having the two results one should do some operations with them over the most important registers. The idea around this is that if the AV is not capable to emulate one of the ways you retrieve a result, it will for sure use it's own result and will render to a fault. However, beware of comparisons and conditional jumps. What I'm referring to is this:
say you made two routines and one of them is for sure impossible to emulate.
The two routines return the same parameter. You put them in the AX and BX registers. If you do something like:

```
CMP AX, BX
JNE I_AM_BEING_EMULATED
```

The code emulator for sure will jump to the conditional jump address, as it cannot compute one of the arguments correctly. One could think that this solves the problem. No way ! As we think of devious methods, so do the AV writers. Anytime a conditional jump is encountered a good emulator will save it's place. If the condition is met it will jump there. The above jump would probably be one to an infinite loop, or program terminate, or halt or stuff like that. The good emulator will not stop, but will return to the prior conditional jump and will try to continue emulating the code like if the condition was not met. This gives the emulator tremendous powers.

However, we can solve that too. Instead of comparing AX with BX we can add both of them let's say to a register that holds the key for the encryption. Or we can subtract one from the other and increment the DS data segment with the amount. Normally, if the code is executed correctly, in the first case the key would be incremented with a known number and in the second case the DS will remain unchanged. BUT if the emulator fails in computing one of the two values, or even worse both, of them, the emulating process will fail altogether.

Now let's see some ways we can fool the emulator.

GETTING 2 VALUES USING INTERRUPTS AND PRIVILEGED INSTRUCTIONS

As we all know, there are certain interrupts that respond by returning a known value. I'm saying 'known' meaning you can get it somewhere else too. Let's see:

```
INT 11 - This interrupt returns the 'Equipment List'
```

The equipment list is a word that hold specific and very useful information about your computer. However, this word can also be found in BIOS at address 0000:0410h. Here goes the code:

```
XOR AX, AX
MOV DS, AX
MOV BX, WORD PTR [0410]
INT 11H
SUB AX, BX
ADD , AX
```

If the emulator skips INT 11, AX will be different from BX, so the value will be corrupted.

INT 12 - This interrupt returns 'The Total Available Memory in Paragraphs'

This word can also be found in BIOS at 0000:0413h. The anti-emu code is the same as above, but the Int value and the BIOS source.

INT 2Fh - The Multiplex interrupt

Now here we can do a lot of things. The very first and very good is this one:

```
MOV AX, 1686H
INT 2FH
```

This one returns 0 in AX if the CPU is in protected mode and something else if it's in real mode. But, we also have this instruction:

```
SMSW BX
```

This instruction (Store Machine Status Word) will put MSW in BX. The MSW (Machine Status Word) is a word with a lot of info on it. For our specific example we are only interested in the first bit. This bit is 0 if the CPU is in real mode and 1 if the CPU is in protected mode. Do you start to see a pattern here? First of all int 2fh is not emulated by many emulator around and the SMSW instruction either...

13.6 THE FPU ATTACK

USING THE FPU INSTRUCTIONS

Ok, nowadays whoever still owns a 286 or less (duh!!) is considered to be owning a pocket calculator. Whoever has a 386 kinda like enters the human kind (;-)), BUT. There's always a but. If he does not posses a FPU (Floating Point Unit) he is also considered obsolete as human ;-). In other words who doesn't have a FPU on his computer could just skip all this stuff and go watch a movie or something ;-)))

Anyway, the FPU is a very powerful thing that wonders around your CPU helping with the math calculation speed. Plus, it's 'floating' prefix gives you an idea about it's main purpose: making floating point calculations. No more only integer numbers, now you can calculate using decimals also. Is this gonna help us ? Well, I tell you: A LOT ! Why ? The first argument is this: no code analyzer / emulator I know about (except probably Dr.Web which emulates a couple of instructions) is able to emulate FPU instructions. Some of them, like TBAV hang while emulating the code. Some of them just jump over the FPU instructions, hoping they are only junk or the program is no virus at all. Actually there are very few viruses out there using the FPU instructions and the explanations for this is that people want to see their viruses spreading. The FPU instructions pose a threat: on computers not equipped with a FPU the code will hang ! In the same idea, the anti-virus products writers didn't attempt to emulate FPU instructions as 99% of the viruses in the wild don't use them. Also, as you read above about how the instructions are emulated, emulating the FPU instructions would probably triple the time the emulator needs to go through the code and, as I said the slower the emulator goes, the worse the AV product is. Combine a FPU oriented decryptor with a huge polymorph generated decryptor and the code emulator will be lost in it.

13.7 SMALL TUTORIAL ON THE FPU INSTRUCTIONS

First of all, in order not to crash the program currently running, one may want to check whether a copprocesor unit is installed. This is done easily by taking the MSW using the instruction SMSW AX, and checking the __bit. If it's set we have a copprocessor. If it's not set and your virus uses FPU in decryptors, then it's a dead cause: get out with an error or something. If you just use FPU to fool the emulators that stop over FPU instructions, just skip the part.

We shall assume that we have a computer that has an installed coprocessor (387, 487, etc...).

First, let's talk about IEEE standard 754. This is the standard Intel uses in order to make the coprocessor 'understand' floating point numbers. Basically, these numbers are coded like this:

```
S, E, F, where:

S = sign
E = exponent
F = fraction part
```

The length of the S is one bit (0 if the number is positive, and 1 if it's negative). The length of the E is calculated like this:

F has a length equal to All_bits - E_length - 1.

Let's see for example how do we code a Double Word floating point number:

```
S - 1 bit
E - 11 bits
F - 20 bits
-----------
= 32 bits
```

So, usualy the floating point number is expressed like this:

```
S, 2^E * F
```

What is very nice about the copprocessor unit is that you really don't need to put up with this crap way of storing the floating point numbers. The Fpu provides it's "stack" in order to help you out. The stack looks like this:

```
ST(0), ST(1), ... , ST(9)
```

The ST's are holders for the floating point numbers (let us understand eachother: a floating point includes an integer number; as you will see this is very important in out bussiness).

So, basically you have the loading instructions. These instructions allow you to load from a certain place a number. The number will be placed at the head of the stack and all other numbers will be pushed up. This goes like this:

```
load m : ST(0) = m ; ST(1) = 0 ; ST(2) = 0 ...
load n : ST(0) = n ; ST(1) = m ; ST(2) = 0 ...
load p : ST(0) = p ; ST(1) = n ; ST(2) = p ...
```

I hope this clears it. The most used stack register is the ST(0).
This is because we have special instructions that use other stack registers to compute as a second operator. First take a look at the FPU instructions in a very nice table I ripped of from TechHelp and then I shall explain more with some examples:

13.8 Data Transfer and Constants

```
FLD src Load real: st(0) <- src (mem32/mem64/mem80)
FILD src Load integer: st(0) <- src (mem16/mem32/mem64)
FBLD src Load BCD: st(0) <- src (mem80)

FLDZ Load zero: st(0) <- 0.0
FLD1 Load 1: st(0) <- 1.0
FLDPI Load pi: st(0) <- ã (ie, pi)
FLDL2T Load log2(10): st(0) <- log2(10)
FLDL2E Load log2(e): st(0) <- log2(e)
FLDLG2 Load log10(2): st(0) <- log10(2)
FLDLN2 Load loge(2): st(0) <- loge(2)

FST dest Store real: dest <- st(0) (mem32/mem64)
FSTP dest dest <- st(0) (mem32/mem64/mem80); pop stack
FIST dest Store integer: dest <- st(0) (mem32/mem64)
FISTP dest dest <- st(0) (mem16/mem32/mem64); pop stack
FBST dest Store BCD: dest <- st(0) (mem80)
FBSTP dest dest <- st(0) (mem80); pop stack

.---------.
| Compare |
'---------'

FCOM Compare real: Set flags as for st(0) - st(1)
FCOM op Set flags as for st(0) - op (mem32/mem64)
FCOMP op Compare st(0) to op (reg/mem); pop stack
FCOMPP Compare st(0) to st(1); pop stack twice

FICOM op Compare integer: Set flags as for st(0) - op (mem16/mem32)
FICOMP op Compare st(0) to op (mem16/mem32); pop
stack

FTST Test for zero: Compare st(0) to 0.0

FUCOM st(i) Unordered Compare: st(0) to st(i) [486]
FUCOMP st(i) Compare st(0) to st(i) and pop stack
FUCOMPP st(i) Compare st(0) to st(i) and pop stack twice

FXAM Examine: Eyeball st(0) (set condition codes)

.------------.
| Arithmetic |
'------------'

FADD Add real: st(0) <- st(0) + st(1)
FADD src st(0) <- st(0) + src (mem32/mem64)
FADD st(i),st st(i) <- st(i) + st(0)
FADDP st(i),st st(i) <- st(i) + st(0); pop stack
FIADD src Add integer: st(0) <- st(0) + src (mem16/mem32)


FSUB Subtract real: st(0) <- st(0) - st(1)
FSUB src st(0) <- st(0) - src (reg/mem)
FSUB st(i),st st(i) <- st(i) - st(0)
FSUBP st(i),st st(i) <- st(i) - st(0); pop stack
FSUBR st(i),st Subtract Reversed: st(0) <- st(i) - st(0)
FSUBRP st(i),st st(0) <- st(i) - st(0); pop stack
FISUB src Subtract integer: st(0) <- st(0) - src (mem16/mem32)
FISUBR src Subtract Rvrsd int: st(0) <- src - st(0) (mem16/mem32)
```

```
FMUL Multiply real: st(0) <- st(0) * st(1)
FMUL st(i) st(0) <- st(0) * st(i)
FMUL st(i),st st(i) <- st(0) * st(i)
FMULP st(i),st st(i) <- st(0) * st(i); pop stack
FIMUL src Multiply integer: st(0) <- st(0) * src (mem16/mem32)

FDIV Divide real: st(0) <- st(0) ÷ st(1)
FDIV st(i) st(0) <- st(0) ÷ t(i)
FDIV st(i),st st(i) <- st(0) ÷ st(i)
FDIVP st(i),st st(i) <- st(0) ÷ st(i); pop stack
FIDIV src Divide integer: st(0) <- st(0) ÷ src (mem16/mem32)
FDIVR st(i),st Divide Rvrsd real: st(0) <- st(i) ÷ st(0)
FDIVRP st(i),st st(0) <- st(i) ÷ st(0); pop stack
FIDIVR src Divide Rvrsd int: st(0) <- src ÷ st(0) (mem16/mem32)

FSQRT Square Root: st(0) <- sqrt st(0)

FSCALE Scale by power of 2: st(0) <- 2 ^ st(0)

FXTRACT Extract exponent: st(0) <- exponent of st(0); and gets pushed
st(0) <- significand of st(0)

FPREM Partial remainder: st(0) <- st(0) MOD st(1)
FPREM1 Partial Remainder (IEEE): same as FPREM, but in IEEE standard [486]

FRNDINT Round to nearest int: st(0) <- INT( st(0) ); depends on RC flag

FABS Get absolute value: st(0) <- ABS( st(0) ); removes sign
FCHS Change sign: st(0) <- -st(0)

.----------------.
| Transcendental |
'----------------'

FCOS Cosine: st(0) <- COS( st(0) )
FPTAN Partial tangent: st(0) <- TAN( st(0) )
FPATAN Partial Arctangent: st(0) <- ATAN( st(0) )
FSIN Sine: st(0) <- SIN( st(0) )
FSINCOS Sine and Cosine: st(0) <- SIN( st(0) ) and is pushed to st(1)
st(0) <- COS( st(0) )
F2XM1 Calculate (2 ^ x)-1: st(0) <- (2 ^ st(0)) - 1
FYL2X Calculate Y * log2(X): st(0) is Y; st(1) is X; this replaces st(0)
and st(1) with: st(0) * log2( st(1) )
FYL2XP1 Calculate Y * log2(X+1): st(0) is Y; st(1) is X; this replaces
st(0)
and st(1) with: st(0) * log2( st(1)+1 )
```

```
.--------------------.
| Processor Control |
'--------------------'

FINIT Initialize FPU
FSTSW AX store Status word: AX <- MSW
FSTSW dest dest <- MSW (mem16)

FLDCW src Load control word: FPU CW <- src (mem16)
FSTCW dest Store control word: dest <- FPU CW

FCLEX Clear exceptions

FSTENV dest Store environment: store status, control and tag words and
exception pointers into memory at dest
FLDENV src Load environment: load environment from memory at src
FSAVE dest Store FPU state: store FPU state into 94-bytes at dest
FRSTOR src Load FPU state: restore FPU state as saved by FSAVE

FINCSTP Increment FPU stack ptr: st(6)<-st(5); st(5)<-st(4),...,st(0)<-?
FDECSTP Decrement FPU stack ptr: st(0)<-st(1); st(1)<-st(2),...,st(7)<-?

FFREE st(i) Mark reg st(i) as unused

FNOP No operation: st(0) <- st(0)

WAIT/FWAIT Synchronize FPU & CPU:
Halt CPU until FPU finishes current opcode.
```

Along these instructions I can add here the

```
FXCH - exchange instruction st(0) <- st(1)
st(1) <- st(0)
```

which is very usefull sometimes.

