آنالیز مسیر اجرایی جهت یافتن rootkit های مبتنی بر هسته

مقدمه

در خلال سالهایی چند، بشر تکنیک های بسیاری را برای پوشش گذاشتن بر وجود نفوذگر در یک سیستم هک شده توسعه داده است. به منظور انجام حضور نامحسوس خود روی یک سیستم، backdoor ها ساختارها و کد هسته را دستکاری می کنند و سبب می شوند که هیچ کس نتواند به آن هسته اعتماد کند. منظور از هیچ کس، همان ابزارهای IDS و ... می باشد.

در این مقاله، تکنیکی را مبنی بر شمارش دستورهای اجرا شده در چند فراخوانی سیستمی ارائه می دهم که می توان آنرا جهت تشخیص rootkit های هسته ای گوناگون بکار برد که شامل SucKIT یا PRRF و Palmo1 می شوند که جدول syscall (فراخوانی سیستمی) را تغییر می دهند. در این مقاله روی هسته نسخه ۲,۴ با یک پردازنده از خانواده اینتل ۳۲-بیت (IA32) تمرکز می کنیم. در انتهای مقاله نیز کدمنبع PatchFinder را نیز خواهید دید که نقش یک PoC را برای تکنیک تشریح شده بازی می کند.

ما در این مقاله قصد نداریم که چگونگی نوشتن یک rootkit هسته ای را تشریح کنیم. برای جزئیات می توانید به کتابها و مقاله های مربوطه مراجعه کنید. اما در این مقاله، به صورت مختصر تکنیک های شناخته شده را مشخص می کنیم و لذا می توان مقاومت آنها را در مقابل روش تشخیصی که این مقاله ارائه می شود سنجید.

پشت صحنه!

اکنون نگاهی سریع به rookit های معمول هسته ای می اندازیم. چنین برنامه هایی بایستی دو مشکل را حل کنند: نخست، یافتن راهی برای ورود به هسته و دوم، دستکاری هسته به یک روش هوشمندانه. در لینوکس اولین کار را می توان با استفاده از Loadable Kernel Module ها (ماژول های هسته ای قابل بارگذاری) یا وسیله /dev/kmem به انجام رسانید.

ورود به هسته

استفاده از LKM راحت ترین و زیباترین راه برای تغییر هسته در حال اجرا می باشد. این روش در ابتدا توسط HalfLife مورد بحث قرار گرفت. Backdoor های مشهور زیادی وجود دارند که از LKM استفاده می کنند. به هر حال این تکنیک یک نقطه ضعف نیز دارد و آن اینکه LKM روی بعضی سیستم ها غیرفعال می باشد.

هنگامی ما پشتیبانی لازم از LKM را نداریم، می توانیم از تکنیک توسعه یافته توسط Silvio Cesare استفاده کنیم که از /dev/kmem جهت دستیابی مستقیم به حافظه هسته استفاده می کند. اما این تکنیک زمینه کاری مناسبی برای روش ما نیست، چرا که patch کردن تابع do_write_mem() کافی نیست، همان طور که اخیرا توسط Guillaume Pelat به اثبات رسید.

تغییر و دستکاری جدول syscall

فرض کنیم که ما می توانیم حافظه هسته را بنویسیم (عمل write را انجام دهیم)، در این صورت از خود می پرسیم که چه چیزی را باید دستکاری کرد و تغییر داد.

بسیاری از rootkit ها جدول syscall را جهت، هدایت (redirect) فراخوانی های سیستمی مناسب مثل sys_read()، sys_write()، sys_getdents() و ... دستکاری می کنند. برای جزئیات بیشتر می توانید به کد منبع یکی از rootkit های مشهور مراجعه کنید (مثل Knar01 و ...). به هر حال این روش را بسادگی می توان ردیابی کرد. این کار با مقایسه جدول syscall فعلی با جدول اصلی که پس از ایجاد هسته ذخیره می گردد انجام می شود.

هنگامی که مکانیزم LKM روی سیستم فعال باشد، می توانیم از ماژول ساده ای استفاده کنیم که جدول syscall را خوانده (به صورت مستقیم به حافظه هسته دستیابی دارد) و سپس آنها در userland (برای مثال ناشی از /proc filesystem) قرار می دهد. متاسفانه هنگامی که LKM مورد پشتیبانی نباشد، ما نمی توانیم حافظه هسته را بخوانیم، لذا از sys_read() یا sys_mmap() یا /dev/kmem mmap برای خواندن استفاده می کنیم. ما نمی توانیم مطمئن باشیم که کد مخربی که سعی در یافتن آن داریم، فراخوانی های سیستمی sys_read() و sys_mmap() را تغییر نداده اند.

