CodeBreakers Magazine
Security & Anti-Security - Attack & Defense

Volume 1, Issue 2, 2006


Beginners Guide to Basic Linux Anti Anti Debugging Techniques

Schallner, M.
May 2006

Abstract
Anti-debugging techniques are a common method for protecting software applications. Meanwhile such kind of protection tricks are often used, several approaches work against such kind of protection. One known method are anti-anti tricks which circumvent the mentioned protection schemes. This paper confines to techniques and methods used for Linux platform applications, especially dealing with the operation platforms specific tools.

This disclaimer is not meant to sidestep the responsibility for the material we will share with you, but rather is designed to emphasize the purpose of this CodeBreakers Magazine feature, which is to provide information for your own purposes. The subjects presented have been chosen for their educational value. The information contained herein consists of Secure Software Engineering, Software Security Engineering, Software Management, Security Analysis, Algorithms, Virus-Research, Software-Protection and Reverse Code Engineering, Cryptanalysis, White Hat and Black Hat Content, and is derived from authors of academically institutions, commercials, organizations, as well as private persons. The information should not be considered to be completely error-free or to include all relevant information; nor should it be used as an exclusive basis for decision-making. The user understands and accepts that if CodeBreakers Magazine were to accept the risk of harm to the user from use of this information, it would not be able to make the information available because the cost to cover the risk of harms to all users would be too great. Thus, use of the information is strictly voluntary and at the users sole risk.

The information contained herein is not a license, either expressly or impliedly, to any intellectual property owned or controlled by any of the authors or developers of CodeBreakers Magazine. The information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of CodeBreakers Magazine hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the contribution.

ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO CODEBREAKERS MAGAZINE.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF CodeBreakers Magazine BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR PUNITIVE OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO CODEBREAKERS MAGAZINE, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGE.

1 Introduction

This paper is an introduction for anti anti debugging techniques on the Linux OS. It covers the very basic anti debugging techniques as introduced by Silvio Cesare's paper [1] (back in 1999) . As I see those techniques are still used in applications and crackmes, this paper should show a) how easy and outdated those techniques are, and b) explain why ptrace() and objdump are not always your friends, but finally there is always a way. Well, as in the mentioned paper one anti dissassembling trick (or better anti objdump trick) is described I will discuss it here as well. Actually there were two basic tricks used, I will separate them, and describe more detailed.

2 False Disassembly

A common used disassembler used is objdump, or disassembler projects that base on objdumps output. Actually there are several ways how to fool objdump as a dissasembler.

2.1 Jumping into the middle of an instruction

Let's take the following code as example:

```
start:
       jmp    label+1
label: DB     0x90
       mov    eax, 0xf001
```

The above code is not yet the "trick", just to have the visibility of the problem. As behind label there follows a single byte opcode 0x90 (nop), the jmp label+1 is NO problem for objdump, as we did not jump into the middle of an instruction:

```
# objdump -d -M intel anti01
anti01:     file format elf32-i386
```

Disassembly of section .text:

```
08048080 <start>:
 8048080:       e9 01 00 00 00      jmp    8048086 <label+0x1>

08048085 <label>:
 8048085:       90                  nop
 8048086:       b8 01 f0 00 00      mov    eax,0xf001
```

The code was dissassembled correctly. Now when using an instruction which assembles into more than 1 byte objdump will not follow this jump, it will just dissassemble linear from start to end.

```
start:
       jmp    label+1
label: DB     0xE9
       mov    eax, 0xf001

# objdump -M intel -d anti02
anti02:     file format elf32-i386
```

Disassembly of section .text:

```
08048080 <start>:
 8048080:     e9 01 00 00 00      jmp    8048086 <label+0x1>

08048085 <label>:
 8048085:     e9 b8 01 f0 00      jmp    8f48242 <__bss_start+0xeff1b6>
```

So the disassembly is false. objdump ignored the jump destination and dissasembled the instruction directly following the first jmp. As we placed an 0xe9 byte there, objdump displays it also as a jmp instruction. Our mov instruction got "hidden".