So, as you saw, mainly all you should use are registers ST(0) and ST(1) because you can use the shorter form of the instruction. Let's imagine we want to compute something like this:

```
cos(((a+b)*(c+d))/f)
```

I will give you a table with the instructions and the state of the stack in the same time so you can understand:

```
fild word ptr [a] ; ST(0) = a
fild word ptr [b] ; ST(0) = b ; ST(1) = a
fadd ; ST(0) = a + b
fist word ptr [temp] ; save result
fild word ptr [c] ; ST(0) = c
fild word ptr [d] ; ST(0) = d ; ST(1) = c
fadd ; ST(0) = c + d
fild word ptr [temp] ; ST(0) = c + d ; ST(1) = a + b
fmul ; ST(0) = (a+b)*(c+d)
fild word ptr [f] ; ST(1) = f
fdiv ; ST(0) = (a+b)*(c+d)/f
fcos ; ST(0) = cos((a+b)*(c+d)/f)
```

See, it's much more easier to make calculations using the FPU. And the conversion between normal registers is done like this:

```
mov word ptr [x], ax
fild word ptr [x]
```

You will ask me why did I use FILD (load integer) instead of FLD (load float) ? Easy, that's because I didn't get the time to fully explain the IEEE 754 standard and the loading instructions expect that the source to contain a number in the IEEE 754 standard. EG, if you have at address [this_Address] the number 1111h, if you do a FILD you will have 1111h into the ST(0), but if you do FLD you will have 1.89793e-40... Kinda nasty. Plus, you must use the form FLD DWORD PTR [x] to load a floating number. Anyway, as I said, these insides are of less importance. The most important thing is to know how to use them and have a good algorithm to use them. Anyway, for those of you who want to study more on the floating point storage way, I have included alongside this article a program in ZIP form copyright by Borland International. Use it wisely... May the FPU be with you ;-)
Look there at the ways you have in order to retrieve the mantisa, the exponent and so on...

Oh, one more thing. I forgot to tell you, those who don't like to read all those .DOC files ;-) that in order to use the FPU with the TASM assembler, you need to use this kind of header for your files (actually this is the header I always use):

```
.386
.387
.model TPascal
.code
org 0
```

In this way you can safely use 32 bit registers and FPU instructions.
I can say it's the best way to compile an ASM file...

13.9 CREATING FOOLING CODE

Now, let's go down to business. We'll come back to the same good old method. We'll try to create a set of instructions that when normally executed will render to a known goal, but when emulated by a code emulator, they will generate a completely different result.

Of course, talking about math coprocessor instructions, we're gonna have to use a lot of math, but not high math, just common, ok, don't get scared.

Ok, let's take a peek at common math. As we all know, an odd number added to an even number always gives as a result an odd number (e.g. 3 + 4 = 7). What do we obtain if we divide an odd number by 4 ? Let's make a table:

```
.============================.
| X | X/4 | (X+1)/2 |
|============================|
| 1 | 0.25 | 1 |
| 3 | 0.75 | 2 |
| 5 | 1.25 | 3 |
| 7 | 1.75 | 4 |
| 9 | 2.25 | 5 |
| 11 | 2.75 | 6 |
| 13 | 3.25 | 7 |
| 15 | 3.75 | 8 |
| 17 | 4.25 | 9 |
| 19 | 4.75 | 10 |
| 21 | 5.25 | 11 |
| 23 | 5.75 | 12 |
| 25 | 6.25 | 13 |
| 27 | 6.75 | 14 |
| 29 | 7.25 | 15 |
'============================'
```

In the first column we have a series of odd numbers, in the second column we have that number divided by 4. As you can see, the decimals vary:
.25, .75, .25, etc. The last column is calculated like this (X+1)/2, where x is the number in the first column on the same line (e.g. (7+1)/2 = 8).
What do we notice ? We notice that every time a .25 appears, next to it in the third column we have an odd number and every time a .75 appears, next to it in the third column we have an even number. Ok, now that we establish some rules let's go specific:

First you must generate 2 random words. After that, be sure one of them is odd and one of them is even. To do that, for the odd number set it's first bit:

```
OR , 0001h
```

And for the even number reset it's first bit:

```
AND , 1110h
```

Now add the two numbers into . Ok, now we know that we have a random odd number in the register .

Next step, make a floating point division with 4 on this odd number and take the real part and save it somewhere (reg4 for ex.). Then take the number again, increase it by 1 and make a floating division by 2. Now, as we divided an even number, the result will be an integer. This integer might be odd or even. To check it's parity we do this:

```
Mov ax, number
Jp odd_number
Even_number:
...
Odd_number:
...
```

If the number is even we are sure that the reg4 contains 75, otherwise, if it's odd we know that the reg4 contains 25. Somewhere you should have an address that holds a double word that equals 25. If reg4 has 75, than negate the number at that address (a good way to do it in order to use more FPU instructions would be to make a floating point subtract by subtracting 25 from 0, obtaining -25). Now that we have this, simply add the double word to the number we have. The two possibilities are:

```
25 + 25 = 50
75 - 25 = 50
```

So, starting from two absolutely random numbers (which, BTW, can be DWORDS or QWORDS or whatever so you can use more FPU instructions), we obtained a fixed number, i.e. 50. Of course, this 50 number will be placed either in a ST(?) register or on a double word address. The only thing to do is crop it's end and just keep the 50 into the CL register.

Now, simply add a 6 to CL. In this way we shall have 56 in the CL register. And here comes the nice part:

```
ROL , CL
```

I hope you got it. As CL is 56 and 56 is divisible by 8, it means that the key register will roll around 7 times, but still remain the same...

Now, what do you think a code emulator will do ? Before you do all the above stuff, put a random number at the address where the 50 will be. Be sure that number is odd. If a code emulator will simply jump over the FPU instructions, as most of them do, at the end it will retract in the CL register the odd random number, which means that the ROL instruction will permanently damage the key making it impossible for the emulator to correctly decrypt the encrypted code.

This is just an idea. You can think of more. For example try dividing by 2. Any odd number divided by 2 gives a .5 decimal. Also you could obtain the 6 in the same devious manner. Let's take an example:

FLDPI ST(?) is a FPU instruction that loads the PI number (3.141592654) into the ST(?) register. Now, we all know that PI*2 = 6.283185307. Which only leaves us to take the integer part of this and we have the 6 !!

This is a simple method. We can think of more complicated ones, like:

Compute ArcTangent(1). This gives us a result equal to PI/4 (0.785398163). FMUL it by 4 and we have PI. Then SQR the number and we have PI*PI (9.869604401), and then FSUB a PI from it and we have 6.728011748. Now just remove the integer part and you'll have 6 !

I tell you, there are hundreds of methods to do that.

Of course, all the above sequences will render to some very easy recognizable signatures. Therefore, these sequences should only be used in a second level decryptor. That means that you have your virus protected by a poly generated decryptor which kills any string scan possibility, but still it cannot have enough auto-generated anti emulating and anti-debugging routines. After the poly decryptor finishes it's work, it should give control to a second decryptor. Here you can insert all the anti-debugging and anti-emulating stuff you like.

13.10CREATING VERY COMPLICATED DECRYPTORS

Well, now I'm gonna go more deep. Do you remember Taylor's formula ?
This formula hides a lot of things we can play with. Let's see. If we have a function and we want to compute the value of that function for a particular value, sometimes it's impossible to do it without Taylor's formula. However, I will use it on a not so difficult function and that is EXP(X), or e to the power of x, where e is 2.718281828...

The general formula for the Taylor series is:

```
.-------------------------------------------------------------------------
.
| (x-a)^1 (x-a)^2 (x-a)^n n |
| f(x) = f(a) + ------- * f'(a) + ------- * f''(a) + ... + ------- * f(a) |
| 1! 2! n! |
'-------------------------------------------------------------------------
'
```

,where a is a choosen constant.

A less difficult approach to this is the MacLaurin series, which is almost the same as Taylor's, with the difference that the constant a is 0.
So we have:

```
.-------------------------------------------------------------------------
.
| x^1 x^2 x^n n |
| f(x) = f(0) + ---- * f'(0) + ---- * f''(0) + ... + ---- * f(0) |
| n! 2! n! |
'-------------------------------------------------------------------------
'
```

And we know that EXP(0) = 1, which means that all the f'(x) is 1 and disappear. So the formula remains like this:

```
.-----------------------------------------.
| x^2 x^3 x^n |
| EXP(x) = 1 + x + --- + --- + ... + --- |
| 2! 3! n! |
'-----------------------------------------'
```

The problem is, how deep are we gonna go in the search for the real result of the calculation ? I mean, which should be the value of the n number ? Let's look at this example table for x=3.

```
.=================================================================.
| X^n | n | n! | X^n/n! |
|=================================================================|
| 1.00 | 0 | 1.00 | 1.00 |
| 3.00 | 1 | 1.00 | 3.00 |
| 9.00 | 2 | 2.00 | 4.50 |
| 27.00 | 3 | 6.00 | 4.50 |
| 81.00 | 4 | 24.00 | 3.38 |
| 243.00 | 5 | 120.00 | 2.03 |
| 729.00 | 6 | 720.00 | 1.01 |
| 2,187.00 | 7 | 5,040.00 | 0.43 |
| 6,561.00 | 8 | 40,320.00 | 0.16 |
| 19,683.00 | 9 | 362,880.00 | 0.05 |
| 59,049.00 | 10 | 3,628,800.00 | 0.02 |
| 177,147.00 | 11 | 39,916,800.00 | 0.00 |
| 531,441.00 | 12 | 479,001,600.00 | 0.00 |
| 1,594,323.00 | 13 | 6,227,020,800.00 | 0.00 |
|=================================================================|
| TOTAL | 20.08 |
'=================================================================='
```

As you can see, starting from n=11 we have only 0 on the last column.
This means we can safely compute only 10 steps. Now, using Taylor's formula we have computed that EXP(3) = 20.08. Now, take a calculator and calculate it. Yes, I know, it's 20.09, or 20.086, depending on the calculator. Anyway, what we are interested in is the integer part. But first, let's look at a way to compute all this:

We need:

1) A factorial routine
2) A power routine
3) A divide function
4) An add function

1) Factorial routine:

This is a somehow optimized factorial routine (doesn't takes into account N=0)

```
;we enter with CX filled with the N number and we exit with AX filled
;with N!

Factorial proc near

fild word ptr [m] ; load 1
fild word ptr [m] ; three times
fild word ptr [m]

repeat:
fmul st(1), st ; multiply by the base
fadd st, st(2) ; increase the base
loop repeat ; and repeat
fincstp ; mov ST(1) to ST(0)
fistp word ptr [m] ; store the result
mov ax, word ptr [m] ; and get it into AX
ret

m dd 1
Factorial endp
```

2) The power routine

We'll use the simple method of consecutive multiplication, as we only have 10 steps to go and the power we raise to is an integer number. The procedure will raise AX to the power CX:

```
Power Proc Near
mov word ptr [m], ax
fild word ptr [m]
fild word ptr [m]
dec cx

repeat:
fmul st, st(1)
loop repeat
fistp word ptr [m]
mov ax, word ptr [m]
ret

m dd 1
Power Endp
```

Of course, the above procedures do not handle the exceptions (like 0!, or x^0). For the complete program, look at the TAYLOR.ASM file included in this tutorial.

And here comes the fun part :

```
ADD , AX
```

So, AX means 20, if the CPU/FPU executed all the instructions as said above, the register that holds the key should increase with 20. If the debugger or code emulator didn't compute correctly one of the instructions than the key we'll be added a random number, killing the decryption process completely (of course don't forget to set AX with some big random number before running the Taylor procedure).

Here is, however one of my favorite decryptors I have ever think of. It's main background is the propriety of three numbers, known as Phytagora's numbers. These numbers (a, b, c) verify the following formula:

```
a^2 = b^2 + c^2
```

Now, all you have to do is find 3 numbers that meet this propriety, like for example a=5, b=4, c=3. In order to do that, you must choose two random numbers (let's call them m and n) and apply the following formulas:

```
a = m*m + n*n
b = 2*m*n
c = |m*m - n*n|
```

The main propriety of the Pythagora's numbers is that if they are used as a triangle's sides, then the angle against the a side will always be 90ø:

```
|\
b | \ a
| \
|_________\
c
```

Therefore, given the fact that one triangle's angles summed give a total of 180ø, we can say that angles B and C summed give 90ø (where B is the angle made by a and b, and C is the angle made by a and c).

We also know how to compute these angles, as:

```
cos(B) = c / a ==> B = arccos(c/a) (1)
cos(C) = b / a ==> C = arccos(b/a) (2)
sin(B) = b / a ==> B = arcsin(b/a) (3)
sin(C) = c / a ==> C = arcsin(c/a) (4)

and B + C = 90ø, which leads us to our main formula:

cos(B + C) = cos(90ø) = 1

and for cos(B+C) we have the following formula:

cos(B+C) = cos(B) * cos(C) + sin(B) * sin(C) = 1, so,

using (1), (2), (3) and (4), we have that:

cos(B+C)=cos(arccos(c/a))*cos(arccos(b/a))+sin(arcsin(b/a))*sin(arcsin(c/a)
)

= 1 (aprox.) ==>
```

we must round cos(B+C) in order to have 1.

So, we choosed 2 random number (which, BTW don't have to be integers, they also may be floating point numbers) which led us to 3 other numbers that meet Pythagora's propriety and using the last formula we are sure we'll obtain a result that equals 1.

As you can see, we are forced to use the ArcSin and ArcCos functions. Unfortunately, the FPU doesn't have these functions. However, it has the FATAN function, which computes the ArcTangent. In order to obtain the arcsin and arccos we can use the following formulas:

```
ArcSin = ArcTan(x/sqrt(1-sqr(x)));
ArcCos = ArcTan(sqrt(1-sqr(x))/x);
```

Let me take a brief example:

```
m = 1
n = 2

a = 1*1 + 2*2 = 1 + 4 = 5
b = 2*1*2 = 4
c = |1*1 - 2*2| = |1 - 3| = |-3| = 3

verification: a^2 = 5*5 = 25
b^2+c^2 = 4*4 + 3*3 = 16 + 9 = 25
```

Now let's compute angles:

```
B = arccos(c/a) = arccos(3/5) = 0.927295218
C = arccos(b/a) = arccos(4/5) = 0.6435011879
B = arcsin(b/a) = arcsin(4/5) = 0.927295218
C = arcsin(c/a) = arcsin(3/5) = 0.6435011879
```

These have been computed using the ArcTan formula presented above.

```
cos(arccos(3/5)) = cos(0.927295218) = 0.9998690361902
cos(arccos(4/5)) = cos(0.6435011879) = 0.9999369305892
sin(arcsin(4/5)) = sin(0.927295218) = 0.0161836481643
sin(arcsin(3/5)) = sin(0.6435011879) = 0.0112309769722
```

so:

```
cos(B+C) = 0.9998690361902 * 0.9999369305892 +
0.0112309769722 * 0.0161836481643 =
.----------------.
= 0.999805975039 + 0.000181758179 = | 0.999987733218 |
'----------------'
==> round(cos(B+C)) = 1 (bingo ! ;-))
```

As I said, the m and n numbers may be floating point numbers which will lead to floating point a, b, c's... Much nicer to handle them.

