시스템 콜의 흐름을 통해 살펴본 ptrace/kmod exploit 분석

Team Null@Root

mutacker(mutacker@null2root.org)
oprix(oprix@null2root.org)
thanks frog(frog@hackerslab.com)

2003년 3월 19일 certcc.or.kr에 Linux Kernel Ptrace 취약점이 보고되었다. 이를 system call의 흐름을 살펴봄으로써, 해당 취약점에 대해 살펴보고자 한다.

분석:

먼저 fork, clone, ptrace, execve 시스템 콜을 모니터링 하는 간단한 커널 모듈을 제작하고, 해당 시스템콜로 넘어오는 정보들을 살펴보았다. 아래에서 보이는 것은 n(system call 번호), u(uid), g(gid), eu(euid), eg(egid), pid순으로 보여주고 있으며, FN은 시스템콜을 호출한 프로세스의 명령어를, Msg는 그에대한 좀 더 상세한 정보를 출력하도록 하였다. 시스템콜의 번호는 2=fork, 11=execve, 120=clone, 26=ptrace이다.

```
( 1) n:  2, u:500, g:500, eu:500, eg:500, pid:10184, FN:bash, Msg: New Forked!!
( 2) n: 11, u:500, g:500, eu:500, eg:500, pid:10297, FN:bash, Msg: ./exptrace
( 3) n:  2, u:500, g:500, eu:500, eg:500, pid:10297, FN:exptrace, Msg: New
Forked!!
( 4) n:120, u:500, g:500, eu:500, eg:500, pid:10297, FN:exptrace, Msg: New
cloned!! pid: 10299
( 5) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 10,
pid: 10299, addr: 0x0
( 6) n: 11, u:500, g:500, eu:  0, eg:  0, pid:10299, FN:exptrace, Msg:
/sbin/modprobe
( 7) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 18,
pid: 10299, addr: 0x0
( 8) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: c,
pid: 10299, addr: 0x0
( 9) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c3d
(10) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c41
(11) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c45
(12) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c49
(13) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c4d
(14) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
```

```
pid: 10299, addr: 0x40011c51
(15) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c55
(16) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c59
(17) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c5d
(18) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c61
(19) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c65
(20) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c69
(21) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c6d
(22) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c71
(23) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c75
(24) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c79
(25) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 4,
pid: 10299, addr: 0x40011c7d
(26) n: 26, u:500, g:500, eu:500, eg:500, pid:10298, FN:exptrace, Msg: req: 11,
pid: 10299, addr: 0x0
(27) n:  2, u:500, g:500, eu:500, eg:500, pid:10297, FN:exptrace, Msg: New
Forked!!
(28) n: 11, u:500, g:500, eu:500, eg:500, pid:10300, FN:exptrace, Msg: /bin/sh
(29) n: 11, u:500, g:500, eu:500, eg:500, pid:10300, FN:sh, Msg: ./exptrace
(30) n: 11, u:  0, g:  0, eu:  0, eg:  0, pid:10300, FN:exgrptrace, Msg: /bin/sh
```

(1) 명령행에서 exploit실행 : bash fork() 수행
(2) exploit이 execve됨
(3) exploit내부에서 child process생성
(4) 부모 프로세스내에 있는 socket() 함수 호출에 의해 clone 시스템콜이 수행 이때, 생성된 새로운 프로세스(pid=10299)는 uid, gid, euid, egid가 모두 일반 사용자상태로 유지되고 있음.
(5) child process에서 clone에 의해 새로 생성된 프로세스에 대해 attach시도 (성공)
(6) attach 상태에 있는 process(10299)가 /sbin/modprobe를 실행하기 위해 execve system call호출 이때, 커널 내부에서 euid와 egid를 모두 0으로 설정해버림
(7) attach 되어져 있는 해당 프로세스(현재 euid와 egid가 0인 상태)에 대해 ptrace(PTRACE_SYSCALL, victim, 0, 0) 수행
(8) ptrace(PTRACE_GETREGS, victim, NULL, &regs) 수행
(9~25) ptrace(PTRACE_POKETEXT, victim, dst++, *src++) 수행 이 부분에서

attach되어 있는 process의 현재 eip가 가리키는 text영역에 쉘코드가 주입됨
(26) ptrace(PTRACE_DETACH, victim, 0, 0) 수행 이후 9~25사이에 주입된 쉘코드가 수행됨 해당 쉘코드는 exploit 프로그램을 소유자가 root이며 mode값이 4755인 파일로 변경시켜줌
(27~30) system(progname); 수행 해당 프로그램이 4755로 설정되어 있기 때문에 곧바로 root shell이 획득되어짐