2.1.1 How to circumvent this problem

To be able to use objdump you have to manually replace the bogus 0xE9 byte with a hexeditor. Of course this helps only for disassembling. As the file is then modified it could behave different when it checksums itself. A better choice is to use a dissasembler like bastard [2], IDA [3], or any other that does control flow analysis. For example when disassembling the same executable (antia02) with lida [4], the result looks like this:

```
 ---- section .text ----:
08048080  E9 01 00 00 00     jmp      Label_08048086
                                 ; (08048086)
                                 ; (near + 0x1)
08048085  DB E9

Label_08048086:
08048086  B8 01 F0 00 00     mov      eax, 0xF001
                                 ; xref ( 08048080 )
```

Which is correct, so using the right tools you would not even recognize a trick here.

2.2 Runtime calculation of destination address

Another trick, to fool even control flow disassemblers is to calculate the destination of jumps during runtime. For doing so, the current EIP is to be retreived and then the difference to the address of the destination from current EIP is added. To retrieve EIP, the common call+pop "technique" is used, as the call instruction stores the return address on the stack, which nobody prevents us to pop it into a register. Here a scheme of a more advanced example than above:

```
; -----------------------------------------------------------------------------
       call   earth+1
   Return:
                    ; x instructions or random bytes here             xb
   earth:           ; earth = Return + x
       xor    eax, eax; align disassembly, using single byte opcode           1b
       pop    eax    ; start of function: get return address ( Return )  1b
                    ; y instructions or random bytes here             yb
       add    eax, x+2+y+2+1+1+z  ; x+y+z+6                    2b
       push eax     ;                                          1b
       ret          ;                                          1b
                    ; z instructions or random bytes here              zb
   ; Code:
                    ; !! Code Continues Here !!
; -----------------------------------------------------------------------------
```

Now an implementation could look like below. I have used for x and z just one byte, again E9, as it eats so many bytes. For Code I have chosen 3 nops, as they are good visible in the outputs you will see:

```
; ----------------------------------------------------------------------------
; antia.s
call         earth+1

earth: DB 0xE9               ; 1    <--- pushed return address,
                         ;                E9 is opcode for jmp to disalign disas-
                         ;           sembly
             pop     eax     ; 1    hidden
             nop             ; 1
             add     eax, 9  ; 2    hidden
             push    eax     ; 1    hidden
             ret             ; 1    hidden

       DB 0xE9               ; 1    opcode for jmp to misalign disassembly

Code:      ; code continues here    <--- pushed return address + 9
             nop
             nop
             nop
             ret
; ----------------------------------------------------------------------------
```

I used nasm -f elf antia.s to create the object file. Of course objdump will be fooled already by the first trick "calling earth+1".

```
# objdump -d antia.o
antia.o:     file format elf32-i386
```

Disassembly of section .text:

```
00000000 <earth-0x5>:
   0:   e8 01 00 00 00          call   6 <earth+0x1>

00000005 <earth>:
   5:   e9 58 90 05 09          jmp    9059062 <earth+0x905905d>
   a:   00 00                   add    %al,(%eax)
   c:   00 50 c3                add    %dl,0xffffffc3(%eax)
   f:   e9 90 90 90 c3          jmp    c39090a4 <earth+0xc390909f>
```

As result you can see our code (3 nops) is fully hidden here at address 0xf. But not only that, also our calculation of EIP is totally hidden for objdump. Indeed this disassembly is totally different than what was coded. But our example not only was good for fooling objdump. Now look, what IDA outputs:

```
.text:08000000 ; Segment permissions: Read/Execute
.text:08000000 _text           segment para public 'CODE' use32
.text:08000000                 assume cs:_text
.text:08000000                 ;org 8000000h
.text:08000000                 assume es:nothing, ss:nothing, ds:_text,
.text:08000000                          fs:nothing, gs:nothing
.text:08000000                 dd 1E8h
.text:08000004 ; -------------------------------------------------------------
.text:08000004                 add     cl, ch
.text:08000006                 pop     eax
.text:08000007                 nop
.text:08000008                 add     eax, 9
.text:0800000D                 push    eax
.text:0800000E                 retn
.text:0800000E ; -------------------------------------------------------------
```

```
.text:0800000F                 dd 909090E9h
.text:08000013 ; ---------------------------------------------------------------
.text:08000013                 retn
.text:08000013 _text           ends
.text:08000013
.text:08000013
.text:08000013                 end
```