Let's see which FPU instruction do we need:

```
- FMUL
- FSIN
- FCOS
- FATAN
- FDIV
- FADD
- FROUND
- FSQR
- FSQRT
- FSUB
```

I would say rather plenty (not counting the loading and storing instructions...). Taking into consideration the quickness of the FPU, the above formula is completed very quickly. I want to see an emulator emulating it !

What do we do with the 1 we obtained ? We can use it to increase the pointer in the code to be decrypted, we can use it to increase the encryption key, or anything we can think of.

Included alongside this article you have a demonstration of the above calculations in the PYT.ASM file.

Also, both methods are used in the CTAYLOR.ASM and CPYT.ASM files which have the purpose to demonstrate the way to use the two methods presented which I called the FPU.Taylor.Crypt and FPU.Pythagoras.Crypt. Basically the programs will display a text on the screen, then it will display it scrambled and then unscrambled again. You can see the speed of the procedures there. I doubt that there exist any code emulator written yet to emulate that code !

13.11CREATING SELF MODIFYING CODE

Another nice way to use FPU instructions is to create self modifying code. Basically this is done like this:

1) make a FPU calculus with a known result
2) store the result on the following dword

For example, we have know how to obtain the number 00001234h. That's 17185 - 12525, for example.

We'll make this:

```
mov al, 13h
mov si, 0
lea bx, patch
add bx, 4
finit
fild word ptr [b]
fild word ptr [a]
fsub ; ST(0)
fist dword ptr [patch]
patch:
sub ax, 14h
nop
nop
js patch
...
a dw 12525
b dw 17185
```

In the moment the integer number is stored over the 'patch' address, the instruction sub ax, 14h changes to:

```
xor al, 12h
add [bx+si], al
```

This means that after the XOR Al will turn to 1.

[bx+si] points to patch+4. By doing the Add [bx+si], the two NOPS will change into Xchg ax, cx. This instruction will put 1 into CX. Furthure, you can use the number 1 in CX in your code. If a code emulator skips the FPU instructions, the whole code goes to hell... This is because sub instruction will get executed and AX will be signed a reeeeeeally long time, which leads us into a very long loop with a conditional jump. This particular kind of jump kills many code emulators which pretend to return to the place where the condition happened and go on with the code... But what do you do when the code goes infinite ?

13.12TIPS & TRICKS

Ok, everybody sometimes thinks that he discovered something marvelous. He is so happy... until he finds out that someone else discovered the same thing like a few years ago... ;-) It doesn't mean you are an illiterate, but, you just didn't read that particular book... Well, this happened to me to. I thought I found out something really neat, but it seems that another guy, a great coder named .... made this up way before I even thought about FPU's. It's called 'moving memory using FPU'. The message about this showed up on my virus mailing-list and I give full credit to it's author, but still I will present it here as I think it's a great idea.

So, the basic beyound this is that we have a load function in the FPU and a store function too. So:

```
; make DS:ESI point to the source code
; make ES:EDI point to the destination code
; ECX = length of code to be moved
; the code length is calculated in 16 bytes chunks

mov_loop:
fild qword ptr [esi]
fild qword ptr [esi+8]
fxch
fistp qword ptr es:[edi]
fistp qword ptr es:[edi+8]
add edi, 16
add esi, 16
sub ecx, 16
jns mov_loop
```

So, this procedure moves memory very quickly and is undetectable for now by any AV or code-emulator. Hope you will use it smartly...

These would be some thoughts and ideas about how you can play with the FPU instructions, but I repeat, there are thousands and thousands of ways to do it. And, as I said, almost no emulator or real debugger can break it. If you can, you should use more than one method just to be sure, because however some AV's started emulating a couple of instructions.

13.13THE ENVELOPE OF THE MATRIX METHOD

I called this method in this way, exactly because we are about to use a matrix in order to obtain our encryption. Ok, so the usual algorithm for encryption just takes one by one bytes or words or dwords or whatever and applies an math operation over them and then stores the result. This is a linear encryption which can be broke very easy by a good programer. However, if we are creating a devious, hard to understand when coded encryption method, we got big chances. So, let's start. Let's say we have to encrypt a part of a file that looks like this:

```
a1, a2, a3, ... , an

where ak are the encryption unit (byte, word, dword, qword, tbyte,..)
```

We than we'll take the first 25 units and arange them in a sqare matrix like this:

```
a11 a12 a13 a14 a15
a21 a22 a23 a24 a25
a31 a32 a33 a34 a35
a41 a42 a43 a44 a45
a51 a52 a53 a54 a55
```

Ok, now let's define what is 'giving a roll to the matrix'. Imagine that the above matrix is a piece of paper. A sqare. And you want to fold it over the first diagonal. You would obtain this result:

```
a _________ _________
| /| | /
| / | ------> | /
| / | | /
| / | | / (we took corner b over corner a)
|/________| |/
b
```

Now, the same thing is what we shall do with our matrix above. We shall take each value from beneath the first diagonal and bring it over the opossite value. We'll do this by applying a math formula. First we are gonna apply an 'ADD-ROLL', which means that each element beneath the first diagonal will be added to it's pair above the diagonal. Let's see what do we get:

```
a11+a55 a12+a45 a13+a35 a14+a25 a15
a21+a54 a22+a44 a23+a34 a24 a25 (FD - ADD-ROLL)
a31+a53 a32+a43 a33 a34 a35 (First Diagonal Add Roll)
a41+a52 a42 a43 a44 a45
a51 a52 a53 a54 a55
```

So, I think it's clear enough. All elements above the first diagonal were added the elements beneth the first diagonal. In the second step we shall apply a SD-SUB-ROLL, which means that we are going to take the left-down corner and put it over the right-up corner and the math operation between the elements will be substract. I'm not drawing another matrix because I hope it's clear. Then we are going to apply a H-XOR-ROLL (horizontal xor roll), which means that we are taking all elements beneath the horizontal middle line of the matrix and xor them over their opossite elements above the horizontal line.
Finally we apply a V-ADD-ROLL (vertical add roll), which means we add every element from the left side of the vertical center of the matrix to their opossite elements on the right side.

After all these are done, we can say that our initial matrix is pretty messed up. Let's call the final scrambled matrix A, and define it
like this:

```
A11 A12 A13 A14 A15
: :
A51 ..... A55
```

So, the final formulas after applying the above rollings are:

```
Encryption formulas (I noted the XOR operation with '|'):

A11 = (a11+a55)|a51
A12 = (a21+a45-a21-a54)|a52
A13 = (a13+a35-a31-a53)|a53
A14 = (a14+a25-a41-a52)|a54 + (a12+a45-a21-a54)|a52
A15 = (-a51-a15)|a55 + (a11+a55)|a51
A21 = (a21+a54)|(a41+a52)
A22 = (a22+a44)|a42
A23 = (a23+a34-a32-a43)|a43
A24 = (-a24-a42)|a44 + (a22+a44)|a42
A25 = (-a25-a52)|(-a45-a54) + (a21+a54)|(a41+a52)
A31 = a31+a53
A32 = a32+a43
A33 = a33
A34 = -a34-a43+a32+a43
A35 = -a35-a53+a31+a53
A41 = a41+a52
A42 = a42
A43 = a43
A44 = a44+a42
A45 = -a45-a54+a41+a52
A51 = a51
A52 = a52
A53 = a53
A54 = a54+a52
A55 = a55+a51
```

So, now we have our scrambled matrix. Of course, as you can see there still are there a couple of codes that didn't get encrypted. No problem !
So, we have 25 elements. Let's see:

if the a's are bytes we have 8*25 = 200 bits
if the a's are words we have 16*25 = 400 bits

Anyway, the total number of bits is divisible by ten. Now here is the thing. You should create a 10 bit long key. Why ? Because this is most unusual. Put the first 8 bits in the register Al, for ex., and the other 2 bits in register Bl, like this:

```
aaaaaaaabb000000
| al || bl |
```

Now, we put our scrambled matrix like this:

```
A11, A12, ... , A21, A22, ... , A55
```

And we look at it at the bit level.
First apply a XOR over the beginning of the elements. Then increase the key like this:

```
000000aaaaaaaabb
| al || bl |
```

This is easily done using the shifting with carry. Then increase the pointer with one byte and apply again. It will be like this:

```
bits to scramble: xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx ...
key: : aaaaaaaa bb000000 aaaaaaaa bb000000
000000aa aaaaaabb 000000aa aaaaaabb ...
```

I hope you get it. You scramble at bit level with a ten letter key, with an interleaved algorithm, leaving four unscrambled bits every 12 bits. I would say it's a rather peculiar cryption. The decryption is very easy. The only nifty thing is that if someone sees the code that does the above thing disassembled he will have to work his butt of a few days to figure out the encryption.

OK ! Now we have our matrix completely encrypted and placed in the encrypted code place. We return to the unencrypted code. We firstly took 25 units. Now we must take another 25 units and continue with the algorithm. And so we do, until we don't have 25 units, so we create a matrix with the last elements padded with zero. And there you have the Envelope of the matrix cryption method.

You will ask, well how the hell do I decrypt it ? Thought I'd leave you here ? ;-)

So, first, when decrypting you must start again by retriving 25 units from the crypted code. Then you decrypt the 10 bit key encryption and you have the originall A11...A55 matrix. Here are the formulas to decrypt the matrix in order to obtain the original a11, a12,..., a15 elements. And no, they are not in a random order. They are in the exact order in which you CAN decrypt them ! Here we go:

```
a11 = A33
a42 = A42
a43 = A43
a51 = A51
a52 = A52
a53 = A53
a54 = A54 - a52
a55 = A55 - a51
a44 = A44 - a42
a41 = A41 - a52
-a45 = A45 + a54-a41-a52
a31 = A31 - a53
-a35 = A35 + a53-a31-a53
a32 = A32 - a43
-a34 = A34 + a43-a32-a43
a11 = A11|a51 - a55
a21 = A21|(a41+a52)-a54
a22 = A22|a42 - a44
a23 = A23|a43 - a34+a32+a43
-a24 = (A24 - (a22+a44)|a42)|a44 + a42
a12 = A12|a52 - a45+a21+a45
a13 = A13|a53 - a35+a31+a53
-a15 = (A15 - (a11+a55)|a51)|a55 + a51
a14 = (A14 - (a12+a45-a21-a54)|a52)|a54 - a25+a41+a52
-a25 = (A25 - (a21+a54)|(a41+a52))|(-a45-a54) + a52

You must negate a45, a35, a34, a24, a15 and a25.
```

That's it ! Applying these formulas you have the original state of the matrix which you put in it's decrypted string place unit after unit.

And as we spoke so much about FPU, I think I don't have to mention that all the above calculations may be done using the faster fpu instructions. I doubt any code emulator will be able to 'violate your mail' by ripping your envelopes ;-)))

I sure hope I'll get the time to write an example on this.
Unfortunately, I didn't...

13.14P-MODE ATTACK

What I am going to tell you now is something I am not quite sure that it will work on all systems, but if you read it, you may try it at least. The main ideea is this: swith to P-Mode using a strange manner, do sumthing and then switch back to real-mode... A code emulator will be dead by then...

Here is the main thing we are interested in:

Some multiplex interrupts.
First, the interrupt that tells us if we have DPMI present and if so, what is the address we need in order to switch to it. It goes like this:

```
Expects: AX = 1687h
----------------------------------------------------------------------
Returns: AX 0000h = successful
else = no DPMI host present
BX flags: bit 0: 0=32-bit programs are not supported
1=32-bit programs are supported
bits 1-15: not used
CL processor type: 02H = 80286
03H = 80386
04H = 80486
05H = Pentium
>5 = reserved for future Intel CPUs
DX DPMI major + minor version number (e.g., 010aH=1.10)
SI number of 16-byte pares needed for DPMI host private
ES:DI entry address to call to enter Protected Mode
----------------------------------------------------------------------
```

SI on return, this is an amount of real-mode memory, in 16-byte paragraphs, that you must supply when you process the switch (see below).
It might be 0000H, indicating no memory needed.