تغییر کد هسته

بجای تغییر اشاره گرهای جدول syscall، برنامه مخرب می توان کدی را در هسته تغییر دهد، مانند تابع system_call. در این مورد، آنالیز جدول syscall چیز خاصی را نشان نخواهد داد. بنابراین، ما می توانیم حافظه هسته را اسکن کرده و ببینیم که آیا ناحیه کد (Code Area) تغییر کرده است یا خیر.

اگر از LKM پشتیبانی بعمل آید اینکار بسیار راحت است و در غیر این صورت ما باید از طریق /dev/kmem به حافظه هسته دست یابیم و مجددا به مشکل sys_read() و sys_mmap() و غیرقابل اعتماد بودن آنها (تغییر یافتن آنها از قبل توسط برنامه مخرب) برمی خوریم.

SucKIT نمونه ای از یک rootkit می باشد که از /dev/kmem برای دستیابی به هسته و سپس تغییر کد system_call استفاده می کند و به جدول syscall اولیه کاری ندارد. اگرچه SucKIT، رفتار sys_read() یا sys_mmap() را تغییر نمی دهد، اما این قابلیت اضافه شده است و لذا تشخیص چنین backdoor های با استفاده از تکنیک های قراردادی (یعنی اسکن کردن حافظه از طریق /dev/kmem و ...) غیرممکن می شود.

تغییر دیگر اشاره گرها

ایده جالبی برای تغییر چندین اشاره گر در /proc filesystem ارائه شده است. اما اگر LKM در سیستم پشتیبانی بعمل آید، می توانیم حداقل از نظر تئوری تمام ساختارهای هسته را چک کرده و ببینیم که آیا شخصی چند اشاره گر را تغییر داده است یا خیر. به هر حال، در عمل این کار اندکی سخت خواهد بود، چرا که ما باید تمام مکان هایی که rootkit ممکن است اکسپلویت کند را از قبل پیش بینی کنیم. اگر LKM غیرفعال باشد، اما با همان مشکلی که در پاراگراف های فوق تشریح شد، مجددا مواجه می شویم.

آنالیز مسیر اجرایی (step کردن هسته)

همان طور که می بینیم، تشخیص rootkit های هسته ای یک کار جزئی نیست. البته اگر از LKM پشتیبانی بعمل آید، می توانیم از نظر تئوری کل حافظه هسته را اسکن کرده و مزاحم را تشخیص دهیم! ب هر حال، باید در تعیین چیزی که بدنبال آن می گردیم باید نهایت دقت را بکار بریم. تفاوت های موجود در کد نشان می دهد که چیزی در این بین اشتباه است! اگرچه تغییر چند داده بایستی بعنوان یک هشدار رفتار کند، اما تغییرات دیگر ساختارها ممکن است وظایف معمول و روزانه هسته را نیز بدنبال داشته باشد و لذا حس سو ظن بسیار کاهش می یابد.

اگر LKM در هسته ما (برای امنیت بیشتر) غیرفعال باشد، چیزهایی ذکر شده پیچیده تر هم خواهند شد. آنگاه، همان طور که گفتیم، نمی توانیم حافظه هسته را بخوانیم، چرا که مطمئن نیستیم که sys_read() بایت های واقعی را برگرداند (لذ نمی توانیم /dev/kmem را بخوانیم). ما همچنین مطمئن نیستیم که sys_mmap2() صفحات map شده را با بایت های صحیح پر کند...

اکنون در نگاه دیگری به این قضیه می نگریم. اگر شخصی چندین تابع موجود در هسته را تغییر دهد، به احتمال بسیار زیاد تعداد دستورات اجرایی در خلال چند فراخوانی سیستمی (برای مثال sys_getdents() که نفوذگر جهت مخفی کردن فایل ها استفاده می کند) با نسخه اصلی هسته متفاوت خواهد بود. در حقیقت، کد مخرب بایستی چند عملیات اضافی را (نظیر قطع کردن نام های فایل محرمانه) قبل از برگشت نتایج به userland انجام دهد. این موضوع دال بر اجرای دستورات بیشتری نسبت به سیستم های غیرآلوده خواهد بود. ما می توانیم این تفاوت را اندازه گیری کنیم!

Step کننده سخت افزار

می توان پردازنده IA32 را در حالت single-step بکار برد. این کار با تنظیم بیت TF (0x100) در ثبات EFLAGS انجام می شود. در این حالت، پردازنده یک استثنا اشکال زدایی[1] (#DB) را پس از اجرای تمام دستورات تولید خواهد کرد.