Well, I do not know why, but IDA did not like the call +1 instruction. After all at least it shows the calculation of EIP, but not from where was called, so you can not immediately say where the code finally will continue after the retn instruction. I have loaded the same file into lida as well:

```
 ---- section .text ----:
08048080  E8 01 00 00 00         call       Function_08048086
                                  ; (08048086) ; (near + 0x1)
08048085  DB E9
Function_08048086:
08048086  58                     pop        eax  ; xref
                                                      ;( 08048080 )
08048087  90                     nop
08048088  05 09 00 00 00         add        eax, 0x9
0804808D  50                     push       eax
0804808E  C3                     ret
0804808F  E9 90 90 90 C3         jmp        CB951124
                                  ;(near - 0x3C6F6F70)
08048094  DB 00, 54, 68, 65, 20, 4E, 65, 74, 77, 69, 64, 65, 20, 41, 73, 73
```

Actually this looks better. Until the ret instruction it is exactly what we have coded. But still the last hurdle is that no disassembler can tell where the code after the ret instruction will continue, until it does code emulation. In this example we could see the cross-reference to the call from address 08048080. So we could calculate the return address and tell the disassembler to start at address 08048090.

2.2.1 How to circumvent this trick

Actually there is no automated way which is 100% accurate. Possibly when a disassembler does code emulation it could do a complete correct disassembly. In reality this is not a big problem, as when using interactive disassemblers you can tell the disassembler where the parts of the code start. Also while debugging you would see what is really going on. This is why I would call those techniques "anti disassembling" techniques.

3 Detecting Breakpoints

The first technique described by Silvio Cesare is really easy to circumvent:

```
// -- antibreakpoint.c --
void foo()
{
        printf("Hello\n");
}

int main()
{
        if ((*(volatile unsigned *)((unsigned)foo) & 0xff) == 0xcc) {
                printf("BREAKPOINT\n");
                exit(1);
        }
        foo();
}
// -- EOF --
```

As described, gdb sets breakpoints by replacing the byte at the address to break with an int 3 Opcode, which is 0xcc. So it is easy for a program to check addresses for 0xcc presence, as above. When running the program, it says "Hello" :), also when running it in gdb. Actually if we place a breakpoint at the function foo, and run, gdb will not break and we will see the output "BREAKPOINT".

```
# gdb ./x
GNU gdb 6.0-2
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-linux-gnu"...Using host libthread_db library "/
lib/tls/libthread_db.so.1".

gdb> bp foo
Breakpoint 1 at 0x804838c
gdb> run
BREAKPOINT

Program exited with code 01.
```

3.1 How to circumvent this trick

Well this is also very easy. To avoid this problem but still be able to break into foo, just look at the disassembly and simply choose your breakpoint not exactly at the functions entry point:

```
0804838c <foo>:
 804838c:       55                      push   ebp
 804838d:       89 e5                   mov    ebp,esp
 804838f:       83 ec 08                sub    esp,0x8
 8048392:       83 ec 0c                sub    esp,0xc
 8048395:       68 c8 84 04 08          push   0x80484c8
 804839a:       e8 0d ff ff ff          call   80482ac <_init+0x38>
 804839f:       83 c4 10                add    esp,0x10
 80483a2:       c9                      leave
 80483a3:       c3                      ret
```

So we can set the breakpoint on all those addresses != 0x804838c. I should mention that the problem with this anti breakpoint technique is not in circumventing it, but to detect it. In this example obviously you will realize it, because the program tells it. In real life it would probably not print something out, but your breakpoint will simply not break. To find the comparison you could either search the disassembly for your address to break for example:

```
# objdump -M intel -d x | grep 804838c
0804838c <foo>:
 804838c:       55                      push   ebp
 80483b4:       a1 8c 83 04 08          mov    eax,ds:0x804838c
 80483df:       e8 a8 ff ff ff          call   804838c <foo>
```

and examine the code after 80483b4. But this potentially could not help you, since the address could be calculated as well. You could also use a short perl script to find all occurrences of an operand 0xcc like

```
#!/usr/bin/perl

while(<>)
{
  if($_ =~ m/([0-9a-f][4]:\s*[0-9a-f \t]*.*0xcc)/ ){ print; }
}
```

And run it as a filter for objdump:

```
# objdump -M intel -d x | ./antibp.pl
 80483be:       3d cc 00 00 00          cmp    eax,0xcc
```

which will give you the address of the compare. Now you can either change the byte 0xcc to 0x00 or nop the instruction out, or do anything you like. Should the code check itself for any changes in the function where the compare is done (in this example main()), changing the 0xcc byte would be detected. It is possible, that not only the functions entrypoint, but the whole function is being checked for 0xcc bytes in a loop. Therefore you can manually place an ICEBP (0xF1) instruction into foo() with a hexeditor or gdb instead of an INT 3. ICEBP also causes gdb to break. And no 0xCC byte is detected, of course.

