Debugging Session : Anti-Anti-ptrace() or Fooling The Debugger Fooler

DarKPhoeniX - [French Reverse Engineering Team]

30 août 2005

Contents

1 Introduction

Dans le domaine du reverse engineering, et plus particulièrement dans l'analyse de binaires éxecutables protegés, on distingue les techniques "offensives" et "défensives". Le reverser essaie simplement d'étudier, par analyse statique et dynamique, le code machine éxecuté par une application. De l'autre bord, la defense consistera a tout faire pour ralentir sa tâche, en utilisant des techniques d'anti-reversing consistant a brouiller toute analyse statique ou dynamique (anti-debug, false disassembly, etc...).
Un anti-debugging bien connu sous la plate-forme UN*X est l'utilisation de l'appel système ***ptrace()***.

*long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);*

Celui-ci permet en temps normal de debugger un processus, mais il peut etre utiliser à des fins d'anti-reversing dans la mesure ou un processus en cours de debugging ne pourra pas se ***ptrace()***lui-même. Autrement dit, un processus pourra tenter de se ***ptrace()***lui-même et ainsi vérifier s'il n'est pas soumis â un debugger.
L'article présent traite des méthodes pour contourner cette protection.

2 Exemple de binaire protégé

Commencons par créer un simple éxecutable en langage C qui compare le premier argument passé avec une constante.

Listing 1: Simple binaire non protégé

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const char password[] = "foobar";

int main(int argc, char **argv) {
        int ret = 0;

        if ( argc < 2 ) {
                printf("Usage: %s <password>\n",argv[0]);
                return 0;
                }

        ret = strcmp(argv[1],password);
        if ( ret == 0 )
                printf("Access granted!\n");
        else
                printf("Access denied!\n");

        return ret;

}
```

```
$ ./binary barfoo
Access denied!
$ ./binary foobar
Access granted!
```

A présent, ajoutons à ceci de quoi détecter un éventuel debugging.

Listing 2: Detection de ***ptrace()***

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>

const char password[] = "foobar";

int main(int argc, char **argv) {
```

```
        int ret = 0;

        if ( ptrace(PTRACE_TRACEME,0,1,0) < 0 ) {
                printf("Are you trying to debug me ?\n");
                return 1;
                }

        if ( argc < 2 ) {
                printf("Usage: %s <password>\n",argv[0]);
                return 0;
                }

        ret = strcmp(argv[1],password);
        if ( ret == 0 )
                printf("Access granted!\n");
        else
                printf("Access denied!\n");

        return ret;

}
```

$ **./antiptrace**
Usage : ./antiptrace <password>
$ **gdb -q ./antiptrace**
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
gdb>**r**
Starting program : /home/judecca/antiptrace
(no debugging symbols found)...(no debugging symbols found)...Are you trying to debug me ?

Program exited with code 01.

En debuggant le programme, gdb utilise l'appel système ***ptrace()***. Le programme le détecte alors et quitte immédiatement.

3 Contourner la protection

3.1 Méthode 1 : Patching

La méthode la plus simple et la moins *propre* consiste à simplement patcher physiquement l'appel à ***ptrace()***dans l'ELF. Bien entendu cette méthode n'est pas conseillée étant donné que le programme peut facilement détecter ce changement en effectuant un contrôle d'intégrité.
De plus, si le programme modifie certaines parties de son code dynamiquement (s'il est packé par exemple) tout patching direct devient impossible.
On recherche simplement l'appel à ***ptrace()***dans le binaire. Un simple *objdump -t* permet de le retrouver rapidement.

```
push 0
push 1
push 0
push 0
call sub_80482FC ; ptrace()
add esp, 10h
test eax, eax
jns short loc_8048424 ; positif ?
sub esp, 0Ch
push offset aAreYouTryingTo ; "Are you trying to debug me ?\n"
call sub_804831C
```