هنگامی که #DB تولید می شود چه اتفاقی رخ می دهد؟ پردازنده اجرای پروسه فعلی را رها کرده و handler مربوط به debug exception را فراخوانی می کند.

در پردازنده های اینتل، آرایه ای از ۲۵۶ دروازه (gate) وجود دارد که هر یک handler را برای یک بردار قاطع خاص توصیف می کند (احتمالا این مورد یکی از اطلاعات محرمانه اینتل می باشد چرا که آنها یان اعداد اسکالری را "vector" یا "بردار" می نامند).

برای مثال در منطقه 0x80، یک دروازه وجود دارد. این دروازه بازگو خواهد کرد که handler مربوط به trap 0x80 – فراخوانی سیستمی لینوکس- در کجا قرار دارد. همان طور که می دانیم این مورد توسط پردازنده با استفاده از دستور 'int 0x80' تولید شده است. این آرایه شامل ۲۵۶ دروازه، Interrupt Descriptor Table یا IDT نامیده می شود و توسط ثبات IDTR مشخص می گردد.

---

[1] Debug Exception

در هسته لینوکس، شما می توانید این handler را در فایل arch/i386/kernel/entry.s پیدا کنید که "debug" نامیده می شود. همان طور که می بینید، بعد از چند عمل، تابع do_debug() را فراخوانی می کند که در arch/i386/kernel/traps.c تعریف شده است.

چون #DB Exception فقط برای single stepping تخصیص نیافته و برای بسیاری از عملیات debugging دیگر نیز بکار می رود، تابع do_debug() اندکی پیچیده می باشد. هرچند برای ما مهم نیست. تنها چیزی که باید به آن توجه کنیم این است که بعد از تشخیص #DB Exception ای که توسط single stepping سبب شد (بیت TF)، یک سیگنال SIGTRAP به پروسه ردیابی شده ارسال می شود. پروسه ممکن است این سیگنال را بپذیرد. پس، به نظر می آید ما می توانیم عملیاتی مانند سناریو فوق را در برنامه userland خود انجام دهیم:

```
volatile int traps = 0;

int trap () {
        traps++;
}

main () {
        ...
        signal (SIGTRAP, sigtrap);

        xor_eflags (0x100);
        /* call syscall we want to test */
        read (fd, buff, sizeof (buff));
        xor_eflags (0x100);

        printf ("testing syscall takes %d instruction\n", traps);
}
```

این برنامه ساده به نظر می رسد. اما یک اشکال دارد و آن اینکه این کد همان طوری که ما می خواهیم کار نخواهد کرد. در متغیر traps ما تنها تعداد دستورهای اجرا شده در userland را خواهیم داشت. همان طور که می دانیم، read() تنها یک wrapper برای دستور 'int 0x80' می باشد که سبب می شود پردازنده exception handler مربوط به 0x00 را فراخوانی کند. متاسفانه پردازنده هنگام اجرای 'int x'، فلگ TF را پاک می کند (و این دستور سبب تغییر سطح دسترسی می شود).

به منظور step کردن هسته، ما باید کدی را به آن اضافه کنیم که مسئول تنظیم فلگ TF برای بعضی پروسه ها باشد. مکان مناسب برای الحاق و اضافه کردن چنین کدی، ابتدای روتین اسمبلر 'system_call' می باشد (که در arch/i386/kernel/entry.s تعریف شده است) که یک entry برای exception handler مربوط به 0x80 می باشد.

همان طور که در قبل ذکر شد، آدرس 'system_call' در دروازه موجود در منطقه 0x80 در IDT می باشد. هر gateway یا پل ارتباطی (IDT شامل ۲۵۶ تا از آنها می باشد) قالب زیر را دارد:

```
struct idt_gate {
        unsigned short  off1;
        unsigned short  sel;
        unsigned char   none, flags;
        unsigned short  off2;
} __attribute__ ((packed));
```

فیلـد 'sel' انتخـابگر قطعـه (segment selector) را نگهـداری مـی کنـد و در خـصوص لینـوکس معـادل KERNEL_CS_ می باشد. روتین handler در off2<<16+off1 و درون قطعه قرار گرفته است و چـون قطعـات در لینوکس دارای پایه 0x0 می باشند، لذا نظر می رود که این روتین معادل با آدرس خطی یا Linear Address مـی باشد.