4 Detecting debugging

```
// -- antiptrace.c --
int main()
{
        if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
                printf("DEBUGGING... Bye\n");
                return 1;
        }
        printf("Hello\n");
        return 0;
}
// -- EOF --
```

This program checks if it could let it debug itself, by trying to set a debugging request to itself. Now if the program is being debugged by gdb, this call to ptrace() fails, as there can only be one debugger. The failure of the call indicates the program it is being debugged.

4.1 How to circumvent this trick (Method 1)

Obviously as this check is only working for debuggers using ptrace(), any debugger not using ptrace() can be used. Alternatively one can patch/wrap the ptrace() function which is a more advanced task. Easier is to either "nop out" the ptrace() call or the checking afterwards. To comfortably be able to do so, we need to find where this ptrace() check is done. If the executable in the unlikely case was compiled without the -s switch (-s Remove all symbol table and relocation information from the executable) then this is very easy:

```
# objdump -t test_debug | grep ptrace
080482c0       F *UND*  00000075              ptrace@@GLIBC_2.0
```

So ptrace is called by the address 080482c0 in this executable. Simply typing:

```
# objdump -d -M intel test_debug |grep 80482c0
 80482c0:       ff 25 04 96 04 08       jmp    ds:0x8049604
 80483d4:       e8 e7 fe ff ff          call   80482c0 <_init+0x28>
```

Shows us where ptrace gets called. Now we can do whatever we like. But before, what to do if the -s option was used while compiling? Then objdump does not show us the output as above. For the above example we can do that easily by using gdb:

```
# gdb test_debug
GNU gdb 6.0-2
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

gdb> bp ptrace
Breakpoint 1 at 0x80482c0
gdb> run
Breakpoint 1 at 0x400e02f0
_______________________________________________________________________________
     eax:00000000 ebx:40143218  ecx:00000001  edx:4014449C     eflags:00200246
     esi:BFFFF5E4 edi:BFFFF570  esp:BFFFF53C  ebp:BFFFF558     eip:400E02F0
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t s Z a P c
[007B:BFFFF53C]---------------------------------------------------------[stack]
BFFFF56C : C4 A9 00 40  18 32 14 40 - 00 00 00 00  70 F5 FF BF ...@.2.@....p...
BFFFF55C : A0 BE 03 40  01 00 00 00 - E4 F5 FF BF  EC F5 FF BF ...@............
BFFFF54C : 00 00 00 00  00 00 00 00 - 40 44 01 40  B8 F5 FF BF ........@D.@....
BFFFF53C : D9 83 04 08  00 00 00 00 - 00 00 00 00  01 00 00 00 ................
[007B:BFFFF5E4]---------------------------------------------------------[ data]
BFFFF5E4 : 8A F7 FF BF  00 00 00 00 - B1 F7 FF BF  C0 F7 FF BF ................
BFFFF5F4 : D3 F7 FF BF  E4 F7 FF BF - F7 F7 FF BF  0B F8 FF BF ................
[0073:400E02F0]---------------------------------------------------------[ code]
0x400e02f0 <ptrace>:    push   %ebp
0x400e02f1 <ptrace+1>:  mov    %esp,%ebp
0x400e02f3 <ptrace+3>:  sub    $0x10,%esp
0x400e02f6 <ptrace+6>:  mov    %edi,0xfffffffc(%ebp)
0x400e02f9 <ptrace+9>:  mov    0x8(%ebp),%edi
0x400e02fc <ptrace+12>: mov    0xc(%ebp),%ecx
------------------------------------------------------------------------------

Breakpoint 1, 0x400e02f0 in ptrace () from /lib/tls/libc.so.6
```