ES:DI on return, this is the Entry Address you must call (via a FAR CALL) in order to switch to protected mode. The calling parameters are:

```
Entry:
AX= 0000H = you'll be running as a 16-bit application
0001H = you'll be running as a 32-bit application
ES= the segment of the memory you're be supplying to DPMI host.
If SI was 0 after INT 2fH 1687H, then ES is ignored.
Return:
CF set (CY) if switch to protected mode failed
(and AX is a DPMI Error Code)
CS = selector for your code segment (64K limit)
SS = selector for your stack segment (64K limit)
DS = selector for your data segment (64K limit)
ES = selector for your program's PSP (256-byte limit)
FS = 0 (on 80386+ CPUs)
```

There's no need to flush here the DPMI error codes... It's either you can or you cannot enter PMODE. Let's check a little program that should (or at least I hope) go into PMODE and back:

```
start:
mov ax, 1687h ; We call the multiplex int
int 2fh ;
cmp ax, 0 ; do we have DPMI ?
jne no_dpmi ; no...
mov switchcall, di ; if so, save the switch call address
mov switchcall+2, es ; offset and segment
cmp si, 0 ; check if we need memory in 16 byte chunks
je no_mem ; no...
mov bx, si ; otherwise allocate memory
mov ah, 48h ; using DOS
int 21h ;
jc error ; if this occurs you have no memory...
; so you might need to shrink mem using 4Ah first...
no_mem:
mov es, ax ; put the new segment in ES
mov ax, 0 ; choose 16 bit application
call switchcall ; and switch to PMODE
jc cantswitch ; error ?
mov ax, 0400h ; try to use PMODE interrupt 31h
int 31h ;
;
mov ax, 4c00h ; switch back to REAL mode
int 21h ;
;
no_dpmi: ;
cantswitch: ;
mov ax, 4c00h ; and quit
int 21h ;
;
switchcall dw 0, 0 ; call switch address
```

So, as you could see, we have the interrupt 31h we can use. In order to use it, you must have a real grip on what a selector, descriptor, etc. is, so better check a DOS32 documentation. The usefull functions are these:

```
AX Function Use
--------------------------------------
0000H (allocate LDT descriptors)
0001H (free LDT descriptor)
0002H (segment to descriptor)
0003H (query selector increment value)
0006H (query segment base address)
0007H (set segment base address)
0008H (set segment limit)
0009H (set descriptor access rights)
000aH (create alias descriptor)
000bH (query descriptor)
000cH (set descriptor)
000dH (allocate specific descriptor)
000eH (query multiple descriptors)
000fH (set multiple descriptors)
0100H (allocate DOS memory block)
0101H (free DOS memory block)
0102H (resize DOS memory block)
0200H (query real-mode interrupt vector)
0201H (set real-mode interrupt vector)
0202H (query processor exception handler vector)
0203H (set processor exception handler vector)
0204H (query protected-mode interrupt vector)
0205H (set protected-mode interrupt vector)
0300H (simulate real-mode interrupt)
0301H (call real-mode for FAR RET return)
0302H (call real-mode for IRET return)
0303H (allocate real-mode callback address)
0304H (free real-mode callback address)
0305H (query state save/restore addresses)
0306H (query raw mode switch address)
0400H (query DPMI version)
0401H (query DPMI capabilities)
0500H (query free memory information)
0501H (allocate memory block)
0502H (free memory block)
0503H (resize memory block)
0504H (allocate linear memory block)
0506H (query page attributes)
0507H (set page attributes)
0508H (map device in memory block)
0509H (map conventional memory in memory block)
050aH (query memory block size and base)
050bH (query memory information)
0600H (lock linear region)
0601H (unlock linear region)
0602H (mark real-mode region as pageable)
0603H (relock real-mode region)
0604H (get page size)
0700H (mark page as demand paging candidate)
0701H (discard page contents)
0800H (physical address mapping)
0801H (free physical address mapping)
0900H (disable virtual interrupt state)
0901H (enable virtual interrupt state)
0a00H (query vendor-specific API entry address)
0b00H (set debug watchpoint)
```

```
0b01H (clear debug watchpoint)
0b02H (query state of debug watchpoint)
0b03H (reset debug watchpoint)
0c00H (setup DPMI TSR callback)
0c01H (protected-mode terminate and stay resident)
0d00H (allocate shared memory)
0d01H (free shared memory)
0d02H (serialize on shared memory)
0d03H (free serialization on shared memory)
0e00H (query coprocessor status)
0e01H (set coprocessor emulation)
```

As you can read in the descriptions, quite a few interesting things out there. But, as I said, I don't have time to write on this anymore, so, just go ahead and try using some of the above functions. I think it would bea really neat to allocate DOS memory from protected mode and then return to real mode and use it... although I didn't try it ;-)).

13.15Final Word

So, as I said, I left this behind me now and I am going towards Win32 programing. In order to do that I spent a lot of time studying, I had to read my butt out and gather utilities and tutorials and tools and everything, so I kinda left this away... So, I guess my next article will be on Win95/98...

Write me anytime with suggestions, ideas or anything at:

lordjulus@geocities.com

From time to time check my page at:

http://members.tripod.com/~lordjulus

If you are interested in virii news and info, you may try to join my
virus list by sending a blank e-mail to:

virus-list-subscribe@makelist.com

All the Best !

Lord Julus - 1998 (c)

I would like to thank the following: Qark, Quantum, RockSteady, DarkAngel, Hellraiser, MrSandman, Darkman, VirtualDaemon, JackyQwerty, Azrael, B0z0, Neurobasher, NowhereMan, TheUnforgiven, LiquidJesus, a.s.o...
Lord Julus

14 Freestyle articles

14.1 Coding Smart And Dynamic Code (For better protections and the art of it!) – by The+Q

14.1.1 Introduction

Running code is beautiful, isn't it? Small islands of code floating in a vast sea of data. Those code islands must run in a strict, pre-defined way in order to work. Normally you would have dozens of files in memory, and even more threads, running on a single CPU. The outcome can be a cool 3D FPS, a MP3 player or even a word processor. The sheer complexity is mind blowing. Think what happens when you break into Softice. Its like freezing rain drops in mid air, while its raining on a vast, green, rain forest. Now imagine you could not only stop those rain drops, but also make them go upwards! Imagine you could create thunder and lightning at your will...

Its possible! In this article we are going to explore both old and new techniques of code manipulation. Dynamic code has a great value to both low level programming and protections alike. Here I wish to show you how it can be done.

14.1.2 Pre Build Power - macros

Where would we be without them? Hash functions would take years to write, code would take an un-usual amount of space, and the term being lazy would not reach the vast corners it reaches now.

here's something very elegant in macros. Take a look at this example:

```
#define EXP8(X) EXP7(X), EXP7(128+X)
#define EXP7(X) EXP6(X), EXP6( 64+X)
#define EXP6(X) EXP5(X), EXP5( 32+X)
#define EXP5(X) EXP4(X), EXP4( 16+X)
#define EXP4(X) EXP3(X), EXP3(  8+X)
#define EXP3(X) EXP2(X), EXP2(  4+X)
#define EXP2(X) EXP1(X), EXP1(  2+X)
#define EXP1(X) EXP0(X), EXP0(  1+X)
#define EXP0(X) ((4*(X)+1)*((X)+1) & 0xff)

static unsigned char hash_table[256] = { EXP8(0) };
```

hash_table is automatically filled by the pre-compiler with all byte values 0-255 in a permuted order.

Macros are well implemented in the C++ environment, but they are much more powerful in *MASM* environment. Just peek at MASM's reference at the keywords as **MACRO**, **MACRO LOCAL**, **FOR** (**IRP**) and **FORC** (**IRPC**). Amazingly, MASM's pre-compiler lets you define local variables inside the macro, just like in a normal function. Logical and mathematical operations on the variables are supported, and not only that, but you can also work on each byte from the argument separately! Macros let you define data and function names dynamically, and you can even define macros inside macros. Macros open the door to any cool thing you can think of =)

With macros alone we can write *dynamic data* - a data array, that will be automatically encrypted by the pre-compiler. Lets see what we need to do in order to achieve it:

Define a macro that takes an entire string as argument

Go through each byte of the string

Encrypt it

And dump the result as data byte

```
EncryptText MACRO text:req
        LOCAL cipherByte
        cipherByte = 0AAh

        FORC plainByte, <&text>

          cipherByte = cipherByte xor '&plainByte'

          db cipherByte

        ENDM
        db 0
ENDM
```

Now in our source code the data is defined with the macro:

```
EncryptText < Sin without deceivers/A God with no believers >
```

And on the compiled version this data is already encrypted. Of course in run time we'll have to decrypt it back to plaintext, but half the job is done by the pre-compiler. Macros are very efficient data manipulators.

14.1.3 Post Build Power - .map file and the external patcher

Dynamic code is a bit more problematic. The problem is that it was never meant to be dynamic! The fact that the code is marked as a read only section is a proof of that =) The more serious problem is that we don't know the *final code* bytes before we compile the code. Even if we write it in assembly. There's a lot of fine tuning done by the compiler and linker to make the code workable. Think of function's stack prologue and epilogue, or calling an API function, or even assembling a jump instruction. The bottom line is that the final shape and form of the code is known only after the build.

Fortunately, the linker provides all the information of the final code in the *.map* file. Lets consider the following problem: code integrity check. If we are going to implement an integrity check on the code, we'll need the *start address* and *end address* of the block to check. We'll also need the *correct* CRC value to check against. Now where are we going to get all these values? Lets consider start and end addresses of the code block. We can try to get them during pre-compile time. For example, if we have a project with only one source file to compile, we'll also have one .obj file, which means that the order of the functions in the source code will also appear in the final .exe file. CRCing one function is also a simple deal - just put **`BlockStartPtr:`** above the function, and **`BlockEndPtr:`** just below it, and send them to the CRC function. BUT, what are we going to do when we have more than one source file? or if we want to check sum the entire application? and how are we going to get the correct CRC value, and store it back in the code, anyway?

The remedy is a new tool we'll build specifically designed for our work. An *external patcher* program that will read the application, manipulate or checksum whatever code we want and even send post-build information back to the application. Its all in our hands now!

The external patcher needs to communicate with the application, and for this we'll write a special structure:

```
in asm:
        EP_INTERFACE_MAGIC              equ     49494949h

        EP_TARGET_INTERFACE STRUCT
                _interfaceFlag   DWORD EP_INTERFACE_MAGIC
                _crcBlockStart   DWORD ?
                _crcBlockEnd     DWORD ?
                _crcCorrectValue DWORD ?
        EP_TARGET_INTERFACE ENDS

or in C++:
        #define EP_INTERFACE_MAGIC              (0x49494949)

        #pragma pack(push, 1) // no data structure alignment
        struct EP_TARGET_INTERFACE
        {
                DWORD           _interfaceFlag;
                DWORD           _crcBlockStart;
                DWORD           _crcBlockEnd;
                DWORD           _crcCorrectValue;
        };
        #pragma pack(pop)
```

Please notice the **`_interfaceFlag`** dword. The external patcher needs to locate the structure in the file, so this flag, initialized with an un-common magic dword, marks the start of the interface. To accomplish code integrity check, the external patcher grabs both the application's .exe file and .map file. From the .map file our patcher calculates the application's image start and end addresses - it can be the entire image, (the entire .text section) or it can be a specific function. Next the external patcher opens the .exe file, and locates the interface with the aid of the flag dword. The external patcher performs the check sum of the specified code block, and stores both addresses and the correct result back to the exe, in the interface. At run time, the application grabs these values, which are now valid, and performs the integrity check! Please note that the addresses should be converted from RVA, and proper error handling should be implemented, but the basic idea is that simple to implement with the aid of the external patcher.

14.1.4 Combining powers – Dynamic Code

Why stop here? Lets work with both pre-build and post-build powers together! Not always when you add two great things, you get an even greater combination. I mean, if you play Beethoven and Mozart together you usually get a classical noise. But not so in our case. Combining macros and post build patcher is a beautiful and amazingly powerful technique.

As an example we will write a function that automatically locks and unlocks itself. Lets see what we need:

1. The code block should be unlocked at runtime. So we'll have a loop that encrypts/decrypts the code. The decryptor will execute at the function's prologue, while the encryptor will execute at the function's epilogue.
2. The code block should be locked in the final .exe file. This means the external patcher will have to lock the block after compilation. Which also means we'll have a flag at the block's first opcode, so the external patcher will be able to locate the block.

Lets start with the macro:

```
AUTO_LOCK_BEGIN MACRO
        LOCAL blockStart, blockEnd, lockLoop

        ;; init
        pusha
        mov esi, offset blockStart
        mov edi, esi
        mov ecx, (offset blockEnd - offset blockStart)

        ;; decrypt
lockLoop:
        lodsb
        xor al, 55h
        stosb
        loop lockLoop

        ;; goto the beginning of the block
        popa
        jmp blockStart

        ;; information for the external patcher:
        dd EP_AUTO_LOCK_MAGIC                   ; flag
        dd (offset blockEnd - offset blockStart)   ; block size

        blockStart LABEL BYTE

        AUTO_LOCK_END MACRO
                blockEnd LABEL BYTE

                ;; init
                pusha
                mov esi, offset blockStart
                mov edi, esi
                mov ecx, (offset blockEnd - offset blockStart)

                ;; encrypt
        lockLoop:
                lodsb
                xor al, 55h
                stosb
                loop lockLoop

                ;; done
                popa
        ENDM
ENDM
```

Please notice how **AUTO_LOCK_END** macro is defined inside **AUTO_LOCK_BEGIN** macro. This means that **AUTO_LOCK_END** can use all the variables declared in the parent macro. This is how these macros are used:

```
ReallyCoolFunctionThatDoesNothingButBragItsProtection PROC

        AUTO_LOCK_BEGIN

        ;; write what ever we want here... guess what this does ;)
        mov     esi,edx
        xor     cl,cl
        shld    edx,eax,1
        shld    esi,eax,2
        adc     cl,0
        xor     edx,esi
        xchg    eax,edx
        xor     dl,cl

        AUTO_LOCK_END
        ret
ReallyCoolFunctionThatDoesNothingButBragItsProtection ENDP
```

The external patcher's job is to locate this code block, use the given block size information, and encrypt the code. If there are more than one block in the file, there will simply be more flags. The external patcher will go through the entire file, and locate all these flags.

Here's a cool optimization to the macro above, that will use the same decryption loop for both encryption and decryption:

```
AUTO_LOCK_BEGIN MACRO
        LOCAL blockStart, blockEnd, lockLoop, lockMain, retJunction

        ;; init
lockMain:
        pusha
        mov esi, offset blockStart
        mov edi, esi
        mov ecx, (offset blockEnd - offset blockStart)

        ;; decrypt
lockLoop:
        lodsb
        xor al, 55h
        stosb
        loop lockLoop

        ;; goto the beginning of the block
        popa

        ;; use SMC to toggle RET / JMP instructions
        xor byte ptr [retJunction], 028h

        ;; sometimes its jump, sometimes ret...
retJunction:
        db 0C3h, 8

        ;; information for the external patcher:
        dd EP_AUTO_LOCK_MAGIC                       ; flag
        dd (offset blockEnd - offset blockStart)    ; block size

        blockStart LABEL BYTE

        AUTO_LOCK_END MACRO
                blockEnd LABEL BYTE

                ;; encrypt
                call lockMain
        ENDM
ENDM
```

14.1.5 Run Time Power - "The Running Line"

In an article titled "Anti Debugging Tricks" posted back in 1994 a technique called "The Running Line" was described (apparently, it was first published by Serge Pachkovksy, but i couldn't find it). This technique reveals a self tracing, self modifying code. The idea is really beautiful. Basically, there's a tracer function that's called after every "normal" instruction. On this tracer function, you could change the code dynamically at run-time and return to it. In practice, exception handling and the trap flag are used. The CPU raises the "single step" exception whenever the trap flag is set. Under DOS this means that int1 is raised. A self-tracing program would take advantage of this feature, and simply by changing the int1 handler and setting the trap flag, you would have a self-tracing application.