On repère rapidement le passage des arguments à ***ptrace()***. On *nop* le saut conditionnel et c'est terminé. On pourrait modifier aussi dynamiquement le contenu de EAX ou du flag SF avant le test et le jns. Un outil existe pour patcher automatiquement un binaire en effectuant un tracing sur celui-ci : `http://www.packetstormsecurity.org/groups/electronicsouls/0x4553_Exorcist.tar.gz`

3.2 Méthode 2 : Hijacking *ptrace()*

La méthode la plus efficace à mon goût est celle consistant à intercepter les appels ***ptrace()***afin d'en modifier la valeur de retour.
Pour cela, nous allons crér une librairie dynamique qui surchargera la fonction ***ptrace()***avant la résolution des fonctions de l'ELF par le linker dynamique ld.
Cette bibliotèque sera donc chargée et linkée en priorité grâce à l'utilisation de la variable d'environnement LD_PRELOAD. Notre librairie contiendra la définition d'une nouvelle fonction ***ptrace()***qui sera chargée d'handler les éventuels appels à la véritable fonction ***ptrace()***.
Voici le code de l'ELF qui sera chargé dynamiquement.

Listing 3: Hooking de ***ptrace()***

```
#include <unistd.h>
#include <stdio.h>
#include <linux/ptrace.h>

/* notre fonction hook */
long ptrace(int request, int pid, void *addr, void *data) {

        printf("ptrace() has been invoked!\n");

        /* le processus essaie de se ptrace ? */

        if ( request == PTRACE_TRACEME )
                printf("This process is trying to ptrace() itself...\n");
        /* else {
                __asm__ __volatile__ (
                        "movl 20(%ebp), %esi\n"
                        "movl 16(%ebp), %edx\n"
                        "movl 12(%ebp), %ecx\n"
                        "movl 8(%ebp), %ebx\n"
                        "movl $0x1A, %eax\n"
                        "int $0x80\n"
                        ); }     */
                return 0; // never fail
}
```

$ **gcc -Wall -c antiantiptrace.c -o antiantiptrace.o**
$ **gcc -shared -o antiantiptrace.so antiantiptrace.o**
$ **gdb -q ./antiptrace**
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
gdb>**set args foobar**
gdb>**r**
(no debugging symbols found)...(no debugging symbols found)...
Are you trying to debug me?
Program exited with code 01.
gdb>**set environment LD_PRELOAD ./antiantiptrace.so**
gdb>**r**
(no debugging symbols found)...(no debugging symbols found)...(no debugging symbols found)...
ptrace() has been invoked!
This process is trying to ptrace() itself...
Access granted!
Program exited normally.

Le programme antiptrace appelle à présent la fonction ***ptrace()***contenue dans notre librairie dynamique en croyant appeler celle contenue dans la libc. Cette fonction renvoyant toujours *0*, le programme est dupé et continue de s'éxecuter normalement ce qui permet de continuer l'analyse.

3.3 Méthode 3 : Hooking the syscall

Cette dernière protection n'est pas efficace dans tous les cas : elle ne fait qu'intercepter l'appel à la fonction ***ptrace()***contenue dans la libc. Or ***ptrace()***est en réalité une fonction kernel accessible via l'appel système 26.
Autrement dit, un programme pourra directement accédé à ***ptrace()***via l'utilisation de l'interruption logicielle *0x80*.