فیلدهای 'none' و 'flags' به این منظور استفاده می شوند که مقداری اطلاعات اضافی را راجـع بـه فراخـوانی handler به پردازنده بازگو کنند. ثبات IDTR به ابتدای جدول IDT اشاره دارد (این جدول آدرس خطـی را تعیـین می کند، نه آدرس خطی که در idt_gate موجود بود):

```
struct idtr {
        unsigned short  limit;
        unsigned int    base;           /* linear address of IDT table */
} __attribute__ ((packed));
```

حال می بینیم که یافتن آدرس system_call در هسته لینوکس کار ساده ای است. عـلاوه بـراین، تغییـر ایـن آدرس به یک آدرس جدید نیز کار ساده ای خواهد بود. البته ما نمی توانیم این کار را از ناحیه userland انجام دهیم. به این دلیل که ما به یک ماژول هسته ای نیاز داریم که آدرس handler یا نگهدارنده 0x80 را تغییر دهد و کد جدید را اضافه کند که ما بعنوان system_call مورد استفاده قرار می دهیم و این کد جدید ممکن است چیزی شـبیه بـه زیـر باشد:

```
ENTRY(PF_system_call)
        pushl %ebx
        movl $-8192, %ebx
        andl %esp, %ebx                 # %ebx <-- current

        testb $PT_PATCHFINDER,24(%ebx)          # 24 is offset of 'ptrace'
        je continue_syscall
        pushf
        popl %ebx
        orl $TF_MASK, %ebx              # set TF flag
        pushl %ebx
        popf

continue_syscall:
        popl %ebx
        jmp *orig_system_call
```

همان طور که می بینید، ما تصمیم به استفاده از فیلد 'ptrace' درون توصیفگر پروسه[2] کردیم تا تعیـین کنـیم آیا یک پروسه مشخص می خواهد به صورت مجزا و واحد ردیابی شود یـا خیـر (اصـطلاحا ایـن عمـل را single trace شـدن یـک پروسـه مـی گـوییم). پـس از تنظـیم فلـگ TF، نگهدارنـده اصـلی system_call اجـرا مـی شـود و تابع sys_xxx() را فراخوانی کرده و سپس روند اجرا (execution) را به userland بر مـی گردانـد کـه ایـن کـار بـا استفاده از دستور 'iret' انجام می شود. تا هنگام 'iret' هر دستور واحد ردیابی خواهد شـد. البتـه مـا همچنـین مجبـور

[2] process descriptor

خواهیم بود تا نگهدارنده #DB خود را ارائه دهیم تا تمام این دستورات را بشماریم (این نگهدارنده جایگزین نگهدارنده سیستمی خواهد شد):

```
ENTRY(PF_debug)
        incl PF_traps
        iret
```

متغیر PF_traps در خلال بارگذاری ماژول، در جایی درون هسته قرار گرفته است. برای کامل کردن قضیه، ما باید یک فراخوانی سیستمی جدید را نیز اضافه کنیم که بتوان آنرا از userland فراخوانی کرد تا فلگ PT_PATCHFINDER را در متغیر 'ptrace' مربوط به توصیفگر پروسه فعلی را تنظیم کند و در نهایت منجر به reset کردن یا برگردادن مقدار شمارش شده شود.

```
asmlinkage int sys_patchfinder (int what) {
        struct task_struct *tsk = current;

        switch (what) {
                case PF_START:
                        tsk->ptrace |= PT_PATCHFINDER;
                        PF_traps = 0;
                        break;
                case PF_GET:
                        tsk->ptrace &= ~PT_PATCHFINDER;
                        break;
                case PF_QUERY:
                        return PF_ANSWER;
                default:
                        printk ("I don't know what to do!\n");
                        return -1;
        }
        return PF_traps;
}
```

در این روش ما هسته را تغییر داده ایم، پس می توانیم اندازه بگیریم که هر فراخوانی سیستمی چه تعداد دستور را اجرا خواهد کرد. module.c را انتهای مقاله جهت اطلاعات بیشتر بررسی کنید.

آزمایشات

فرض کنیم که هسته به ما اجازه می دهد تا دستورات را در هر فراخوانی سیستمی بشماریم، در این صورت با این مشکل مواجه می شویم که چه چیزی را باید بشماریم و چه توابع هسته ای را باید چک کرد؟

برای پاسخ به این سوال ما بایستی فکر کنیم که وظیفه اصلی هر rootkit چیست!؟ وظیفه اصلی یک rookit، مخفی کردن وجود پروسه، ارتباطات و فایل های نفوذگر در یک سیستم می باشد و این چیزها بایستی در مقابل ابزاری مانند ls، ps، netstat و ... نیز مخفی بمانند. این برنامه ها اطلاعات سیستمی را از طریق چند فراخوانی سیستمی شناخته شده جمع آوری می کنند.