커널 버젼 2.4.12이상에서는 setuid가 설정되어 있는 프로그램의 경우 ptrace에 의해 trace되어지는 것을 금지하고 있다. 하지만, 본 exploit에서 사용되어진 프로세스의 경우에는 초기에는 setuid가 설정이 되어있지 않다가 커널 내부에서 euid와 egid값을 0으로 변경함으로 인해 setuid가 설정된 프로세스에 대한 ptrace금지 규칙을 우회할 수 있었다. 즉, ptrace로 attach를 시도하는 시점에서는 일반 프로세스 상태로 있다가 커널에 의해 내부적으로 변경되어짐으로 인해, 정상적인 ptrace를 수행할 수 있었던 것이다.

아래의 코드는 커널 내부에서 해당 프로세스의 euid와 egid값을 0으로 변경하는 부분으로 문제로 지적된 부분이다.

/usr/src/linux/kernel/kmod.c
(Linux kernel 2.4.20 기준)

int exec_usermodehelper(char *program_path, char *argv[], char *envp[])의 일부

```
132         /* Give kmod all effective privileges.. */
133         curtask->euid = curtask->fsuid = 0;
134         curtask->egid = curtask->fsgid = 0;
135         cap_set_full(curtask->cap_effective);
136
137         /* Allow execve args to be in kernel space. */
138         set_fs(KERNEL_DS);
139
140         /* Go, go, go... */
141         if (execve(program_path, argv, envp) < 0)
142                 return -errno;
143         return 0;
```

위의 코드가 수행되어지기 위해서는 모듈을 처리하기 위한 호출 발생이 필요한데, 이를 위해 exploit에서는 socket()을 이용하였다. socket(AF_SECURITY, SOCK_STREAM, 1); 부분은 /usr/src/linux/net/socket.c 내부에서 다음의 코드를 수행하기 위함으로 socket(AF_UNSPEC,SOCK_STREAM,1); 을 호출하더라도 동일한 결과를 얻을 수 있다.

-- /usr/src/linux/net/socket.c --

(Linux kernel 2.4.20 기준)

```
858         if (net_families[family]==NULL)
859         {
860                 char module_name[30];
861                 sprintf(module_name,"net-pf-%d",family);
862                 request_module(module_name);
863         }
```

이 현상은 AF_INET 의 경우는 이런 현상이 일어나지 않았다. 이것은 즉 잘 쓰이지 않는 코드(AF_SECURITY)를 호출하면 net-pf-14 모듈을 올리는 경우에 일어나는 현상이다. AF_IRDA 로 실험을 해 보았는데 마찬가지의 결과를 얻었다.

-- /usr/include/linux/socket.h --

```
#define AF_UNSPEC   0
#define AF_UNIX     1   /* Unix domain sockets      */
#define AF_LOCAL    1   /* POSIX name for AF_UNIX   */
#define AF_INET     2   /* Internet IP Protocol     */
#define AF_AX25     3   /* Amateur Radio AX.25      */
#define AF_IPX      4   /* Novell IPX               */
#define AF_APPLETALK    5   /* AppleTalk DDP        */
#define AF_NETROM   6   /* Amateur Radio NET/ROM    */
#define AF_BRIDGE   7   /* Multiprotocol bridge     */
#define AF_ATMPVC   8   /* ATM PVCs                 */
#define AF_X25      9   /* Reserved for X.25 project    */
#define AF_INET6    10  /* IP version 6             */
#define AF_ROSE     11  /* Amateur Radio X.25 PLP   */
#define AF_DECnet   12  /* Reserved for DECnet project  */
#define AF_NETBEUI  13  /* Reserved for 802.2LLC project*/
#define AF_SECURITY 14  /* Security callback pseudo AF */
#define AF_KEY      15  /* PF_KEY key management API */
#define AF_NETLINK  16
```