Listing 4: Appel de ***ptrace()***via syscall

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>

const char password[] = "foobar";

int main(int argc, char **argv) {
        int ret = 0;

        __asm__ (
                "xorl %ebx, %ebx\n"       // %ebx = 0
                "movl %ebx, %ecx\n"
                "movl %ebx, %edx\n"       // %edx = 0
                "incl %ecx\n"             // %ecx = 1
                "movl $0x1A, %eax\n"      // ptrace
                "int $0x80\n"             // syscall
                "movl %eax, -4(%ebp)\n" // addr de ret sur la stack
                );
        if ( ret < 0 ) {
                printf("Are you trying to debug me ?\n");
                return 1;
        }

        if ( argc < 2 ) {
                printf("Usage: %s <password>\n",argv[0]);
                return 0;
                }

        ret = strcmp(argv[1],password);
        if ( ret == 0 )
                printf("Access granted!\n");
        else
                printf("Access denied!\n");

        return ret;

}
```

En essayant de réutiliser notre librairie dynamique présentée en **3.2**, nous obtenons :
gdb>**r**
(no debugging symbols found)...(no debugging symbols found)...
Are you trying to debug me ?
gdb>**set environment LD_PRELOAD ./antiantiptrace.so**
gdb>**r**
(no debugging symbols found)...(no debugging symbols found)...(no debugging symbols found)...
Are you trying to debug me ?

Le ***ptrace()***ne peut plus être hijack via notre librairie. Il va donc falloir l'intercepter directement dans le kernel.

3.3.1 Création d'un Loadable Kernel Module

Nous allons donc coder un LKM que nous injecterons dans Linux.
La table des syscalls contient les differents pointeurs pour rediriger les appels vers leur fonction. Le module kernel se chargera de patcher le pointeur de ***ptrace()***pour le faire pointer vers notre propre fonction. Celle-ci comparera le PID du processus appelant avec celui du processus débuggué et laissera passer le syscall en fonction.
Le code source ci-après est destiné a un kernel 2.6.x. En effet, Linux n'exporte plus sa table des syscalls depuis cette version. Nous récupérons donc son adresse hardcodée dans *System.map*

$ **grep sys_call_table /boot/System.map-‘uname -r‘**
c03b4740 D sys_call_table

Listing 5: Loadable Kernel Module

```
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/sched.h>
#include <asm/uaccess.h>

// extern void *sys_call_table[];  // for linux <= 2.4
/* hardcoded from System.map on my 2.6.12 */
#define SYS_CALL_TABLE 0xc03b4740

static unsigned int pid = 0;
int savedpid;
static asmlinkage long (*old_ptrace)(long,long,void*,void*);

module_param(pid, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(pid,"Process-ID to restrict");
```

```
asmlinkage
long my_sys_ptrace(long request, long pid, void *addr, void *data)
{
        /* Check if our proggy is using ptrace */
        if ( current->pid == savedpid ) {
                printk("call has been caught!\n");
                return 0;
                }
        else old_ptrace(request,pid,addr,data);
}

/* entry of lkm */
int init_module(void)
{
        printk(KERN_INFO "Anti-ptrace() LKM loaded\n");
        /* Save the original pointer */
        old_ptrace = *(long (*))(SYS_CALL_TABLE+__NR_ptrace*sizeof(void *));
        printk("ptrace at 0x%X - ",old_ptrace);
        /* Replace it with our function */
        *(long (*))(SYS_CALL_TABLE+__NR_ptrace*sizeof(void *))
                = &my_sys_ptrace;
        printk("Syscall hooked\n");
        printk("Spying PID = %u\n", pid);
        savedpid = pid;
}

void cleanup_module(void)
{
        if ( *(long (*))(SYS_CALL_TABLE+__NR_ptrace*sizeof(void *))
                != &my_sys_ptrace ) {
                printk(KERN_ALERT "Syscall has been overwriten");
                printk(KERN_ALERT " by somebody else!\n");
                printk(KERN_ALERT "System can now be unstable...\n");
                }
        *(long (*))(SYS_CALL_TABLE+__NR_ptrace*sizeof(void *))
                = old_ptrace;
        printk("Original syscall restored\n");
        printk(KERN_INFO "Anti-ptrace() LKM unloaded\n");
}
```

La fonction *init_module*, équivalente à *main* pour un programme classique, patche la table tandis que *cleanup_module* (équivalente à *atexit*) la restaure.
Le danger de cette opération provient du fait qu'un autre module pourrait tenter de hooker ce syscall en même ce qui entrainerait un conflit dans la restauration des pointeurs.
Ce module a testé sur un kernel 2.6.12 non-patché (sans PaX, rsbac, ou autre grsec...). Il faut penser à changer l'adresse de *sys_call_table* avant de vouloir le recompiler.

3.3.2 Compilation du module

Il est nécessaire de posséder les sources de son kernel pour pouvoir compiler le module. Celles-ci sont disponibles sur `http://www.kernel.org`
Voici le contenu du fichier Makefile nécessaire pour compiler le module.

Listing 6: Makefile