حتی اگر backdoor مستقیما به syscall دسترسی نداشته باشد (مثل prrf.o)، چندین تابع هسته ای را که توسط یکی از فراخوانی های سیستمی فعال شده اند را تغییر خواهد داد. اما مشکل اینجاست که توابع دستکاری شده

(تغییر یافته) در هر فراخوانی سیستمی اجرا نخواهند شد. برای مثال، اگر ما تنها یک اشاره گر را در خواندن توابع در procfs تغییر دهیم، آنگاه کد نفوذگر تنها زمانی اجرا خواهد شد که تابع ()read فراخوانی شود و این تابع تنها زمانی فراخوانی می شود که نیاز به خواندن یک فایل خاص مانند proc/net/tcp/ باشد.

این امر عملیات تشخیص را اندکی پیچیده تر می کند، چرا که ما مجبور به اندازه گیری زمان اجرای هر فراخوانی سیستمی با آرگومان های مختلف هستیم. برای مثال، ما تابع ()sys_read را با خواندن "/etc/passwd"، "/dev/kmem" و "/proc/net/tcp" تست می کنیم (یعنی فایل منظم، وسیله و شبه فایل-پروسه ای را می خوانیم).

ما تمامی فراخوانی های سیستمی را (که حدود ۳۲۰ تا هستند) تست نمی کنیم،چ را که فرض بر آن است که backdoor بعضی عملیات روتین (منظور عملیات شناخته شده ای هستند که توابع API و فراخوانی های سیستمی آنها به راحتی قابل شناسایی است و حتی گاهی اوقات برنامه نویسان آنها را حفظ هستند!) را انجام می دهد، مثل مخفی کردن پروسه ها یا فایل ها و لذا در این صورت زیرمجموعه کوچکی از فراخوانی های سیستمی را مورد استفاده قرار داد.

آزمایشاتی که در PatchFinder وجود دارند در فایل tests.c تعریف شده اند. در زیر سعی بعمل می رود که آیا شخص چندین پروسه و/یا فایل را در procfs مخفی می کند یا خیر:

```
int test_readdir_proc () {
        int fd, T = 0;
        struct dirent de[1];

        fd = open ("/proc", 0, 0);
        assert (fd>0);

        patchfinder (PF_START);
        getdents (fd, de,  sizeof (de));
        T = patchfinder (PF_GET);

        close (fd);
        return T;
}
```

البته اضافه کردن یک آزمایش و تست جدید در صورت لزوم نیز کار ساده ای خواهد بود. به هر حال یک مشکل وجود دارد: False Positive ها. هسته لینوکس یک برنامه پیچیده است و بسیاری از فراخوانی های سیستمی شامل عبارت های if-then (شرطی) زیادی هستند. لذا بر اساس فاکتورهای بسیاری که در این بین وجود دارند، انواع و نتایج گوناگونی از اجرای یک تابع واحد بدست خواهد آمد. این فاکتورها شامل cache ها و 'وضعیت درونی سیستم' نیز می شود (که برای مثال می تواند تعداد ارتباطات باز TCP باز باشد و ...). تمام این موارد سبب می شوند که بعضی مواقع ببینید که دستورات بیشتر یا کمتری نسبت به حالت عادی اجرا می شوند. اصولا این تفاوت ها کمتر از ۱۰ می باشد (اختلاف تعداد دستورها در حالت طبیعی و حالت های خاص)، اما در بعضی اعمال و وظایف (مانند نوشتن در یک فایل) این اختلاف حتی به ۲۰۰ هم می رسد!

این مورد را می توان با افزایش تعداد تکرار هر آزمایش گرفته شده به حداقل رساند. اگر می بینید که خواندن "proc/net/tcp" زمان بیشتری را طلب می کند، سعی کنید ارتباطات TCP را reset کرده و سپس آزمایشات خود را مجددا تکرار کنید. اما اگر تفاوت ها جدی بود (مثلا بیشتر از ۶۰۰ دستور)، به احتمال بسیار زیاد، شخصی هسته شما را patch کرده است (منظور از patch کردن، تغییر یک چیز طبق میل خود می باشد). اما شما باید در هنگام

رویارویی با این موارد نیز بسیاری دقیق عمل کنید، چرا که این تفاوت ها ممکن است از ماژول های جدیدی که قبلا بارگذاری کرده بودید (و احتمالا این عمل غیرآگاهانه انجام شده است) ایجاد شده باشند.

