Scan Of The Month 33

Fabrice Desclaux
desclaux at droids-corp dot org

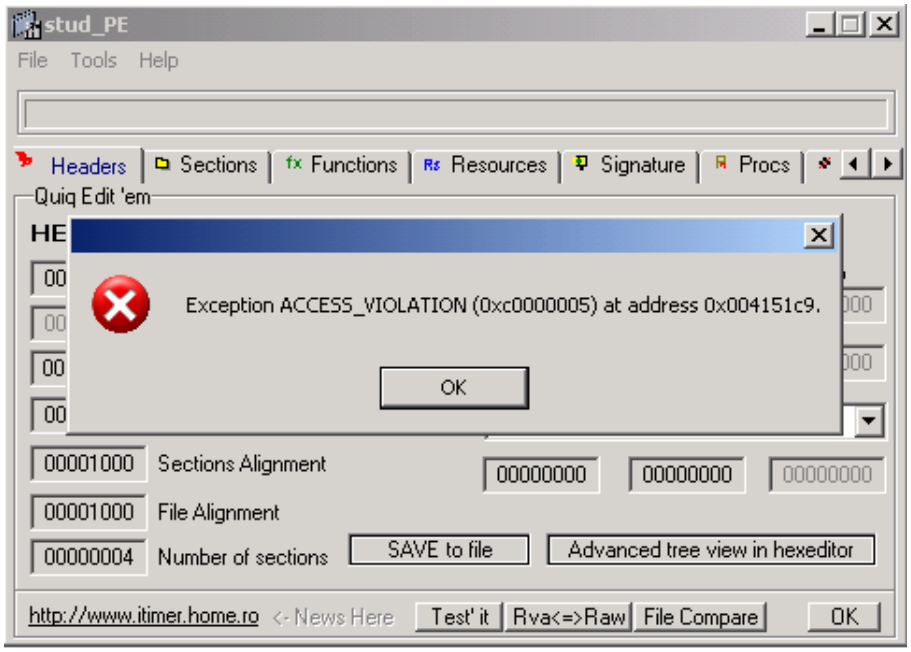
November 25, 2004

1 Analysis

1.1 Introduction

The goal of the challenge this month is to analyze a binary that could be found on the web : the particularity is that this binary has been protected against reverse-engineering. The same techniques could be used in the future to avoid quick analyze of an exploit, or any other kind of malware,. . .

A first glance at the binary using a PE format editor shows us some interesting pieces of information : When we open the binary with *Stud PE*, it pops up an error.

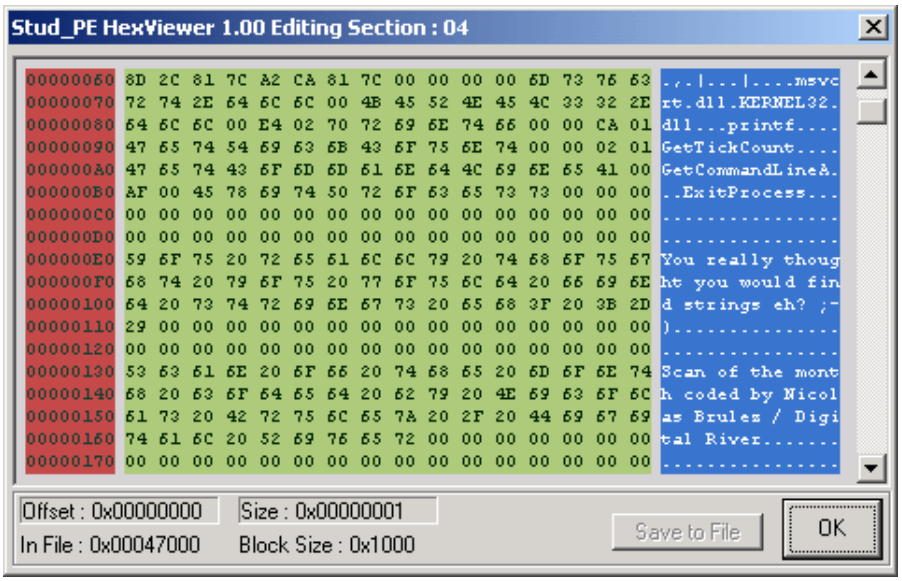


StudPE opens it anyway. Here is the description of the sections in this binary.