-- /usr/src/linux/kernel/kmod.c --

(Linux kernel 2.4.20 기준)

```
219         pid = kernel_thread(exec_modprobe, (void*) module_name, 0);
220         if (pid < 0) {
221                 printk(KERN_ERR "request_module[%s]: fork failed, errno %d\n",
module_name, -pid);
222                 atomic_dec(&kmod_concurrent);
223                 return pid;
224         }
```

```
148 /*
149        modprobe_path is set via /proc/sys.
150 */
151 char modprobe_path[256] = "/sbin/modprobe";
152
153 static int exec_modprobe(void * module_name)
154 {
155                     static  char  *  envp[]  =  {  "HOME=/",  "TERM=linux",
"PATH=/sbin:/usr/sbin:/bin:/usr/bin", NULL };
156         char *argv[] = { modprobe_path, "-s", "-k", "--", (char*)module_name,
NULL };
157         int ret;
158
159         ret = exec_usermodehelper(modprobe_path, argv, envp);
160         if (ret) {
161                 printk(KERN_ERR
162                        "kmod: failed to exec %s -s -k %s, errno = %d\n",
163                        modprobe_path, (char*) module_name, errno);
164         }
165         return ret;
166 }
```

Exploit 에 관해서

초기에 발표된 Wojciech Purczynski [cliph@isec.pl] 의 Exploit 의 경우 ShellCode 에서 실행되는 프로그램에 Root 권한 Suid 를 주기 위해서 /proc/[pid]/exe 파일을 읽는다. 이후에 발표된 anszom@v-lo.krakow.pl 의 Exploit 경우 /proc/[pid]/exe 와 실제 프로그램이 같은 것인지 확인하는데 사용된다. 따라서 chmod 700 /proc 가 중간에서 막을 수 있는 부분이 가능했다. 그렇지만, 실제로는 이 부분은 별로 중요하지 않은 부분이기 때문에, 이런 검사를 무시하고 작동시키는 snooq의 Exploit 인 myptrace.c 가 만들어졌고 실제로도 이 Exploit 은 잘 작동된다. 이 문제의 해결책은 Patch 를 하는 방법 밖에 없다.

Kernel Patch 에 관해서

linux-2.4.18-ptrace.patch

아래 형식의 kernel_thread 를 다시 작성하고 arch_kernel_thread 함수를 새로 작성하는 방법의 패치를 했다.

```
long kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
+{
+   struct task_struct *task = current;
+   unsigned old_task_dumpable;
+   long ret;
+
+   /* lock out any potential ptracer */
```

```
+   task_lock(task);
+   if (task->ptrace) {
+       task_unlock(task);
+       return -EPERM;
+   }
+
+   old_task_dumpable = task->task_dumpable;
+   task->task_dumpable = 0;
+   task_unlock(task);
+
+   ret = arch_kernel_thread(fn, arg, flags);
+
+   /* never reached in child process, only in parent */
+   current->task_dumpable = old_task_dumpable;
+
+   return ret;
+}
```

ptrace 가 된 상태에서 arch_kernel_thread 에 접근할 경우 에러를 내는 부분과 Process Memory 를 Dump 권한이 있는지 확인하는 검사 루틴이 추가 되었다.

결론:

위의 커널 코드는 socket()을 통해 취약점이 존재하는 exec_usermodehelper함수가 수행되어지는 과정을 보여주고 있다. 처음 프로세스가 kernel_thread()에 의해 생성이 되어지는 시점에서는 setuid가 설정되어있지 않은 상태의 프로세스가 생성되어졌다가 이후 exec_usermodehelper()함수 내부에서 그 설정값을 변경해 버림으로 인해 setuid가 설정된 프로세스에 대해 trace를 할 수 없다는 규칙을 우회한 방법을 보여준 예라고 할 수 있겠다.