Fortunately, this technique is also applicable under windows. Only this time there's no int1, but a normal exception, and a normal exception handler. This technique is extremely efficient against debuggers and dis-assemblers. SoftIce messes the trap flag, so you cant single step into a self-tracing function, and under a disassembler a naive code block might change in run time, thanks to the running line handler. Here's a simple tracer that changes one of the code's instructions:

```
        ;; SEH macros
        SEH_NODE STRUCT
                _prevHandler     DWORD ?
                _exceptionHandler         DWORD ?
        SEH_NODE ENDS

        PUSH_SEH MACRO sehHandler:req
                ASSUME FS:NOTHING
                mov eax, fs:[0]
                ASSUME eax:ptr SEH_NODE

                push sehHandler
                push [eax]._exceptionHandler
                mov fs:[0], esp
        ENDM

        POP_SEH MACRO
                pop fs:[0]
                add esp, 4
        ENDM

        ;; exception handler
        ExpHandler PROC c expRecord:DWORD, expFrame:DWORD,
contextPtr:DWORD, dispContext:DWORD
                pusha
                mov ebx, contextPtr
                ASSUME ebx:ptr CONTEXT

                ;; clear trap flag
                and [ebx].regFlag, 0FFFFFEFFh

                ;; change the opcode to NOP
                mov ebx, [ebx].regEip
                mov byte ptr [ebx], 90h

                popa
                mov eax, ExceptionContinueExecution
                ret
        ExpHandler ENDP
```

```
;; Self tracing function
SelfTracingCode PROC

        ;; set up the handler
        PUSH_SEH < offset ExpHandler >

        ;; set the trap flag
        pushf
        or byte ptr [esp+1], 1
        popf

        ;; this will not be traced
        xor eax, eax
        ;; endless loop - this code will change at run-time
        JMP $

        ;; remove the handler
        POP_SEH

        ret
SelfTracingCode ENDP
```

The possibilities of the running line technique are endless. The tracer is like an invisible storm that can change the code at run time. Self modifying, self decompressing and even self compiling code is no fiction.. it can be done =)

14.1.6 This is only the beginning...

You can probably guess what im going to say... "Why stop here?" Good question! :) Lets combine all the above techniques! The final example we are going to explore is one funky chicken. We are going to write code that *runs backwards*.

In theory its very simple, we'll use the running line handler to re-set the **EIP** register one instruction upwards, and then restore execution for one more step. When this is done on a block of opcodes, we'll have a whole block that actually runs backwards. In practice, if we want to re-set the **EIP** register one instruction upwards we'll have to know the size of the opcode, and decrease this value from **EIP** register. Getting the opcode size is simple:

```
opcodeStart:
          xor eax, eax
          db ($ - offset opcodeStart)
```

And there, we have the opcode, followed by its size. Doing this for all the opcodes in the block can be a real pain, so we'll do it in a macro:

```
OPREVBEGIN MACRO
          LOCAL opcodeStart, opcodeEnd

          ;; dump the size
          db (offset opcodeEnd - offset opcodeStart)

          opcodeStart LABEL BYTE

          OPREVEND MACRO
                    opcodeEnd LABEL BYTE
          ENDM
ENDM

;; wrap the opcode with the macro above
R MACRO opcode:req
          OPREVBEGIN
          opcode
          OPREVEND
ENDM

;; a block of opcodes, each one has its size attached
R< rol eax, 10 >
R< mov ecx, "hi, " >
R< xor edx, ecx >
R< add eax, "man!" >
R< stosd >
```

Next we'll have to make things simpler to the external patcher. We should supply the block's beginning address, and its total size. While we are at it, we'll make things simpler to the running line handler:

```
REV_BLOCK_BEGIN MACRO
          LOCAL blockStart, blockEnd

          ;; set up exception handler to RevHandler function
          PUSH_SEH < offset RevHandler >

          ;; make sure ebp points to opcodes sizes array
          mov ebp, (offset blockStart)

          ;; set trap flag
          pushf
          or byte ptr [esp+1], 1
          popf

          ;; this runs backwards, so jump to the end =)
          jmp blockEnd

          ;; information for the external patcher:
          dd EP_REVERSECODE_MAGIC                         ; flag
          dd (offset blockEnd - offset blockStart)        ; block size

          blockStart LABEL BYTE

          REV_BLOCK_END MACRO
                    blockEnd LABEL BYTE

                    ;; remove exception handler
                    POP_SEH
          ENDM
ENDM
```

In the source code, this is how a block that runs backwards is implemented:

```
CoolFunctionThatRunsBackwards PROC
          ;; mark the beginning
          REV_BLOCK_BEGIN

          ;; every opcode here should have its size attached
          R< ror eax, 10 >
          R< mov ecx, "what" >
          R< xor edx, ecx >
          R< add eax, "s up" >
          R< nop >

          ;; mark the end
          REV_BLOCK_END
          ret
CoolFunctionThatRunsBackwards ENDP
```

Lets order things a bit. This is the above code after compilation:

```
{ prologue code ;
 < OP_1 size, OP_1 >,
 < OP_2 size, OP_2 >,
 .
 .
 < OP_N size, OP_N >,
 ; epilogue code }
```

The external patcher's job is post build code manipulation. First it finds the block with the aid of the flag. Next it moves all the sizes to the start of the block, and finally it sorts all the opcodes one after the other in a reversed order. This is how it will look after the external patcher's job:

```
{ prologue code ;
 < OP_1 size >,
 < OP_2 size >,
 .
 .
 < OP_N size >,
 < OP_N >,
 < OP_N-1 >,
 .
 .
 < OP_2 >,
 < OP_1 >,
 ; epilogue code }
```

The tracer's job is to grab the opcode's size from the array, and decrease it from **EIP** register. The output of this application is a full blown function that runs backwards! Sure, it has its disadvantages, like not supporting jump / loop instructions, but it beats writing a full disassembler to accomplish this task.

14.1.7 Final Words

What a rush, ahh? Just imagine what more we can do with these techniques...

For example, we can even combine both **AUTO_LOCK** and **REV_BLOCK** on the same function, one on top the other. If we make sure the external patcher first processes the **REV_BLOCK** code, and then processes **AUTO_LOCK** code, then at run time this function will unlock itself, then execute backwards and finally lock it self back!

Personally I would like to thank 29A guys for their magazine and wonderful low level work. I would also like to thank whoever wrote "assemblur" crackme (search at *crackmes.de* under DOS crackmes). This crackme is a beautiful piece of work, and it gave me many ideas about how far dynamic code can go.
And, of course, thank you for reading all this ;)

Cheers,
The+Q

14.2 DLL Injection Part I - .exe surgery without an incision – by isildur

The following article will show you how you can do code injection without modifying in any way the target application. No patching or whatsoever. We will inject our own dll in the target address space and then take complete control of it.

14.2.1 How it's done

There is more than one way to acheive this and the following method is one of them. It works on all 32 bit Windows platform.

Ok, so how do you do this? With a loader and using a Windows API function called SetWindowsHooksEx. This function is used to monitor the system for specific events. We want to monitor the messages sent to our target's message queue. For that we will use the WH_GETMESSAGE type of hook.

So we will use WH_GETMESSAGE as the first parameter of that function. The second parameter is a pointer to a callback function that will be called when a message is posted to our target. For our needs, that function will reside in a separate dll, not directly inside our loader. That's the key for our trick to work (more on that in a moment).
The third param is our dll's instance handle. The last param is the target's main thread ID.

So basically, to take control over our target, we need to find it's main window handle first. Then, using that handle we get the thread ID for that window. With that information, we can then set the hook and inject our dll. Our hook function will then subclass the target's main window with our own window procedure. With that, we can then do lots of things like add or modify menus, modify the behavior of the application etc. The limit is your imagination...

So you say "Yeah but how does the dll get injected?". Windows will do that automatically for you when you set the hook. Isn't that great? Ok, but we still have to follow some strict rules for everything to work fine.

14.2.2 In details

Remember when I said that the function pointer has to reside in our dll, not in the loader itself?
Here is what happens:

After setting the hook on our target:

1. A message is about to be dispatched to our target main window.
2. The system verifies if there is a WH_GETMESSAGE hook installed for this thread.
3. It finds our hook, then it checks to see if the dll containing the hook callback function is already mapped in the target's address space.
4. It will see that it's not mapped there, so it loads the dll in the target's address space. Bingo!
5. It then calls our callback funtion, and that's where the fun begins...

14.2.3 Implementation

Target: Notepad
Goal: Alter the way Notepad looks (adding a menu to change text and backgroud color)

Tools needed: MSVC (Other C compilers may work but I had to use some #pragma directives that may be different with other compilers.)

Let's have some fun with Notepad!

We will build a little loader and a dll to inject in Notepad. The loader will execute Notepad, get its main window handle and the window's thread ID. The loader will then set up the hook that will inject the dll in Notepad's address space. Our dll will add color functionality to Notepad.

You will need to download the source for the loader and dll to understand the rest of this article. Get it here (link) or at www.geocities.com/v_d_d (link).

14.2.4 Inside the loader

Look at the file notepadLoader.c. This is a very simple dialog based win32 application.
The function that does the job is InitializeApp(). Inside, we call loadNotepad() which just executes Notepad using CreateProcess. Then we load our dll using LoadLibrary and we get the pointers of two functions inside the dll, SetHook and UnsetHook. We then get the main window handle with a call to FindWindow using the class name of that window, which in our case is "Notepad". (By the way, make sure notepad is not running when you execute the loader because I did not put any code to update multiple instances of the target.)

I found the class name for Notepad's main window using a utility called Spy++ that comes with the MSVC compiler. Then we get the thread ID of that window with a call to GetWindowThreadProcessId.

And last but not least, we call our dll function SetHook passing the thread ID and window handle to it. That's all the loader does. When we close the dialog, we call UnsetHook to clean up, because it is not recommended to leave unused hooks behind in the system.

14.2.5 Inside our DLL

The first function we will look at is SetHook. We first check if there is already a hook.
We want to set the hook only once. We save the window handle to a global variable for later use. We make a call to SetWindowsHookEx with WH_GETMESSAGE. The second parameter we put is a callback function pointer, in this case called GetMsgProc. This callback function is implemented in the dll. This is the function that the system will call every time a message is posted to Notepad. I will describe it in a moment. We pass it the dll instance handle that we saved when the dll got loaded (in the DLL_PROCESS_ATTACH section).

The last param is the thread ID that the loader passed to SetHook. We save the caller's thread ID by calling GetCurrentThreadId. It is sometimes usefull if we want to send information back to the calling thread (not used in this example). Then we do a PostThreadMessage to Notepad's thread posting WM_NULL just to make it enter our hook callback function for the first time.

The next function we look at is our famous hook callback function called GetMsgProc. In our case, we want to do something only the first time this function is called, so we set a static flag to make it do stuff just the first time. This is the place you can call some of your own functions doing what ever you need to do. This gets executed in the target's process address space.

For our example, we want to subclass Notepad's main window procedure. The way you do this is by calling SetWindowLong. You pass Notepad's window handle (that we saved earlier in SetHook), GWL_WNDPROC as second param and last, a pointer to our own window procedure (that we implement in the dll also). This call returns a pointer to Notepad's original window procedure, so we save it to a global variable for later use (we need it). What this does is that every time Notepad's window procedure is called, it will call OUR OWN window procedure and we can filter the messages BEFORE Notepad.

In our window procedure, we return by calling the ORIGINAL window procedure for any message we want Notepad to process.

Now that we subclassed Notepad's window proc, we need the window handle of the edit box (where you type text in Notepad). So we call FindWindowEx which will search for child windows of the window handle we provide. We give it Notepad's main window handle and we use the Edit box's class name "Edit" to find its handle. We save the Edit box's handle to a global variable for later use. Then we call createAndLoadMenuItems which is a function we implemented that will add new menu items under the "Format" menu. We add a "Set background color" and a "Set text color" menu item. Look at the source for more details on that one.

We now have to implement the stuff that will happen when a user clicks on those new menu items. We do that in our window procedure. Look at the source code if you want the details for that. You could do anything you want at this point, now that you know how to set things up. My point was to show you how inject the dll, not teach the whole Win32 API ;-)

So if you click on the new menu items, you will be able to change the text and backgroud colors. Neat?

14.2.6 Important Note

If you look at the dll's source, you will see some #pragma directives at the top. This is vital for our trick to work. What this does is reserve some global variables to be shared between all processes that loads our dll. That means that a value set to one of those variables by one process will be reflected in the other processes that also loaded the dll. If we don't do that, each process will have a different copy of those variables. You have to understand that there is one process that loads the dll (our loader) and sets up the hook, and a second process (Notepad) that gets our dll loaded in its address space. Our dll is loaded by 2 different processes. So it is important that some variables be shared between the two processes for this method to work. Try it without the #pragma directives and you will see what I mean.