We can conclude on some points with these pieces of information : first of all, the entry point is in the section DATA, not in the section CODE. Then we have

No	Name	VirtualSize	VirtualOffset	RawSize	RawOffset	Characteri...
01	CODE	00001000	00001000	00001000	00001000	E0000020
02	DATA	00045000	00002000	00045000	00002000	C0000040
03	NicolasB	00001000	00047000	EFEFADFF	00047000	C0000040
04	.idata	00001000	00048000	00001000	00047000	C0000040

a strange section called NicolasB (the author, Nicolas Brulez) that tells us that the binary will be a real pleasure to reverse :) It's raw size is 0xEFEFADFF. This can be an interesting point : some PE editors or disassemblers could have strange behavior because of that. Note that for each section, the raw size matches the virtual size. This could means that the binary wont grow in memory, so, that the binary is not compackted, (but it may be encrypted :). To finish with the interpretation of these first results, the section .idata in which we have usually the imported fonctions, but here is what we can find in it:



The first idea could be to try to disassemble the binary with *IDA* or something else. The problem is that the goal of the challenge implies that it would be useless. Worst : the previous challenge by the same author (securitech,...) has learnt us that M. Brulez was trying to find bugs in specific disassemblers like IDA to make it crash with special tweaks. Then in special cases, it could be hazardous not to launch the binary, but simply in disassembling it. So the best idea here seems to be to study the binary dynamically and not statically.

1.2 Analysis

The tool we have used is Soft-Ice. Why Soft-ice and not a simpler debugger such as Win32Dasm or OllyDbg ? Because of the author ! It would not be surprising that he has used some hacks in order to execute some code in ring 0 state. So, we need a debugger that enables us to debug such a code, just like Soft-Ice (or

maybe soon, the rr0d `http://rr0d.droids-corp.org/` :). The first thing to do is to install a break point at the entry point of the binary in order to debug it from the first instruction. For that, we will put the opcode CC (int 3) on the file at its entry point. Stud PE shows us its raw offset at 2000. This can be done with an hexadecimal file editor. Then we must ask Soft-Ice to break when it encounters the interruption 3 (`i3here on`). So let's run `0x90.exe` !

As we expected, Soft-Ice pops up. But it pops **after** the execution of the int3 instruction. So to come back to the old real entry point, we need to set back EIP to DE2000 instead of the current DE2001 and to put the old byte that was at this address (we have noted it down before changing it to CC) Then we have the binary in memory, set to the good entry point, and ready to be studied.

The first things found that slow the study of the binary is the *appearence changing code* : this is the name given to code that often jumps to destinations that are in the middle of a mnemonic. So when we look at the asm listing, (which has been done linearly) and we step, the code jumps to the middle of a previous mnemonic and then the next listing has another appearence. This makes us to be very cautious to the code that is really executed.

Another way used to disturb the dynamic study is the use of exception handlers. This is ofted used in order to catch exceptions (division by 0, segmentation fault,...). Windows has a special structure that enables a program to set the address of a handling code to call when the corresponding exception is raised. The structure is written to the dedicated segment and address fs:[0].

So when we step in the code, and if the program commits an error, the exception handler is silently called and can come back to the next instruction, and we don't see the code executed in the handler. Here, the program uses that structure to make special tweak : when those handlers are called, Windows pushes on the stack many things like registers values, floating point registers, EIP, and debug registers. The joke is that if we change on the stack the debug registers, when we are back from the handler, the debug registers keep ower value.

This trick is there to clear over Hardare Breakpoints. In a nutshell, the X86 processor has 8 debug registers. There are used to configure 4 hardware breakpoints. They can be used to break on execution, read or write access. If debuggers make use of hardware breakpoints, execption handlers can erase them, so the debugger become useless.

At this point, we can say byebye to our hardware breakpoints (or try to use counter tricks:). We can notice another problem we have to face with the exception handling code : to make sure that noboddy steps the code, it does some time measurements. So, normally, it could call windows API to ask what time it is before the exception handler is called, then to ask what time it is in the handler. In most cases, the difference is very short. But if someone is stepping the program, its very long. But do not even think that the author will let you monitor those time calls : first it could introduce weaknesses and moreover Soft-Ice freezes Windows time, and then makes those API useless.