```
obj-m += noptrace.o

all:
	make -C /lib/modules/`uname -r`/build M=`pwd` modules

clean:
	make -C /lib/modules/`uname -r`/build M=`pwd` clean
```

$ **make**
Building modules, stage 2.
MODPOST
CC /home/judecca/noptrace.mod.o
LD [M] /home/judecca/noptrace.ko

Et voila notre LKM *noptrace.ko* prêt à l'emploi !

3.3.3 Utilisation du module

Le module a besoin du PID du processus débuggué pour fonctionner efficacement. Nous lancons donc une session gdb.

$ **gdb -q ./antiptrace**
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
gdb>**b main**
Breakpoint 1 at 0x80483a6
gdb>**r**
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x080483a6 in main ()

$ **ps aux | grep antiptrace**
judecca 6247 0.2 0.9 10460 4648 pts/2 S+ 20 :20 0 :00 gdb -q ./antiptrace
judecca 6248 0.0 0.0 1388 232 pts/2 T 20 :20 0 :00 /home/judecca/antiptrace
judecca 6252 0.0 0.1 3752 720 pts/1 R+ 20 :20 0 :00 grep antiptrace
$ **su**
Password :
/sbin/insmod noptrace.ko pid=6248
lsmod | grep noptrace # on vérifie s'il est bien chargé
noptrace 2444 0

gdb> **c**
Usage : /home/judecca/antiptrace <password>

Le processus n'a donc pas detecté le ***ptrace()***et gdb s'est éxecuté correctement.
Le module a donc bien marché ! On peut alors le décharger du kernel.

/sbin/rmmod noptrace
dmesg | tail -n6
Anti-ptrace() LKM loaded
ptrace at 0xC0107A96 - Syscall hooked
Spying PID = 6248
call has been caught !
Original syscall restored
Anti-ptrace() LKM unloaded

3.4 Méthode 4 : Ne pas utiliser *ptrace()*

La fonction ***ptrace()***n'est en fait pas indispensable pour débugguer un binaire ELF, de nombreux debuggers permettent un tracing sans avoir recours à cet appel système : c'est le cas des debuggers kernel comme LinICE ou PrivateICE, TheDude. La récente version de elfsh (`http://elfsh.segfault.net/`) propose un debugger, *e2dbg*, n'utilisant pas ***ptrace()***. En matière de reverse engineering, la très peu discrète fonction ***ptrace()***trouve vite ses limites, c'est pourquoi il est naturellement pensable que gdb soit mis de coté face aux plus récents debuggers.

4 Remerciements

Merci à *kryshaam* d'avoir relu mon article, à *Z* pour son coup de main, ainsi qu'à toute la FRET et aux personnes qui animent le forum.

References

[1] French Reverse Engineering Team, `http://reverseengineering.online.fr/forum/`

[2] Beginners Guide To Basic Linux Anti Anti Debugging Techniques, `http://home.pages.at/f001/linux_anti_anti_debugging4CBJ.txt`

[3] The Linux Kernel Module Programming Guide, `http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`

[4] Mammon Homepage, `http://www.eccentrix.com/members/mammon/`

[5] The ELFsh Project, `http://elfsh.segfault.net/`

[6] Linice Kernel Debugger, `http://www.linice.com/`