14.2.7 Final words

This was just a small example basically to show you how to set things up but with a little bit of imagination, you will soon find that the possibilities are great. What is cool about this method is that we didn't have to patch a single byte of the target application. The disadvantage though is that we need a loader to set up the hook. But depending on what you need to do, you can save loads of time by being able to code your stuff in C or C++ using the full extent of the Windows API. No messing with import tables and calculating jumps and addresses in machine opcode. You could even transform Notepad into a full fledged word processor! lol :)

In part II of this series of articles, I will show a similar method without the use of a loader. The byte patching is minimal though and the implementation gets done in a dll like with this method. I hope you learned something and that I was clear enough in my explanations.

isildur

v_d_d(a t)yahoo(d o t)com

14.2.8 Sources

14.2.8.1 Inject.h

```
// Copyright 2002 Isildur, all rights reserved
//
// email: v_d_d@yahoo.com
// web: http://www.geocities.com/v_d_d
//
////////////////////////////////////////////

////////////////////////////////////////////
#ifndef _INJECT_H
#define _INJECT_H

#ifdef __cplusplus
extern "C" {
#endif

int WINAPI SetHook(DWORD threadID, HWND hwnd);
void WINAPI UnsetHook(void);
LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK DLLNewProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK DLLEditProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam);
int createAndLoadMenuItems();

typedef int (WINAPI* TDefSetHook)(DWORD threadID, HWND hwnd);
typedef void (WINAPI* TDefUnsetHook)(void);



#ifdef __cplusplus
}
#endif

#endif // _INJECT_H
```

14.2.8.2 Resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by inject.rc
//

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        101
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

14.2.8.3 Inject.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifndef _MAC
////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x40004L
 FILETYPE 0x2L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904b0"
        BEGIN
            VALUE "Comments", "www.geocities.com/v_d_d\0"
            VALUE "CompanyName", "Virtual D@t@ Designs\0"
            VALUE "FileDescription", "inject this dll in Notepad's memory space\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "inject.dll\0"
            VALUE "LegalCopyright", "Copyright © Isildur 2002\0"
            VALUE "LegalTrademarks", "Isildur\0"
            VALUE "OriginalFilename", "inject.dll\0"
            VALUE "PrivateBuild", "\0"
            VALUE "ProductName", "inject\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
            VALUE "SpecialBuild", "\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END
```

```
#endif    // !_MAC


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

14.2.8.4 Inject.c

```
// Copyright 2002 Isildur, all rights reserved
//
// email: v_d_d@yahoo.com
// web: http://www.geocities.com/v_d_d
//
////////////////////////////////////////////

////////////////////////////////////////////
//
// Shows a DLL that can be injected in another
// process. This example modifies Notepad.
// See tutorial for more details.
//
////////////////////////////////////////////

#include <windows.h>
#include <windowsx.h>

#include "inject.h"


// start shared data between processes
#pragma data_seg("Shared")
HHOOK g_hook = NULL;
DWORD g_threadID = 0;
HWND  g_hwnd = 0;
#pragma data_seg()
// end shared data

// tell linker to make section readable, writable and shared
#pragma comment(linker, "/section:Shared,rws")


////////////////////////////////////
HINSTANCE g_dll;
WNDPROC OldProc;
HWND g_hwndEdit;

COLORREF g_backColor = RGB(255,255,255);
COLORREF g_textColor = RGB(0,0,0);
HBRUSH g_bgBrush = 0;
```

```
BOOL WINAPI DllMain(HINSTANCE hDLLInst, DWORD fdwReason, LPVOID lpvReserved)
{

    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
        {
                    // save the dll's instance
                    g_dll = hDLLInst;
             }
        break;

        case DLL_PROCESS_DETACH:
             {

             }
        break;

        case DLL_THREAD_ATTACH:
             {

             }
        break;

             case DLL_THREAD_DETACH:
             {

             }
        break;
    }
    return TRUE;
}
```

```
// threadID is the thread you want to hook to
// the hwnd should be the thread's main window
int WINAPI SetHook(DWORD threadID, HWND hwnd)
{

	if(!threadID)
		return 0;

	// make sure there is not already a hook
	if(g_hook)
	{
		UnhookWindowsHookEx(g_hook);
		g_hook = NULL;
	}

	// save the window handle
	g_hwnd = hwnd;

	// set hook
	g_hook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, g_dll, threadID);

	if(!g_hook)
	{
		return 0;
	}

	// save the caller's thread id, can be usefull
	// to send a message back
	g_threadID = GetCurrentThreadId();

	// forces the remote process to call our GetMsgProc
	// and map our dll in it's memory space
	PostThreadMessage(threadID, WM_NULL, 0, 0);

	return 1;

}

void WINAPI UnsetHook()
{
	if(g_hook)
	{
		UnhookWindowsHookEx(g_hook);
		g_hook = NULL;
	}
}

// our main entrance in the remote process's memory space
LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam)
{
	static BOOL firstTime = TRUE;

	// do this only the first time
	if(firstTime)
	{
		firstTime = FALSE;

		// subclass the main window of our target
		OldProc = (WNDPROC) SetWindowLong(g_hwnd, GWL_WNDPROC, (LONG)
DLLNewProc);

		// get the window handle of Notepad's edit box
		g_hwndEdit = FindWindowEx(g_hwnd, NULL,"Edit",NULL);

		// load our new menu items
		createAndLoadMenuItems();
	}

	return(CallNextHookEx(g_hook, nCode, wParam, lParam));
}
```

```
// this is the subclass message proc
// here we intercept all messages sent to the target's main window
LRESULT CALLBACK DLLNewProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
             case WM_COMMAND:
             {
                    switch(LOWORD(wParam))
                    {
                           // menu for setting bg color
                           case 57602:
                           {
                                  static CHOOSECOLOR cc ;
                                  static COLORREF    crCustColors[16] ;
                                  BOOL result;

                                   cc.lStructSize    = sizeof (CHOOSECOLOR) ;
                                   cc.hwndOwner      = NULL ;
                                   cc.hInstance      = NULL ;
                                   cc.rgbResult      = g_backColor;
                                   cc.lpCustColors   = crCustColors ;
                                   cc.Flags          = CC_RGBINIT | CC_FULLOPEN ;
                                   cc.lCustData      = 0 ;
                                   cc.lpfnHook       = NULL ;
                                   cc.lpTemplateName = NULL ;

                                   result = ChooseColor (&cc);

                                   if(result)
                                   {
                                          if(g_bgBrush)
                                          {
                                                DeleteObject(g_bgBrush);
                                                g_bgBrush = 0;
                                          }

                                          g_backColor = cc.rgbResult;

                                          g_bgBrush = CreateSolidBrush(g_backColor);

                                          InvalidateRect(g_hwndEdit, NULL, 1);
                                   }
                           }
                           break;
```

```
                        // menu for setting text color
                        case 57603:
                        {
                               static CHOOSECOLOR cc ;
                               static COLORREF    crCustColors[16] ;
                               BOOL result;

                                cc.lStructSize    = sizeof (CHOOSECOLOR) ;
                                cc.hwndOwner      = NULL ;
                                cc.hInstance      = NULL ;
                                cc.rgbResult      = g_textColor;
                                cc.lpCustColors   = crCustColors ;
                                cc.Flags          = CC_RGBINIT | CC_FULLOPEN ;
                                cc.lCustData      = 0 ;
                                cc.lpfnHook       = NULL ;
                                cc.lpTemplateName = NULL ;

                                result = ChooseColor (&cc);

                                if(result)
                                {
                                       g_textColor = cc.rgbResult;

                                       InvalidateRect(g_hwndEdit, NULL, 1);
                                }
                        }
                        break;
                }
        }
        break;

        case WM_CTLCOLOREDIT:
        {
               // message we get when the app wants to draw the edit control
               if(!g_bgBrush)
               {
                      g_bgBrush = CreateSolidBrush(g_backColor);
               }

               SetBkColor((HDC) wParam, g_backColor);
               SetTextColor((HDC) wParam, g_textColor);

               return (LRESULT)g_bgBrush;
        }
        break;

        case WM_DESTROY:
        {
               // clean up
               if(g_bgBrush)
               {
                      DeleteObject(g_bgBrush);
               }
        }
        break;
    }
    return CallWindowProc(OldProc, hwnd, message, wParam, lParam);
}

int createAndLoadMenuItems()
{
       HMENU mainMenu, subMenu;
       mainMenu = GetMenu(g_hwnd);
       subMenu = GetSubMenu(mainMenu, 2);
       AppendMenu(subMenu, MF_SEPARATOR, 0, NULL);
       AppendMenu(subMenu, MF_STRING, 57602, "Set background color...");
       AppendMenu(subMenu, MF_STRING, 57603, "Set text color...");

       return 1;
}
```

14.2.8.5 Notepadloadererres.h

```
/* Weditres generated include file. Do NOT edit */

#define      DLG_100      200
```

14.2.8.6 Notepadloader.rc

```
/* Wedit generated resource file */
#ifdef __LCC__
#include <windows.h>
#endif
#include "notepadloaderres.h"


DLG_100 DIALOG 7, 20, 195, 86
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Notepad Loader by isildur"
FONT 8, "Helv"
BEGIN
    PUSHBUTTON      "Exit", IDCANCEL, 78, 50, 40, 14
END
```

14.2.8.7 Notepadloader.c

```
// Injecting a dll in a target address space example
//
// By: isildur, December 2002 for anticrack.de
//
// email: v_d_d@yahoo.com
// web:   http://www.geocities.com/v_d_d
//
// You will also need the code for the inject.dll that this loader needs
//
//
// Warning: This code is just an example. There is almost no error checking
//          to make things simple. You will need to check the return values
//          of every function if you want to use this code for something robust.
////////////////////////////////////////////////////////////////////////////////


#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <string.h>
#include "notepadLoaderres.h"

typedef int (WINAPI* TDefSetHook)(DWORD threadID, HWND hwnd);
typedef void (WINAPI* TDefUnsetHook)(void);

HWND notepadHwnd;
DWORD notepadThreadID;
STARTUPINFO si;
PROCESS_INFORMATION pi;
HMODULE ourDll;

BOOL hookPresent = 0;

// our dll function pointers
TDefSetHook DLLSetHook;
TDefUnsetHook DLLUnsetHook;

// prototypes
static BOOL CALLBACK DialogFunc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM
lParam);
int loadNotepad(void);
void cleanUp(void);



int APIENTRY WinMain(HINSTANCE hinst, HINSTANCE hinstPrev, LPSTR lpCmdLine, int
nCmdShow)
{
	WNDCLASS wc;

	memset(&wc,0,sizeof(wc));
	wc.lpfnWndProc = DefDlgProc;
	wc.cbWndExtra = DLGWINDOWEXTRA;
	wc.hInstance = hinst;
	wc.hCursor = LoadCursor(NULL, IDC_ARROW);
	wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
	wc.lpszClassName = "notepadLoader";
	RegisterClass(&wc);

	return DialogBox(hinst, MAKEINTRESOURCE(DLG_100), NULL, (DLGPROC)
DialogFunc);
}
```

```
static int InitializeApp(HWND hDlg,WPARAM wParam, LPARAM lParam)
{
	int retVal;

	// load notepad
	retVal = loadNotepad();

	if(!retVal)
		return 0;

	// load our dll
	ourDll = LoadLibrary("inject.dll");

	if(!ourDll)
	{
		return 0;
	}

	// get the address of our SetHook function in our dll
	DLLSetHook = (TDefSetHook) GetProcAddress(ourDll, "SetHook");

	// get the address of our SetHook function in our dll
	DLLUnsetHook = (TDefUnsetHook) GetProcAddress(ourDll, "UnsetHook");

	// get notepad's main window handle using class name
	notepadHwnd = FindWindow("Notepad", NULL);

	// get notepad's thread ID
	notepadThreadID = GetWindowThreadProcessId(notepadHwnd, NULL);

	// now set the hook
	retVal = DLLSetHook(notepadThreadID, notepadHwnd);

	return retVal;
}
```

```
int loadNotepad(void)
{
	int retVal;


	si.cb = sizeof(STARTUPINFO);

	si.lpDesktop = NULL;
	si.lpTitle = NULL;
	si.dwFillAttribute = 0;
	si.dwX = 0;
	si.dwXCountChars = 0;
	si.dwXSize = 0;
	si.dwY = 0;
	si.dwYCountChars = 0;
	si.dwYSize = 0;
	si.wShowWindow = 0;
	si.dwFlags = 0;
	si.hStdError = 0;
	si.hStdInput = 0;
	si.hStdOutput = 0;
	si.cbReserved2 = 0;
	si.lpReserved = NULL;
	si.lpReserved2 = NULL;

	retVal = CreateProcess( NULL,
					"notepad.exe",
					NULL,
					NULL,
					FALSE,			// handle
inheritance option
					NORMAL_PRIORITY_CLASS,
// creation flags
					NULL,			// new
environment block
					NULL,		// current
directory name
					&si,
					&pi);


	if(retVal)
		// wait for notepad to finish loading
		WaitForInputIdle(pi.hProcess, 10000);
	else
		{
			retVal = GetLastError();

			return 0;
		}
	return 1;
}
```

```
static BOOL CALLBACK DialogFunc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{
	switch(msg)
	{
		case WM_INITDIALOG:
		{
			hookPresent = InitializeApp(hwndDlg,wParam,lParam);
			if(!hookPresent)
				MessageBox(NULL, "Error setting hook!", "Error",
MB_ICONERROR);
		}
		return TRUE;

		case WM_COMMAND:
		{
			switch (LOWORD(wParam))
			{
				case IDOK:
					EndDialog(hwndDlg,1);
				return 1;
				case IDCANCEL:
					cleanUp();
					EndDialog(hwndDlg,0);
				return 1;
			}
		}
		break;

		case WM_CLOSE:
			cleanUp();
			EndDialog(hwndDlg,0);
		return TRUE;
	}

	return FALSE;
}

void cleanUp(void)
{
	if(hookPresent)
	{
		DLLUnsetHook();
	}
}
```

14.3 Playing XM Files in your Source code without 3d party dlls – by Ben[9]

14.3.1 Over View

Heya all,

the perfect way to play XM music is by using the MiniFmod, since it is free to use, we can produce really cool keygens. I have chosen keygens as the perfect target to play music on, as we all know its cool in the end.

all right, lets begin.

first we will deal with the music.