The trick used here is to ask directly the time to the processor by calling the mnemonic RDTSC. It puts in the registers EAX and EDX the number of cycles of the processor since it is up (coded then on 64 bits), and even Soft-Ice cannot freeze that. So one call to RDTSC is done before the SEH (Set Exception Handler), one after the exception. Then in the handler, a substraction is done.

A look at the difference between the two values helps to detect a debugguer.

If the program is debugged, it won't exit directly because we could monitor the function exitprocess to look were the program has detected us. Instead, it calls RDTSC once again and uses the value of EAX (the low part of the cycle counter) as the new EIP value. So the program jumps anywhere, and causes in most cases a segmentation fault. So we cannot know where the true error has occured.

To bypass that, the fast way is to look and change the result of the substraction. Another way could be to use the following : RDTSC belongs to a special group of instruction that can be turned into "privilegied instructions" by tweaking the control registers of x86. Then we can make a kind of driver that switches this mode on, and that handles the privilegied exception that each execution of RDTSC throws when done while in ring 3. Then it changes the EAX and EDX to good values, and back as if nothing had happened (sexy isn't it ?).

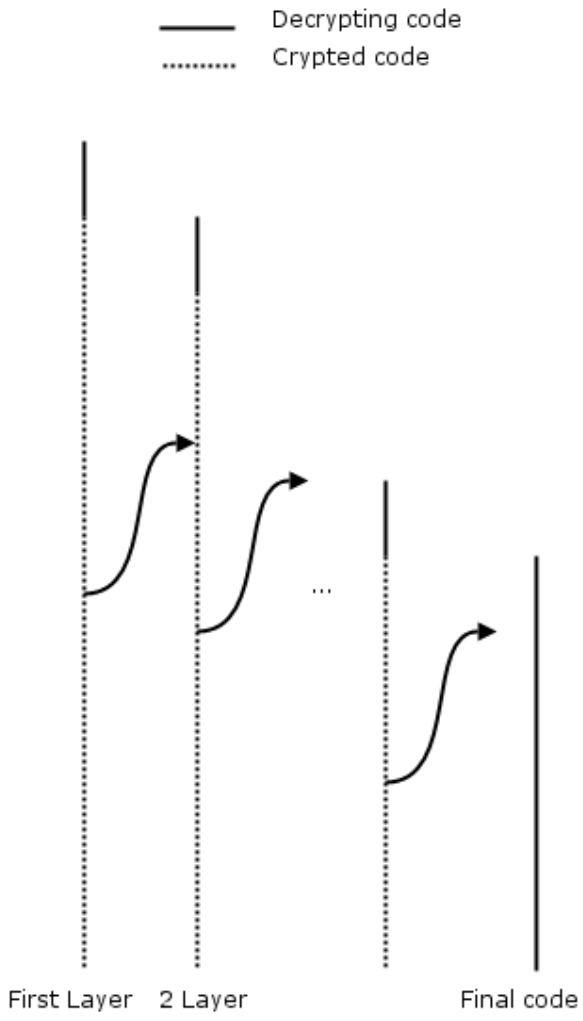
After those checks the other thing that has to be checked to be sure that the program is running on a *clean* system is watching software breakpoints on API functions that will be used in the future. This is done by checking the presence of the opcode CC (int 3) at the beginning of those API (GetcommandlineA,...).

At this point, (at least in theory) the system is supposed to be clean. No hardware breakpoints, no software breakpoints on API functions, no dynamik debugging. So the program can starts unpacking the code :). How do we know that the program is decrypting itself ? Well by looking at what the code is doing : fisrt we are in front of a little loop that cycles 4 times. The first 3 times, it does nothing interesting, and during the last one, the code starts xoring each byte of memory from 0xDE5423 on a range of 0x40F38 with the low part of this counter. Then, it checks if there is no softare breakpoint at the end of the loop (looks for CC at 0xE26367). Then it installs an exception handler, does a page fault, erases hardware breakpoints. Then, the code jumps to the part that has just been deciphered. And it does it again :). Here is a summury of the code :

- executes a loop that randomly decrypts code (first pass or second,...) ;
- makes sure the guy didn't pass the random loop by putting a breakpoint at the end of the loop ;
- installs a handler and erases hardware breakpoints ;
- jump to the next layer.