برنامه PatchFinder

اکنون زمان نشان دادن کارکرد برنامه است. یک PoC (Proof-Of-Concept) در انتهای مقاله وجود دارد که مربوط به این برنامه می باشد. ما این برنامه را PatchFinder نامیدیم. این برنامه شامل دو قسمت است: یک ماژول که هسته را patch می کند بطوریکه اجازه debug کردن (اشکال زدایی) syscall ها را می دهد و یک برنامه userland که آزمایشات را انجام داده و نتایج را نشان می دهد. در ابتد شما باید فایلی را با نتایج آزمایش در سیستم مورد نظر ایجاد کنید (این فایل بعنی از این شما هسته جدید را نصب می کنید ایجاد می شود). سپس می توانید در هر زمانی سیستم خود را چک کنید، تنها به یاد داشته باشید که یک ماژول patchfinder.o را قبل از انجام آزمایش اضافه کنید. پس از آزمایش بایستی ماژول را حذف کنید. به خاطر داشته باشید که این ماژول جایگزین Native Debug Exception Handler در لینوکس می شود!

نتایج در یک سیستم سالم (منظور سیستمی که هسته آن patch نشده باشد)، چیزی شبیه به زیر خواهد بود (تفاوت های کوچکی در این بین وجود دارند که می توانید برای پی بردن به آنها به ستون diff نگاه کنید):

```
   test name      | current | clear | diff  | status
-------------------------------------------------------
open_file         |   1401|   1400|     1|  ok
stat_file         |   1200|   1200|     0|  ok
read_file         |   1825|   1824|     1|  ok
open_kmem         |   1440|   1440|     0|  ok
readdir_root      |   5784|   5774|    10|  ok
readdir_proc      |   2296|   2295|     1|  ok
read_proc_net_tcp |  11069|  11069|     0|  ok
lseek_kmem        |    191|    191|     0|  ok
read_kmem         |    322|    321|     1|  ok
```

آزمایشات در همان سیستم با برنامه در حال اجرای Adore انجام شد و نتایج زیر از آن دریافت شد:

```
   test name      | current | clear | diff  | status
-------------------------------------------------------
open_file         |   6975|   1400|  5575| ALERT!
stat_file         |   6900|   1200|  5700| ALERT!
read_file         |   1824|   1824|     0|  ok
open_kmem         |   6952|   1440|  5512| ALERT!
readdir_root      |   8811|   5774|  3037| ALERT!
readdir_proc      |  14243|   2295| 11948| ALERT!
read_proc_net_tcp |  11063|  11069|    -6|  ok
lseek_kmem        |    191|    191|     0|  ok
read_kmem         |    321|    321|     0|  ok
```

هنگامی که کد منبع Adore (یک backdoor) را آنالیز می کنید همه چیز سالم و بدون نقص (دست نخورده) خواهد بود. نتایج مشابه را می توان برای دیگر rootkit های معمول و مشهور (مثل Knark یا prrf.o و ...) نیز دریافت کرد (لطفا به خاطر داشته باشید که prrf.o جدول syscall را به طور مستقیم تغییر نمی دهد).

هنگامی که می خواهید هسته ای را که با SucKIT مورد حمله قرار گرفته است، آزمایش کنید، اتفاقات جالبی روی می دهند. به این صورت که چیزی شبیه به زیر دریافت خواهید کرد:

---== ALERT! ==--
It seems that module patchfinder.o is not loaded. However if you are sure that it is loaded, then this situation means that with your kernel is something wrong! Probably there is a rootkit installed!

دلیل این خطا این است که SucKIT جدول syscall اصلی را در مکان جدیدی کپی می کند و آنها به حالتی مثل knark یا adore تغییر می دهد و سپس آدرس جدول sycall را در کد system_call تغییر میدهد، بطوریکه به این کپی جدید از جدول syscall اشاره داشته باشد. چون این جدول sycall کپی شده حاوی هیچ فراخوانی سیستمی patchfinder نمی باشد (ماژول patchfinder تنها قبل از آزمایشات اضافه می شود)، لذا برنامه قادر به برقراری ارتباط و گفت و گو با ماژول نخواهد بود و گمان می کند که این ماژول بارگذاری نشده است. البته این وضعیت بسادگی فاش می کند که چیزی در هسته اشتباه است (یا شما بارگذاری ماژول را فراموش کرده اید)!

به خاطر داشته باشید که اگر patchfinder.o بارگذاری شده باشد، شما نمی توانید SucKIT را اجرا کنید. به این دلیل که روش نصب SucKIT طوری است که به چگونگی کدباینری system_call مربوط می شود. SucKIT به هنگام رویارویی با PS_system_call بجای handler اصلی 0x80 در لینوکس تعجب بسیاری خواهد کرد!!

نکته دیگری نیز وجود دارد که باید آنرا خاطر نشان ساخت. برنامه آزمایش کننده ما، قبل از شروع آزمایشات، سیاست زمان بندی[3] SCHED_FIFO را با بیشترین rt_priority تنظیم می کند. در حقیقت، در خلال آزمایشات، تنها پروسه patchfinder پردازنده (CPU) را در اختیار خواهد داشت (تنها انقطاع های سخت افزاری سرویس دهی می شوند) و تا زمان پایان آزمایشات، هیچ گاه حق تقدم پیدا نخواهد کرد. سه دلیل برای چنین خط مشی هایی وجود دارد.

