Linux 反跟踪技术(迷惑调试器)

作者：Silvio Cesare
翻译：王筱[AKA]

- Silvio Cesare <silvio@big.net.au>
- http://www.big.net.au/~silvio
- http://virus.beergrave.net
- January 1999

目录

概述

这篇文章介绍 x86 平台上的反跟踪技术(虽然这些技术不是 x86 平台所特有的)。这一技术可以迷惑，终止和改变对目标程序的跟踪。这一技术可以用来开发病毒和那些需要被保护的软件。

反汇编失效

这是一流的技术可以使反汇编所列出的代码失效。这是用跳转到指令的中间来做到的。实际代码的开始点是在指令的中间，但是反汇编是用完整的代码来进行处理，因此后面断续的反汇编就不能还原真实的代码。

```
        jmp antidebug1 + 2
antidebug1:
.short 0xc606
        call reloc
reloc:
        popl %esi
        jmp antidebug2
antidebug2:
        addl $(data - reloc),%esi
        movl 0(%esi),%edi
        pushl %esi
        jmp *%edi
```

```
data:
        .long 0

--

$ objdump -d a.out

.
.
.

 8048340:       55              pushl  %ebp
 8048341:       89 e5           movl   %esp,%ebp
 8048343:       eb 02           jmp    0x8048347
 8048345:       06              pushl  %es
 8048346:       c6 e8 00        movb   $0x0,%al
 8048349:       00 00           addb   %al,(%eax)
 804834b:       00 5e eb        addb   %bl,0xffffffeb(%esi)
 804834e:       00 81 c6 0f 00  addb   %al,0xfc6(%ecx)
 8048353:       00
 8048354:       00 8b 7e 00 56  addb   %cl,0xff56007e(%ebx)
 8048359:       ff
 804835a:       e7 00           outl   %eax,$0x0
 804835c:       00 00           addb   %al,(%eax)
 804835e:       00 89 ec 5d c3  addb   %cl,0x90c35dec(%ecx)
 8048363:       90
 8048364:       90              nop

.
.
.
```

侦测断点

一个断点是用一个 int3 **中断例程**(0xcc)**重写其断点地址来定义的。如果一个程序正被追踪**(**查看** ptrace man **页**)**那么一个** int3 **中断将使其进程停止。这一行为将使其调试父进程取得控制权。继续执行将进入调试器将原有的中断例程重设后的中断例程。因此侦测一个断点，简单的做法是检查** int3 **中断的中断例程。另一种方法是检查代码映像和。如果检查代码映像和失败，则代码已被修改过并且一个断点可能以被植入其中。**

```
void foo()
{
        printf("Hello\n");
}
```

```
int main()
{
        if ((*(volatile unsigned *)((unsigned)foo + 3) & 0xff) == 0xcc) {
                printf("BREAKPOINT\n");
                exit(1);
        }
        foo();
}

--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...done.
(gdb) break foo
Breakpoint 1 at 0x8048373: file break.c, line 3.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
BREAKPOINT

Program exited with code 01.
(gdb) quit
$ ./a.out
Hello
$
```

设置无效断点

作为一个简单的开始， 一个断点是由用 int3 中断代码重写其地址来建立的。设置一个无效断点的简单的做法是插入一个 int3 到代码中。这也就产生一个 SIGTRAP 信号，因为我们设置了信号处理的代码，所以我们可以在断点后继续我们的进程。

```
#include <signal.h>

void handler(int signo)
{
}

int main()
{
    signal(handler, SIGTRAP);
```

```
__asm__("
    int3
");
    printf("Hello\n");
}

--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x80483c3 in main ()
(gdb) c
Continuing.
Hello

Program exited with code 06.
(gdb) quit
$ ./a.out
Hello
$
```

侦测跟踪

这是一个一流的侦测调试器或跟踪器的技术例如 strace **或** ltrace **被使用在目标程序上时。这一技术使用的前提是一个** ptrace[PTRACE_TRACEME]**不能在一个进行中被多次成功调用。所有的调试器和跟踪程序都使用这一技术来调试一个进程。**

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

```
--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...done.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
DEBUGGING... Bye

Program exited with code 01.
(gdb) quit
$ ./a.out
Hello
$
```

附件：英文原文

LINUX ANTI-DEBUGGING TECHNIQUES
(FOOLING THE DEBUGGER)

- Silvio Cesare <silvio@big.net.au>
- http://www.big.net.au/~silvio
- http://virus.beergrave.net
- January 1999

TABLE OF CONTENTS

INTRODUCTION

This article describes anti debugger techniques for x86/Linux (though some of these techniques are not x86 specific). That is techniques to either fool, stop, or modify the process of debugging the target program. This can be useful to the development of viruses and also to those implementing software protection.