And so on... This schema is called *crypted layer*. The goal is first to make sure that the environment is clean, second to hide the real code to a disassembler, and third, not to allow the buddy to access to the real code easily.

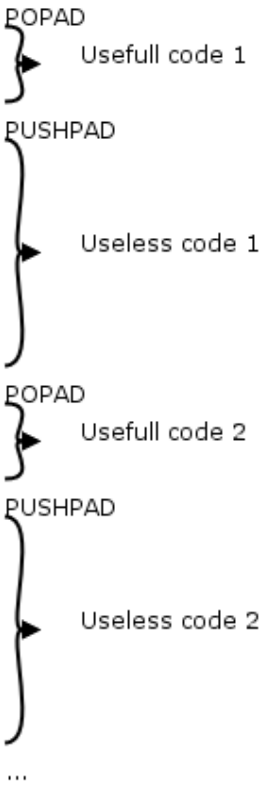
How can we bypass this system ? There are many ways to do so : if the number of layers is not important, we can trace them manualy. If there are too many layers, we can write an "autotracer" : a small program simply does what we would do by hand. Another way is to test it by "timing" attack : a second program that runs the program at a time T, then looks whether we are in a layer or not, then does another test but waits a bit more. This method (I will use a sentence from a friend) is a little sucky, but works in some cases. However, the author may put some malicious code in the middle of the layers, to check this...


Decrypting code
Crypted code
...
First Layer
2 Layer
Final code

The method I have used to bypass the layers is even more sucky : as I saw that the code tests whether there are a software breakpoint on the API GetcommandlineA, I decided to put one after the test has been done. So, I let the program decrypt itself and hope the function will call GetCommandlineA at the beginnning of the real code. And it does so :) (0xDEF952). But we can miss a lot of code with that method, so please be careful, and DONT DO that at home ;). At this point we are IN the real code, but maybe we have missed some important code. So we can try to look back and find the address where the code has just finished to decrypt itself and the real code begins. We can use the address where the API is called to tweak the programs described before.

1.3 The Real code

We know that there is no additionnal layers because the code does not modify itself anymore. So we could here dump the memory to the disk, and then launch IDA for example on the piece of code to study that statically. But my first idea has been to firt look at what the code was doing. So I continued with the debugger. And then another schema appears: the code was constituted by a sequence of useless instructions, then usefull ones, and so on :



The registers are restored, so have "good" values, then the real code operates on those registers, then, registers are saved on the stack. Here, messy code is done that do nothing but mess the man that tries to understand it. So the program restores the registers and so on. Here I stopped in order to find a way

to save time. First, I have thought of a tracer that steps in until it disassembles the POPAD (or something like POPAD, such as REPNZ POPAD,...) then puts a breakpoint, and shows the real code. Then finds the PUSHAD (or an equivalent) and puts a breakpoint... Then, the goal would be just to push the key "run" on the debugger and look at what the real code does. But then I have noticed that the number of messy instructions was always about 172 (maybe a little random value plus 172). And Soft-Ice has a command that autotraces X instructions. So we can assign a Soft-Ice key that autotraces 172 instructions, then we trace with the step in key the real code, then autotrace,...

This works fine. As this method was very fast, I began to study the code : I noticed the code "usually" gets some bytes from ESI, then gets some values from a table indexed by this byte, and to go to a code dependant of this address. Sometimes, the code didn't use the @ESI byte, but the whole 32 bits value, which was used in an addition or something else.. After some operations, a hard coded value is added to ESI, and so on. People who are familiarized with emulators or interpretors have already bounced on their chairs. The code seems to be an interpretor of the table of bytes pointed by ESI. To make a parallel with the x86 processor, ESI is the equivalent of the EIP of the x86, the bytes at ESI are the code to execute. Then the program can be cut out in four parts :

1. the instruction fetcher which loads the bytes of the next instruction ;
2. the instruction decoder that analyses those bytes to know what it means ;
3. the instruction executor that does what the instruction means ;
4. the last one that updates the registers after the instruction has been executed.

So I had chose the method to use to analyse the binary as fast as possible. The goal is to see what does the binary, but not to spend monthes to do it.