بیت TF به ابتدای system_call تنظیم شده است و هنگام اجرای دستور 'iret' در انتهای exception handler پاک می شود. در خلال زمان تنظیم بیت TF، تابع sys_xxx() فراخوانی می شود، اما بعد از آن، چند چیز زمان بندی شده نیز اجرا می شوند و می توانند به پروسه سوئیچ شده هدایت شوند و این مسئله مناسب نیست، چرا که سبب می شود دستورات بیشتری اجرا شوند (در هسته، ما به دستوراتی که در پروسه سوئیچ شده اجرا می شوند، توجهی نداریم).

یک مسئله مهم تر نیز وجود دارد. من در خلال بررسی ها و تحقیقات مشاهده کردم که هنگامی که اجازه process switching را با تنظیم بیت TF می دهم، پس چند صد سوئیچ، این کار سبب restart شدن پردازنده می شود! من در فضای اینترنت هیچ توضیحی را راجع به چنین رفتاری پیدا نکردم. اما هنگامی که SET_SCHED تنظیم شود، مشکل مطرح شده در بالا رخ نمی دهد.

---

[3] scheduling policy

دلیل سوم برای استفاده از سیاست زمان-واقعی (real-time)، تنظیم وضعیت stable بودن سیستم تا جای ممکن است. برای مثال، اگر آزمایش ما با پروسه ای تداخل (parallel) داشت که فایل های بسیاری را باز کرده و می خواند (مثل grep)، آنگاه این مسئله بعضی آزمایشات مرتبط با sys_open() و sys_read() را تحت تاثیر قرار می دهد.

تنها زیان استفاده از چنین خط مشی هایی این است که سیستم شما در خلال آزمایشات غیرقابل دسترسی می باشد. اما، این آزمایشات در یک نشست آزمایشی نمونه زمان زیادی را نمی گیرد (بستگی به تعداد تکرارها در هر آزمایش دارد) و تنها ۱۵ ثانیه برای تکمیل آزمایش نیاز است.

اکنون به یک مسئله تکنیکی توجه کنید: کد منبع موجود در انتهای مقاله از LKM برای نصب تعمیم های هسته ای[4] توصیف شده استفاده می کند. در ابتدای مقاله گفتیم که در بعضی سیستم LKM در هسته کامپایل نشده است. لذا در این موارد ما تنها می توانیم از /dev/kmem استفاده کنیم. همچنین گفتیم که نمی توانیم به /dev/kmem متکی باشیم، چرا که از syscall ها برای دستیابی به آن استفاده می کنیم. اما این موضوع برای ابزارهایی مانند patchfinder مشکلی نخواهد بود، چرا که اگر rootkit در خلال بارگذاری تعمیم های ما مشکلاتی را ایجاد کند، در این صورت احتمالا برنامه کار نخواهد کرد (و این خود نیز نکته ای گویا در جهت تشخیص خواهد بود).

فریب دادن و مستحکم کردن برنامه PatchFinder

اکنون سعی در بحث راجع به روش های ممکن جهت کشف روش ارائه شده در حالت کلی و برنامه patchfinder در حالتی خاص، در این مقاله خواهیم کرد. همچنین سعی خواهیم کرد که چگونگی دفاع در برابر چنین حمله هایی را نیز به شما نشان دهیم (که احتمالا توزیع بعدی patchfinder این مباحث را به طور کلی پوشش خواهد داد)...

اولین چیزی که یک کد مخرب بررسی می کند این است که آیا ردیابی می شود یا خیر. این کد شاید به سادگی قطعه کد زیر را اجرا کند:

```
        pushf
        popl %ebx
        testb $0x100, %ebx
        jne i_am_traced
        # contine executing
        ...

i_am_traced:
        # deinstall for
        # a moment
        ...
```

[4] kernel extensions

هنگامی که کد مخرب پی می برد که تحت ردیابی است، معمولا خودش را از syscall مشخص پاک می کند. اما قبل از انجام این کار یک زمان سنج (تایمر) interrupt handler تنظیم می کند و پس از مثلا ۱ دقیقه، مجددا به آن syscall بر می گردد. چطور در برابر این حقه دفاع کنیم؟

به خاطر آورید که ما (یعنی patchfinder) کد را در همه زمان ردیابی می کنیم. لذا debug handler (که توسط ما ارائه شده است) می تواند دستور 'pushf' را بمحض اجرا شناسایی کند. سپس مکن است 'eflags' ذخیره شده روی پشته را تغییر دهد، بطوریکه برای پروسه ردیابی شده به نظر رسد که فلگ های TF پاک شده اند.