FALSE DISASSEMBLY

This elegant technique produces false disassembly when listed. It produces this by jumping into the middle of instruction. The real code starts in the middle of this instruction, but the disassembly uses the entire instruction and thus continues disassembly not alligned to the real assembly.

```
        jmp antidebug1 + 2
antidebug1:
.short 0xc606
        call reloc
reloc:
        popl %esi
```

```
        jmp antidebug2
antidebug2:
        addl $(data - reloc),%esi
        movl 0(%esi),%edi
        pushl %esi
        jmp *%edi
data:
        .long 0

--

$ objdump -d a.out

.
.
.

 8048340:       55              pushl  %ebp
 8048341:       89 e5           movl   %esp,%ebp
 8048343:       eb 02           jmp    0x8048347
 8048345:       06              pushl  %es
 8048346:       c6 e8 00        movb   $0x0,%al
 8048349:       00 00           addb   %al,(%eax)
 804834b:       00 5e eb        addb   %bl,0xffffffeb(%esi)
 804834e:       00 81 c6 0f 00  addb   %al,0xfc6(%ecx)
 8048353:       00
 8048354:       00 8b 7e 00 56  addb   %cl,0xff56007e(%ebx)
 8048359:       ff
 804835a:       e7 00           outl   %eax,$0x0
 804835c:       00 00           addb   %al,(%eax)
 804835e:       00 89 ec 5d c3  addb   %cl,0x90c35dec(%ecx)
 8048363:       90
 8048364:       90              nop

.
.
.


DETECTING BREAKPOINTS
---------------------
A breakpoint is defined by overwriting the breakpoint address with an int3
opcode (0xcc).  If a program is being traced (man ptrace) then an int3 will
cause the process to stop.  This is when the parent process debugging takes
over control.  To continue processing it is up to the debugger to overwrite
the int3 opcode with the original opcode.  Thus to detect a breakpoint, the
```

program simply has to check for an int3 opcode. Another solution is to checksum the code image. If the checksum fails, the code has been modified, and a breakpoint is probably the culprit.

```
void foo()
{
        printf("Hello\n");
}

int main()
{
        if ((*(volatile unsigned *)((unsigned)foo + 3) & 0xff) == 0xcc) {
                printf("BREAKPOINT\n");
                exit(1);
        }
        foo();
}

--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...done.
(gdb) break foo
Breakpoint 1 at 0x8048373: file break.c, line 3.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
BREAKPOINT

Program exited with code 01.
(gdb) quit
$ ./a.out
Hello
$
```

SETTING UP FALSE BREAKPOINTS

As stated earlier, a breakpoint is created by overwriting the address with an int3 opcode (0xcc). To setup a false breakpoint then we simply insert an int3 into the code. This also raises a SIGTRAP, and thus if our code has a signal handler we can continue processing after the breakpoint.

```
#include <signal.h>

void handler(int signo)
{
}

int main()
{
    signal(handler, SIGTRAP);
__asm__("
    int3
");
    printf("Hello\n");
}

--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x80483c3 in main ()
(gdb) c
Continuing.
Hello

Program exited with code 06.
(gdb) quit
$ ./a.out
Hello
$
```

DETECTING DEBUGGING

This is an elegant technique to detect if a debugger or program tracer such as strace or ltrace is being used on the target program. The premise of this technique is that a ptrace[PTRACE_TRACEME] cannot be called in succsession more

```
than once for a process.  All debuggers and program tracers use this call to
setup debugging for a process.

int main()
{
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
        printf("DEBUGGING... Bye\n");
        return 1;
    }
    printf("Hello\n");
    return 0;
}

--

$ gdb
GDB is free software and you are welcome to 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.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc.
(gdb) file a.out
Reading symbols from a.out...done.
(gdb) run
Starting program: /home/silvio/src/antidebug/a.out
DEBUGGING... Bye

Program exited with code 01.
(gdb) quit
$ ./a.out
Hello
$
```