The first method that looks sexy is to write our own interpretor for the code. This means reprogram the four parts and to run it on the array of code stored in ESI. This means learn ALL the instructions the code can do, the way they work, the arguments they use,... Sounds a bit time consuming. Note that it seems that IDA full version includes an SDK that enables to write processor packages : it might be possible to write a package for IDA that disassembles the program ? (erf, maybe I should have asked that) The second one I thought consists to "hook" the CORE of the instruction, *i.e.* the third part. Then we can debug by going from instruction to instruction. The drawback of this method is we must find where to put the breakpoints and if we forget one, we will miss a lot. Moreover, malicious code could be executed in the code of one of the other 3 parts, so it might be unsafe to miss them.

The last one was to continue with the help of the short key that passes 172 instructions. Here, it could be very very time consuming.

But how could I chose the good one ? The *simulated* code is at ESI. Moreover, I have noticed that the size of an instruction is between about 3 and 7 bytes. Another thing to notice is that ESI uis only growing *i.e.*, there is no JUMP or CALL equivalent instruction that changed ESI (for the moment). And the last thing was that there was a string in memory about 250 bytes after the current ESI, so the code was about 250 bytes long. I could then try to approximate the number of instructions of this buffer : between 250/3 and 250/7,

i.e. between about 35 and 85 instruction. Then I could evaluate the time it would take by hand (time for one instruction * number of instrcution). Then I could compare it to the evaluation to reprogram an emulator. I decided then to continue by hand :) (with the help of the magic key 172 instrcutions and some breakpoints too).

First I have noticed some informations : ESI is (as I said it) the equivalent of an EIP. After each substraction or addition, a test is done and the byte at 0xDE874A is put to 1 or 0 according to whether the result is null or not. This is in fact a kind of ZER0 flag. Address of the next instruction are in arrays of instrucitons that are a bit messed by operations at run time (address + 0xADD01337,...)

1.4 Results

After the call to GetcommandlineA, the binary looks for the byte 0x20 in the arguments. This looks for a space caracter in the argument line. Then it takes the first four bytes after the space and do some operations on it. Note that if the binary is stored in a directory that contains spaces, the 4 bytes are then part of the directory name...

The operations made on this 32 bits are the following :
32bits arg
+0x1D9BDc45
-0xAD45DFE2
+0x68656c6c
-0x41776169
+0x69767320
-0x6E757473
-0x65683f21
&0xDFFFFFFF

In parallel, other operations are executed on another fixed value, with other hard coded values. At the end of thoses operations, the program tests whether both values are egual (at 0xDFF51B) if not, go out :). So to pass the first test, we must have our value that is egual to 0xDF807499 after the operations. Operations are similar to one addition with 0x914d3068 and a AND with 0xDFFFFFFF. So we conclude that the good value of the 4 caracters of the arguments are "1D3n"

Then the program takes the next two caracters of the arguments, with which it XOR a word (2 bytes) in the code somewhere (we'll have to remerber that in the future).

Then it takes the fifth and the seventh byte of the argument, add them, then sub 0x5A, and compare it to 0x4E at 0xDFC98A. 'a' + 'G' is a orking exemple. (0x61+0x47-0x5A = 0x4E)

Here, the code takes the sixth and the eighth add them, then substract 1 and 0x4E and stores this value. The code then uses this value to XOR the bytes at ESI ! Remerber, this represents the opcode emulated. So in one hand the code is emulated, but moreover the code is decrypting its own opcodes ! The problem is to find the good value that will deXOR correctly the code (actually the code XORs the first byte with the value, then the second with the value-1 and so on...). By the way, all this code is done in a loop emulated that i didn't take

in account when calculating the time it will take in the total study's time. The fact that helps us to imagine the good byte is that the next instruction code emulated is a string: so ESI must absolutely be changed. So the instruction must jump out of there ! Moreover, this means some others instructions that I didn't take in account un the calcul of the time it will take to debug ! But hey, maybe we are near the end of the program :). To find an instruction that works, I decided to do a simple brute force :). I put a breakpoint on the part that updates ESI and tests values. The decrypted value that seems to change ESI is 0x05. So the unXORED one should be 0x42. Then a working exemple can be 'A'+'N'. When the code is correctly decrypted, the buffer XORed lets appear a string in memory : "Exploit for it doesn't matter 1.x Courtesy of Nicolas Brulez".