The best way to find our XM music is the mod archive located at:
http://www.modarchive.com/

it is a huge archive, and allot of cool music can be found there, so just before coding, select ur file (recommended size : 2k-30k).I especially like the "Hybrid Song.XM", (i first heard it in a installer of Worms :-)) or "trainer.XM", but i am sure there are millions of them out there.

once we choose our music, we need to dump its content!!

now, sicne this article is for Visual C++ coders, our dump is apparently C++ style hex.
for the dumping rutine we will use Thigo's excellent Table Extractor, located at protocols[10] / anticrack[11] or just google for it.

14.3.2 Duming Table

```
/* Dumping Hybrid Song.XM (51k) */
```

Load your XM file in Thigo's Table extractor, now we need to supply few info to the program:

```
Address (HEX): 0
Items: Bytes
Size (Bytes) of Table(Hex): CCD4 /* File Size */
Number Of Items on a Line: 20    /* put how many you like personally i put
                                    20 */
C/C++                            /* Dump Style Language  */
Dump to file: checked            /* [v] */
```

ok, we dont need anymore options filled.
Press the "Go !" button and we see our thread make an effort to dump
the Image as fast as possible ;-),
0....100%, great Table dumped in result.txt (same dir as the XM file).

---

[9] The Sources are included in the magazines file package
[10] http://protools.cjb.net
[11] http://www.anticrack.de

14.3.3 Coder Point of View

Look at the dumped results.txt. geesh, it is very big..(300k).
Yes, that's the coast of files that have 20++k and above!
So keep in mind that since your making a keygen, the file should be as minimal as possible, but consider the best sound u can get :-)
Save this file somewhere, we need to use some source code from now on.

14.3.4 The MiniFmod

Information about MiniFmod 1.6:

- This small XM replay system only adds 5k to your exe! Now including FULL SOURCE CODE!!!
- FEXP tool to export a header based on your song, which will be compiled into MiniFMOD and exclude whole portions of code!
- Pre-buffered output for 0 latency, and high output stability 100% click free.
- XM sample callbacks for user generated or compressed XM samples!
- File system callbacks so you can specify whatever loading system you like! (disk/wad/memory)

Copyright 2001-2002 Firelight Technologies Pty.

The MiniFmod can be found at http://www.Fmod.org or for quick download:

http://www.fmod.org/files/minifmod160.zip

note: I dont use the GCC, so if you do, u'll have to sort things out :-)

14.3.5 The Tools:

Inside the MiniFmod package, there is a small tool called Fexp.exe, this will be our XMeffects optimizer for the dumped result.txt!!
Other that, we have the *.c/*.h files that we will include in our program.

I have included a keygen template which will suite us for including the sound code.
That way, we will code together step by step.

Ok, get the template and load the DSW into VC++ IDE. it is a very basic keygen with 3 buttons, Generate, Play, About.

Our main goal is to make the sound be playable via the Play button, it is later to be loaded automatically when keygen starts. Ok, so we have our keygen.cpp with everything ready.

14.3.6 Starting to Code

the Sound must have an Handle, of course, that way we can determine whenever we try to stop, reload, and even sharing with other sound players (i.e: Winamp)

As a global variable, we start with:

```
FMUSIC_MODULE *mod; /* fmod music handler */
```

Now, before we add more code, lets add to our project the minifmod functions and main code.

Make a new folder in the keygen's main folder, call it "Lib". You can see it is also in the minifmod16.zip. copy the files from the minifmod16.zip\lib -> our lib folder.
Note: in the zip\lib there is 2 folders named "debug/final", do not copy its content!

Alright, now back to our project, in the project dir viewer, make a folder named "MiniFmod.c" (under the Source Files folder). Add all *.c files into it from our lib folder.

these files are:

```
1. Fmusic.c
2. Fsound.c
3. mixer_clipcopy.c
4. mixer_fpu_ramp.c
5. music_formatxm.c
6. system_file.c
```

Now we need to add the header files (*.h), make a new folder named "MiniFmod.h" (under the Header Files folder). Add all *.h files into the it from our lib folder + the lib file.

These files are:

```
1.  minifmod.h
2.  Mixer.h
3.  mixer_clipcopy.h
4.  mixer_fpu_ramp.h
5.  Music.h
6.  music_formatxm.h
7.  Sound.h
8.  system_file.h
9.  system_memory
10. xmeffects.h
11. winmm.lib    ; not an *.h file, but dont forget to add it too!!!
```

Now, since its a keygen and using the standard minifmod source is not enough, we will make some modifications into its code, that means some files from the original has been slightly modified to suite our needs!

Note: the modified files have been added to the source code, so i took of it already.

Another thing we need to do is to create a new folder named "loadmusic.h" (under the Header Files folder). Now listen carefully, remember the result.txt we have dumped ?
We will use it's hex dump as the file it self!, rename the result.txt -> music.h (as specified in the modified files :-))
Open it (be careful, editing big files can cause a crash :-)), now we need to edit the file a bit.

```
original Dump:
==============
static unsigned char table[52436] = {};

Modified Dump:
==============
#define MUSIC_LEN 4092

static unsigned char music[]=
{
};
```

Note: the diffrences, always do this when trying to load new music into ur keygen!!!
Once we finished editing and saving, we need to make a new *.h file:

Open notepad and write/paste this code:

```
//=============================================================
#ifndef __LOADMUSIC_H__
#define __LOADMUSIC_H__

// Memory functions
unsigned int memopen(char *name);
void memclose(unsigned int handle);
int memread(void *buffer, int size, unsigned int handle);
void memseek(unsigned int handle, int pos, signed char mode);
int memtell(unsigned int handle);

void loadmusic(void); // Load music fucntion

#endif
//=============================================================
```

It's a few add-ons, basically we are making it easier for us to load the dumped xm file into memory and make it playable. save this file as loadmusic.h
Finally, we add those 2 files (music.h / loadmusic.h) into the loadmusic folder.

Last and not least, we need to add the loadmusic.cpp, which is ofcourse not a part of minifmod, as i said it is our modified code.
Make a folder named "loadmusic.c" (under the Source Files fodler).
Add this file (loadmusic.cpp)

All right, this fun the setup part of minifmod, we now need to add those as #include in our main keygen.cpp

```
#include "loadmusic\loadmusic.h"
#include "lib\minifmod.h"
#include "lib\buffer.h"
```

Note: if you take a look at loadmusic.cpp you can see our music.h declared there, along with our new modified functions.

14.3.7 The Last Touch

we have declared the handle of our fmod music, remember?

```
FMUSIC_MODULE *mod;
```

Right, all we need to do is to set the Fmod up and load the dumped xm file (music.h) into memory and alst call the play function!

At our play button id code (case IDC_PLAY:)

We add this code:

```
if (mod == NULL) // mod handle is free? (though it will work fine with
other loaded audio devices)
{
   // We defined our music file to be loaded in loadmusic.cpp //
   //=============================================================//
   loadmusic(); // Call & set ready memory to load the music
   if(!FSOUND_Init(44100, 0)) // intialize memory for sound and format
   {
      break;
   }
   mod = FMUSIC_LoadSong(NULL, NULL); // handle = LoadSong()
   FMUSIC_PlaySong(mod); // Play it (from memory), way cool!!
}
```

Note: if you put this code in the WM_INITDIALOG, you will get autoplay effect when loading the keygen!

If we want to Stop the music only thing we need to do is add new button called Stop, choose an ID to it and use this code:

```
if (mod != NULL) // handle is loaded (playing)?
{
   FMUSIC_FreeSong(mod); // Free memory (handle)
   FSOUND_Close();       // Close it (stop it from playing)
   mod=NULL;             // make handle to be Free again
}
```

Note: add same code at WM_CLOSE msg!!

14.3.8 Important Tips

This is the part where if u doesn't read it, u will get funny results when playing your XM file!!

```
/* from Firelight's readme */

Making your exe even smaller with XMEFFECTS.H and FEXP.EXE:
- Note all the effects in the xm replay code are wrapped with #ifdef's.
They are defined in
  xmeffects.h.
- Have a look in the zip/lib directory .. There is an executable called
FEXP.EXE
- Use FEXP.EXE on an xm file and it will generate xmeffects.h for you!
- Recompile the minifmod library with the new xmeffects.h
- See your exe size go down AGAIN!
- (only play the song you fexp'ed or it will screw up on other songs
probably :) ..)
```

Yup, this mean, we make a new xmeffects.h for each music file we dump!!
Else as said, we have a screwed up music :-)

-> fexp Hybrid_Song.XM -> xmeffects.h

Overwrite the old xmeffects.h in our keygen\lib\ folder, and recompile our source code, see your music plays fine.

Some little note is that the minifmod a bit looses the quality of the music, well this is understandable since minifmod is not a part of the fmod library, so it does miss some important stuff, but hey, nothing is perfect!!

For me, it rox to have cool sound in keygens, and now you guys too.

14.3.9 Ending

Well, this complete the essay, the best way to learn from it is by using the files included in the compiled keygen, that way u can always have this as a base keygen.

MiniFmod is coded by FireLight (r)
Keygen source code + music coded by Ben & [Goofy] of FreeStylerS 2002

Greets to all RCE members, and the REA community!

14.3.10 Message to the Magazine

tools needed to recompile:
Thigo's table extractor
minifmod 1.6 (http://www.fmodorg)
visual c++ 6.0

Note: keygen with sound already have everything + modified version so its not really necessary to add the minifmod.zip, but i do recommend to to put the fexp.exe

Ben

14.4 OllyDbg 1.08b – A craxor review by FuZzYBiT

Hello dear readers.
That's my new essay which is being published in this mag. Hope you like it.
And sorry for my bad English.

OllyDbg is a application level debugger. It is different from SoftIce, which is a system level one. Unless you're working with the Windows kernel or Windows loader or drivers or other such low-level things you probably won't need SoftICE, as OllyDbg should be able to do everything you need in regards to debugging applications.

So, let's get back to work.

14.4.1 Is there a way to make things clearer?

Yes, there is.

Hit ALT+O and you will be prompted with DEBUGGING OPTIONS screen.

In DISASSEMBLER you can set:

- Tab between mnemonics and arguments
- Extra space between arguments
- Show default segments
- Always show size of memory operands
- Show local module name
- Show symbolic addresses

In CPU you can set:

- Underline fixups
- Show direction of jumps (this is useful, you can know if it jumps forwards or backwards)
- Show jump path (it displays a line pointing where it will jump)
- Show greyed path if jump is not taken
- Gray commands that fills gaps between procedures

In ANALYSIS 1:

- Show ARGS and locals in procedures
- Show arguments of known functions (This is pretty nice)
- Guess number of arguments of unknown functions
- Auto start analysis of main code
- Analyse code structure
- Decode cascade IF's as SWITCHES (That's cool … the code gets clearer)
- Trace contents of registers
- Decode tricky sequences
- Keep analysis between sessions

Wow, now the code will be like that:

```
0040100B  |.  8B45 08       MOV     EAX, [ARG.1]
0040100E  |.  8BD0          MOV     EDX, EAX
00401010  |.  8955 FC       MOV     [LOCAL.1], EDX
00401013  |>  33C9          XOR     ECX, ECX
00401015  |.  8B45 0C       MOV     EAX, [ARG.2]
00401018  |.  8B55 FC       MOV     EDX, [LOCAL.1]
0040101B  |>  8B32          MOV     ESI, DWORD PTR DS:[EDX]
0040101D  |.  8B7A 04       MOV     EDI, DWORD PTR DS:[EDX+4]
00401020  |.  0FAF78 0C     IMUL    EDI, DWORD PTR DS:[EAX+C]
00401024  |.  0FAF30        IMUL    ESI, DWORD PTR DS:[EAX]
```

Are you seeing [ARG.1] and [LOCAL.1]? Cool, huh?

In STRINGS:

- Decode pascal-style string constants
- Mode of string decoding (you can choice what you prefer here)

14.4.2 How to set breakpoints?

You just hit F2 when tracing your code. A breakpoint will be set. Hit F9 and you will see!

14.4.3 Setting breakpoints on sections: .code, .text, and so on.

Hit ALT+M, and you will see MEMORY MAP. Look for your filename in column OWNER …
Now you can hit F2 on your desired section.

14.4.4 Conditional Breakpoint

This is a nice feature. It's a little hidden one. So, you can breakpoint setting some conditions.
Hit CTRL+T… or even (which is different from the first one) hit SHIFT+F4.
As you can see … you can set EIP range … so you can break at it.
But if I want to break when EAX=00402558?
Well, check CONDITION IS TRUE… and fill it with: "EAX==004025558".
Done! ☺
The usage is pretty straightforward. I just gave one example.

14.4.5 Tracing the code!!

You can trace until finding a RETN command … so, first you should STEP INTO a procedure (F7) and … when you are done with you tracing inside it, you can EXECUTE UNTIL RETURN (CTRL+F11). If you don't want to trace inside a routine, just STEP OVER (F8).

Wow, I guess that's we are waiting for. Here we will see a lot of new things that can be made with OllyDbg.

14.4.5.1 How do I set command line arguments?

Easy. Menu DEBUG->ARGUMENTS …
You will be prompted to enter your command line parameters. Restart your application with CTRL+F2. Done!

14.4.5.2 How do I know what's going on withing procedure calls?

Hummm… I just can help you to figure it out.
Local variables are stored at the stack. We've already changed our configurations so OllyDbg can display LOCAL variables and ARGUMENTS.

Some useful values may be at the stack. The stack stuff is located at your down-right window.
Tracing and will see where LOCAL variables are stored now. ☺

And ... if you are dealing with recursive calls, you can Now you can hit ALT+K and see call stack … this is also usefulif you are dealing with nested function calls as well.

14.4.5.3 How can I see what is stored at memory locations?

Nice. Well … you just right click the line you want to check. And then FOLLOW IN DUMP -> MEMORY ADDRESS.

At your left-down window you will see what is stored there!!!