What we have done is set a breakpoint on ptrace() itself. Now after typing pret we are back in the test_debug executable:

```
gdb> prêt
_______________________________________________________________________________
     eax:FFFFFFFF ebx:40143218  ecx:FFFFFFFF  edx:FFFFFF00     eflags:00200246
     esi:BFFFF5E4 edi:BFFFF570  esp:BFFFF540  ebp:BFFFF558     eip:080483D9
     cs:0073  ds:007B  es:007B  fs:0000  gs:0033  ss:007B    o d I t s Z a P c
[007B:BFFFF540]---------------------------------------------------------[stack]
BFFFF570 : 18 32 14 40  00 00 00 00 - 70 F5 FF BF  B8 F5 FF BF .2.@....p.......
BFFFF560 : 01 00 00 00  E4 F5 FF BF - EC F5 FF BF  C4 A9 00 40 ...............@
BFFFF550 : 00 00 00 00  40 44 01 40 - B8 F5 FF BF  A0 BE 03 40 ....@D.@.......@
BFFFF540 : 00 00 00 00  00 00 00 00 - 01 00 00 00  00 00 00 00 ................
[007B:BFFFF5E4]---------------------------------------------------------[ data]
BFFFF5E4 : 8A F7 FF BF  00 00 00 00 - B1 F7 FF BF  C0 F7 FF BF ................
BFFFF5F4 : D3 F7 FF BF  E4 F7 FF BF - F7 F7 FF BF  0B F8 FF BF ................
[0073:080483D9]---------------------------------------------------------[ code]
0x80483d9 <main+29>:    add    $0x10,%esp
0x80483dc <main+32>:    test   %eax,%eax
0x80483de <main+34>:    jns    0x80483fa <main+62>
```

```
0x80483e0 <main+36>:    sub    $0xc,%esp
0x80483e3 <main+39>:    push   $0x80484e8
0x80483e8 <main+44>:    call   0x80482e0
------------------------------------------------------------------------------
0x080483d9 in main ()
```

From here we also see by where we landed, the return address from ptrace(). Now we can patch the file and nop out the jns instruction, or change the eax register during runtime.

```
gdb> set $eax=0
gdb> c
everything ok

Program exited with code 016.
______________________________________________________________________________
No registers.
gdb>
```

So everything is OK although we were debugging. That is fine!

4.2 How to circumvent this trick (Method 2)

Another option to bypass the debugger would be to write your own ptrace() function, which as a minimum always returns 0. Then the LD_PRELOAD environment variable can be set to point the executable to the own ptrace() function. Example: First we make a test executable that implements the anti debugging technique:

```
// -- antiptrace.c --
int main()
{
  if (ptrace(0,0,1,0) < 0) {
          printf("DEBUGGER PRESENT!\n");
          exit(1);
  }
  printf("Hello World!\n");
}
// -- EOF --
```

compile it with # `gcc antiptrace.c -o antiptrace.`

Then we will use a simple ptrace() function and build a shared object of it:

```
// -- ptrace.c --
int ptrace(int i, int j, int k, int l)
{
        printf("  PTRACE CALLED!\n");
}
// -- EOF --
```

Compile it with

```
# gcc -shared ptrace.c -o ptrace.so
```

Running the program, it prints:

```
# ./antiptrace
Hello World!
```

Running it in gdb, it prints:

```
# gdb ./antiptrace
GNU gdb 6.0-2
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

gdb> run
DEBUGGER PRESENT!

Program exited with code 01.
gdb>
```

Now we can use our own ptrace function by setting the environment variable LD_PRELOAD for our executable. In gdb this is done by:

```
gdb> set environment LD_PRELOAD ./ptrace.so
gdb> run
  PTRACE CALLED!
Hello World!

Program exited with code 015.
gdb>
```

We can see the executable did not detect the debugger and our ptrace() function was called.

5 Conclusions

These anti debugging (and anti anti) techniques are the very basic ones, all relying on gdb is used as debugger, and are easy to defeat as you can see.

6 References

1. Cesare, S., *Linux Anti Debugging Techniques - Fooling the Debugger*. 1999.
2. mammon_, et al., *bastard - The Bastard Disassembly Environment*. 2002.
3. Datarescue, *IDA Pro - Disassembler and Debugger*. 2004.
4. Schallner, M., *lida - Linux Interactive DisAssembler*. 2004.
5. Mammon, *Mammons Tales*, in *Assembly Programming Journal*. 2004.
6. Unknown, *LinIce - Linux Debugger*, http://www.linice.com/.
7. Unknown, *The Dude*, http://the-dude.sourceforge.net/.