Then ESI is changed to another location. Here is another piece of code to study... But if we look at ESI closely, we can notice this : do you remember the word XORed by the arguments ? this is it ! So we are in front of another opcode to decrypt correctly. Here I was a bit annoyed. I didn't have a clue on the correct byte. But I decided to test some values: the next opcode bytes seems to be normal code, so it may not be a changing ESI opcode. So i tested 0000, and the opcode parser did a little addition and went on the code. It did on some addition on address and finished with a printf ! The thing seems to work (even if the first operation seems to be out of context...).

Then the code calls Exitprocess. Happy things always have an end.

2 Questions

2.1 Identify and explain any techniques in the binary that protect it from being analyzed or reverse engineered.

The binary uses multiples techniques :

- messy sections that can crash PE analyser or disassemblers is they are buggy ;
- "Appereance Changing code" : code that seems to change at runtime ;
- exception handlers to erase hadware breakpoints and change EIP to loose the reverser ;
- timing mesures to detect tracing ;
- useless code parts ;
- multiple crypted layers ;
- Softare Breakpoint detection on some API functions and loops ;
- Code self modifying depending on arguments ;
- Emulator :).

And all this full of messy code that hides all references such as addresses of libraries or kernel functions or even addresses of the program itself, hardcoded values (0xCC for example),...

2.2 Something uncommon has been used to protect the code from beeing reverse engineered, can you identificate what it is and how it works ?

The original method used to protect the binary is to emulate a custom processor. This is done to prevent people from understanding the behavior of the real code simulated. The way this works is really similar to a real processor : we have an array of bytes representing the opcodes of the program. The program counter is represented by ESI. So a part of the emulator loads those bytes, decodes them and then executes the corresponding instruction. Then the registers are updated and ESI is set to the next instruction. Each instruction emulated has its code messy in order to slower the study.

2.3 Provide a means to "quickly" analyse this uncommon feature.

The easy one is to do some macro to pass messy code and that stops to the "real" instruction core. But the time it takes is a multiple of the number of instructions the code has.

The good one seems to analyse the instruction of the virtual processor, then to code a tool that simulates the custom opcodes.

Maybe we can use IDA full version SDK to write our own disassembler package.

I chose the first solution, and this was not very time consuming because of the number of instructions.

2.4 Which tools are the most suited for analysing such binaries, and why ?

Malwares could be very hazardous to study : the computer that runs such software must be protected. So first the analyser should use an emulator like Bochs or Qemu or a virtualizer like VMware. With those tools we can relatively avoid to affect a real computer.

To analyse the binary code itself, a debugguer (or tracer) is a must : crypted layers cannot be correctly analysed with only a disassembler. But the situation here makes me feel that custom tools should be done and would be efficient if many binary of that kind have to be analyzed. Here the custom emulator should be coded, or at least an auto tracer that could analyze layers of crypted code or here the layer of useless code.

2.5 Identify the purpose (fictitious or not) of the binary.

The goal maybe for an exploit to check if the current user is an "authentificated" one by checking a key. Then it prints "Exploit for it doesn't matter 1.x Courtesy of Nicolas Brulez". So maybe this means the exploit should start. And then the process dies. The binary maybe then a fictitious one that does nothing but diplays some stuff.

But the way it has been coded gives clearly the idea that if the author wants, it could be hard to analyse a 0-day exploit.

2.6 What is the binary waiting from the user ? Please detail how you found it.

The binary requires an argument on the command line : once it has one, it "parses" it to checks some values. Then it uses some other parts as "keys" to decrypt some parts of the code.

So, the command line is parsed in two parts : only the part after the first space found in the command line is usefull. The four first bytes of this part are used to generate a 32 bit value which is compared to another fixed one. So by calculating this value, we can conclude that those bytes must be 1D3n.

Then the binary takes the 5th and 7th byte and adds them, substracts 0x5A and checks if matches with 0x4E. If yes, it takes the 6th and 8th bytes, adds them, substracts 1 and 0x4E and uses the result to XOR a part of the binary. Moreover, the 5th and 6th are used to XOR another part of code. The correct values have been found by studying the actions that produced with multiples tests (sort of little brute force)

Note that a more precise study could avoid that by studying the opcodes corresponding to each instruction.

Thanks

- Droids Corporation – `http://www.droids-corp.org/`

- Team RSTACK – `http://www.rstack.org/`