14.4.5.4 I'm running a threaded proggy. How to deal with that?

Hummm, Menu VIEW-> THREADS.
You will see the handle of each thread. And you can know the entry point of each one.
You can set breakpoints like if you were tracing a normal proggy.
You can see if it is active (running) or if it was already closed.

14.4.5.5 I'm looking for string references…

Right Click … SEARCH FOR -> BINARY STRING … fill it with what you need and Voi la!

14.4.5.6 Is there a way to make things easier?

Comment your code, so you can keep you brain "awake" … without loosing precious information. Double click a line at fourth column and you will be prompted to fill a comment.

14.4.5.7 Is there a way to breakpoint when a DLL is loaded?

Yes! If the DLL is loaded explicity via LoadLibrary or on demand (DelayLoad feature), you can "BREAK ON NEW MODULE" … which is under DEBUGGING OPTIONS->EVENTS.

If you only want to trace a known specific DLL export, set a breakpoint in that list. Go and it will break on the first byte of the function prolog.
You can then evaluate the call stack to get the caller. Using the disassembly you can now identify the parameters and the calling convention (cdecl, stdcall and so on).

14.4.5.8 Oh … but I need something OllyDbg can't do. SoftIce can trace application messages. And Olly?

And so does OllyDbg. That's a very "hidden feature". I guess it is sooo useful.

1. Open program
2. Names window [CTRL+N in CPU Window]
3. Find User32.TranslateMessage API
4. right click/FindReferences
5. conditional breakpoint [SHIFT+F4]
6. expression: MSG
7. Log function arguments: Always

If you cannot find it, try right click SEARCH FOR-> ALL INTERMODULAR CALLS.

But if I want to trap a specific message like WM_COMMAND?

To Log Only WM_COMMAND

Do it in this fashion:

1. Open program
2. Names window [CTRL+N in CPU Window]
3. Find User32.TranslateMessage API
4. right click/FindReferences
5. conditional breakpoint [SHIFT+F4]
6. Condtion box: MSG==WM_COMMAND
7. Log function arguments: On Condition

If you can't find User32.TranslateMessage API, do the same as above.

14.4.5.9 Patching files and Code injection!

Uhu! Nice stuff now!

In order to patch files, you will use a command named ASSEMBLE… you just double-click a line.

You can change the instruction.
If you go down until you find the end of code and insert commands there, you will be injecting code.
You can sure write JMP 00401050 and ASSEMBLE it.
It will calculate the correct opcode for you.
If you made a mistake, you can ALT+BACKSPACE. It will undo what you have messed up. ☺

If you want to make those changes PERMANENT, select all code you injected/changed, just right click, select COPY TO EXECUTABLE FILE. And now, confirm changes by selecting a new filename if you want.
Your changes will be saved and you will have a nice way to inject your code. ☺

14.4.5.10 Hummm, is there something else?

Nah, we're almost done.
I forgot to mention that you can set a new origin, so you can "retrace" any part of your code. That is useful instead of restarting your application.

Just right-click and NEW ORIGIN HERE.

You can use it to edit string if you want… just select what you want to edit (hold shift if you want to edit more than one char and CTRL+E). Use it combined with FOLLOW IN DUMP and your left-down window.

Ok … I hope you enjoy this article.
See ya!
FuZzYBiT

15 Monthly contest

Here we go with a short “coding” contest. Actually this is not a real contest. If you are looking for contests please refer to http://contests.anticrack.de. Because this is NOT a real contest, we will publish some good solutions next issue but there will be no “winner” at all.

This monthly contest idea comes by +Q.

```
The challange is to find a cool and interesting way to implement an
algorithm that produces shuffled numbers.
I mean, if we choose base 3, the output should be: 012 021 120 102 210 201

All possible combinations with 0,1 and 2, where each digit appears only
once.

The good solutions could be the smallest code, and a cool indirect way to
generate these numbers.
```

Submit your solutions to:
codebreakers@anticrack.de

16 Some interesting RCE links – by ManKind

Previously there is list of RCE-related links created by CrashTest. I am, however, sad to say that many of the links listed there are already down due to the proactive legal steps taken by law-keepers recently to shut down sites of such nature. However the emergence of some very good sites after that has brought the RCE scene to a totally new era. It is hoped the current good sites can be maintained and at the same time new sites would mushroom just in to time to revive the RCE scene to what it was like before or even lift it into a higher level. Below are some of the good RCE sites currently available.

http://www.anticrack.de
Internet-Portal for Scientific Software-Protection, Scientific Reverse-Engeneering and IT-Security. This is the best place for the latest (including some older) RCE-related news, articles, downloads and links. Also in my personal opinion this site and its affiliates are the most revolutionary RCE effort ever. You'll know how good this site really is when you realize that some of its best affiliates are also listed here.

http://www.reverser-course.de/ - The Reverse-Engineering-Academy
I regard this as the most organized RCE revolution. The REA is open to everyone, it's free, members can learn and proceed at their own pace and best of all it is also 100% legal. Anyone who completed all the tasks can really regard themselves as truly capable reverse engineers. This site is the answer to the fall of many great sites in recent months. It deserves freqeunt visits, participation and undivided supports from our community.

http://www.crackmes.cjb.net
The good old crackmes site which has survived thus far. It now has new interface as well as efficient web database system (and of course the crew behind it) to maintain crackmes and their solutions effectively. A mine of legal protections for you to try your hand on.

http://www.crackmes.de / www.reversemes.de
(added by Zero ;))

http://mup.anticrack.de/ - Code_Inside's Manual Unpacking (MUP) Page
As the name suggests, this site is about manual unpacking. Thus it contains a whole lot of unpacking tutorials and some stuffs on PE information. Deserves some browsing if you're keen to learn unpacking.

http://www.protools.cjb.net
Well what can I say? This site has served all the many reverse engineers, programmers as well as power users through all thick and thin. An always up-to-date site, (almost) daily updates, and offering extremely good collections of utilities we would ever have needed. Thumbs up to the person behind it and hope this site could stay with us for as long as possible.

http://www.tsehp.cjb.net
Last Fravia's mirror of Reverse code engineering. Contains full archive of (old) fravia's RCE contents, and new articles, tutorials and stuffs which the old sites never and would not have. However I notice that updates are rare now, and this site may be good only for keeping and archiving now, rather than being depended upon as a means of up-to-date knowledge and studies. Nevertheless, it's a good site to check out the Revirgin tool, which is a very good tool to assist in unpacking tasks.

http://www.tres2000.cjb.net
This is one nice group's site with contents which certainly worth viewing. It seemed it has not been updated since late 2001 but there is statement saying that the group is alive again which was posted in late 2002. Let's hope we can see how lively the group could be then.

http://execution.cjb.net/
This group really deserves anything beyond praise. The members have produced many quality tools and utilities which has certainly helped ease some RCE tasks. Besides I have heard a few times that this was barely alive in the past. However, congratulations to them, owing to the members perseverance, they managed to maintain the group thus far. Keep up the good work guys.

http://home.no/detten/ - BiW Reversing
This is a great site. Some creative crackmes and good solutions available. Besides there are also some good tutorials, reversemes, and tools.

http://www.new2cracking.cjb.net/ - New 2 Cracking
This is the site of a group of people who have broken off the links with C4N and probably against the way it was run, somehow of an outcast. However, they are keen on helping newbies and wannabes to learn about cracking, and I think they deserve to be listed just based on that reason. What more with their effort and desire to really spread cracking knowledge?

http://unpack.cjb.net/ - Unpacking Gods
They call themselves the unpacking gods. I don't know what the other thinks but I think they deserve such a name, judging from the excellent unpackers for various packers as well as crypters that they produce. There are also some invaluable source of unpackers for studying purposes. This group is nothing short of a miracle, assembling all the best people in the unpacking business.

http://kickme.to/dbc
A very large lists of tutorials and tools. Some are really good, and worth checking out.

http://evidence.6x.to/
To start with, what a nice site design. There is also equally good tutorials, utilities and releases section. All to all a good site to visit, although most crackers would appreciate to do with more tutorials on it.

http://www.lockless.com/
A very good site. I am especially attracted to its (quite) big cryptography content. Nevertheless, it also offers some tutorials, artworks, and tools. It already has so much to offer even when some of its sections are still under construction. I think I would have great pleasure browsing around this site when it is 100% complete.

http://alfredkmlo.cjb.net/ - Alfred Lo's Information Security Page
Page of the author of "Software Protection and its Annihilation". Quite a detailed piece of article explaining, well, as the name says, software protection, ways around it and how to protect better. I would recommend anyone who do not consider themselves as master of cracking to obtain and read this.

http://ghiribizzo.virtualave.net/icedump/icedump.html
Homepage of the excellent icedump tool. icedump is a free plug-in that extends NuMega's SoftICE with new commands and features.

http://www.secretashell.com/tKC/main.html
This is tKC's latest site. Many crackers just love his tutorials collections. However I realize that the quality of the tutorials in his most recent collections are way beyond what it should be. Not that tKC's own essays are bad, his most recent one is damn good. I guess he just had to stop publishing some lamers' tutorials which do not teaching anything at all. This page definitely worth a look, for it has all tKC's tutorials and some of his own programs.

http://www.pilotwarez.com/
The best site for PDA cracking. There are many tutorials, tools and links which you point you into the right way of PalmOS software cracking.

http://scared.at/scarebyte
ScareByte has some nice tutorials here.

http://www.fraviamb.cjb.net - +Fravia's General Reversing Board
A real good messageboard for the cracking community. It is popular among us and is a good place for active discussions and feedback.

http://board.anticrack.de/ - Reverse-Engineering Forums
This is the only RCE messageboard which has gained fastest popularity, I think. It has its reasons and uniqueness for such success. A more than complete messageboard anyone would have needed. Check it out and do participate actively.

http://www.s-i-n.com/chaos/ - Trainerology: The School of Advanced Game Hacking
One of the rare and excellent game training site. It's one of the interesting RCE path. I don't if many people like the idea of game training, so I provided only one link. This site, however, will provide you with the links you need if you ever want to know more about game training.

http://aod.anticrack.de/ - The Art of Disassembly
Have you ever wondered how disassemblers work? Want to be closely involved in building a real disassembler? Then look no further than this. This is an excellent effort by the crew of the ANTICRACK portal. Art of Disassembly understands itselves as a Handbook for writing a disassembler.

http://f0dder.has.it/ - f0dder's site
f0dder is a very good cracker and coder. It seemed to me that he made it sure that his site only contains quality stuffs like crackmes, tools, articles and tutorials. He certainly emphasises most on quality and least on quantity. Nice site worth looking into and learning from.

http://crudd.cjb.net/
Creative site design. Has some nice tutorials, crackmes and reversemes. Worth checking into.

http://kanal23.knows.it - Kanal23
A new RCE group.

17 Final Words

Well, this is the end of the first issue of the CodeBreakers magazine.

For sure not everything is maybe as you whished or would like to have but we think it was interesting for you to read.

If you have any comments,suggestions or ideas please contact us at codebreakers@anticrack.de.

I have to say thanks to all the contributors of this issue, especially to +Q who worked as “beta-reader” and helped to delete useless informations.

The next issue will be published in about 3 moths. Collecting this whole information material, sorting it out and formatting it takes shit of time so please be patient with us. Better more quality than bad quantity.

17.1 Where the RCE network wants to go…

Well, where do we want to go ?

Since the last 2-3 years there had been several changes within our reverse-engineering network.

First we had been able to focus the important “teaching” sites together. After reordering the different sites and “putting” them together it seems that sites like the crackmes site got more visitors.

Next we had several problems with different providers. After all this we decided to install an own server with very fast connection. On this server we installed ONLY RCE sites – no more idiots crashing SQL servers with buggy scripts and more.

Actually we count over 2.5 Million hits a month, the AntiCrack portal alone with 750000 hits. With traffic of about 30 GB a month we are sure that people are interested in our profession.

The reverse-engineering network contains now the following sites:

AntiCrack Deutschland	www.anticrack.de
Crackmes Website	www.crackmes.de
Reversemes Website	www.reversemes.de
The Reverse-Engineering-Academy	www.reverser-course.de
Codeinside Webpage	codeinside.anticrack.de
Fravia (old frozen mirror)	fravia.anticrack.de
Manually Unpacking Page	mup.anticrack.de
^Daemon^ Cave	daemon.anticrack.de
Our Forum	forum.anticrack.de
BOFH	bofh.anticrack.de
CodeBreakers Magazine	codebreakers.anticrack.de
Art Of Disassembly	aod.anticrack.de

and the relocators:
www.reverse-engineering.net
www.zerosecurity.de
www.reverse-engeneering.de

So where do we want to go ?

First we want to keep the current network stable and fast. Next I want to calm everything a little down. There are several projects waiting to be finished. “The Art Of Disassembly” will be our next big project we want to finish. Currently it is more a “collection” of interesting material. From 5 coders at the beginning there are now only 2 left – and one has big real life at the moment.

Next I plan to offer a kind of “Archive” site with old but still wanted dead sites. I have some (very) more sites in my archive than the RCE CD contains, but first the other projects. So please don´t ask during the next time for this archive.

Finally I want to get the RCE network a little more closely. Currently there are many small sites, most with useless informations. But sometimes there is the one or other article which is really interesting. Therefore I think about offering webspace for some RCE guys to publish their articles. Why leaving them at damn slow “freenet” accounts? This is a future plan, so please don´t request them now.

Then we want to focus more on the websites with “other” languages than English. Currently there do exist some very good websites like the polish crackmes website. For my opinion we should try to get those sites closer. Therefore I think about (minimum) mirroring the polish crackmes website and some more.

There is one future I don´t like to see:
At the moment it seems that the RCE community is splitting in 2 parts:
English speaking and Russian/polish speaking

It makes me nervous to see many good articles written in Russian or Polish but many people can not read them because they don´t speak these languages. Therefore I think twice to publish them. On the other site it seems that English is not the language of choice for the Russian and Polish people.

I don´t know where we will go here, but we should really think about how we could get closer…

Zero – Main Author