در زمان حاضر هیچ گونه حمله کلی در برابر آنالیز ردیابی اجرا (execution trace analysis) را نمی شناسم. به هر حال، احتمال انجام تقلب روی یک پیاده سازی (مانند patchfinder) می رود.

نفوذگر می تواند sys_write() را تغییر داده و بررسی کند که اگر پروسه دارای نام 'patchfinder' می باشد، یک خط به خصوص را در stdout بنویسد- یک خط با یک نتیجه از آزمایش (دروغین). آنگاه بافر write() را تغییر داده و کاربر نتایج دروغین را مشاهده خواهد کرد. مشابها، rootkit می تواند تشخیص دهد که فایلی با طرح بندی سیستم سالم خوانده شود و بافر برگشتی از sys_read() را بخواند و در این صورت patchfinder ارجاعات غلط را دریافت می دارد.

بعلاوه، هنگامی که patchfinder از یک عدد ثابت برای فراخوانی سیستمی خود استفاده می کند، rootkit مخرب می تواند نتایج برگشتی توسط آن فراخوانی سیستمی را تغییر دهد و در نتیجه فرآیند آزمایش را در رابطه با تعداد دستورات اجرا شده با مشکل مواجه سازد.

ما می توانیم در برابر چنین حملاتی دفاع کنیم. به این ترتیب که روی هر سیستم فایل اجرایی patchfinder با نام متفاوتی ظاهر شود و همچنین فایل را با طرح بندی های سیستم سالم رمزنگاری کنیم. بعلاوه، ما بایستی از شماره ها و اعداد متفاوتی برای فراخوانی سیستمی patchfinder استفاده کنیم. Rootkit می تواند براحتی پروسه patchfinder را ردیابی کند، به این ترتیب که مشاهده می کند که برنامه patchfinder، فراخوانی سیستمی خاصی را فراخوانی می کند.

ما تنها برای خالی کردن فراخوانی های سیستمی محدود نشده ایم. اگر چند عدد ناشناخته در آرگومان ها وجود داشته باشند، ما می توانیم هر فراخوانی سیستمی موجودی و اولین بررسی (آزمایش) را جایگزین کنیم. در مواردی که هیچ آرگومان ناشناخته ای وجود نداشت می توانیم sys_xxx() حقیقی را فراخوانی کنیم.

اگرچه استفاده از عدد تصادفی برای فراخوانی سیستمی patchfinder، کار را اندکی برای rootkit جهت تغییر نتایج برگشتی به پروسه patchfinder دشوار می سازد، اما هنوز برای کدمخرب این امکان وجود دارد تا بفهمد چه syscall ای مورد استفاده قرار می گیرد. این کار با جستجوی الگویی برای دستورهای باینری خاص (specific binary instruction) انجام می گیرد. این عمل براحتی انجام می پذیرد، چرا که نفوذگر همه چیز را راجع به کدمنبع (و باینری) مربوط به برنامه patchfinder می داند.

روش دیگر اینکه patchfinder، پروسه ای را علامت گذاری میکند به به روشی خاص ردیابی شود (یعنی تنظیم یک بیت در فیلد 'ptrace' برای توصیفگر پروسه). Rootkit مخرب می تواند روتین system_call را با نسخه خود تعویض کند. این نسخه جدید بررسی می کند که آیا پروسه توسط patchfinder علامت گذاری شده یا خیر و سپس از جدول syscall اصلی استفاده می کند. اگر پروسه توسط پروسه آزمایشی علامت گذاری نشده باشد، جدول syscall دیگری مورد استفاده قرار خواهد گرفت (که در آن چند تابع sys_xxx() جایگزین شده اند). چک

کردن اینکه rootkit سعی در جستجو چه جاهایی دارد (مثلا فیلد'ptrace')، برای #DB Exception Handler مشکل خواهد بود، چرا که کدی که این عمل را انجام می دهد حالت های مختلفی را می تواند دارا باشد.

کد debug exception handler نیز می تواند جایی را که متغیر شمارشگر (PF_traps) در حافظه قرار گرفته است، فاش کند. با دانستن این آدرس، rootkit می تواند این متغیر را از انتهای کد'عملیاتی'آن کم کند.

تنها راه حلی که من برای نقاط ضعف بالا درک می کنم، می تواند پلیمورفیسم مستحکم باشد (توجه داشته باشید که پلیمورفیسم یکی از ارکان اصلی در برنامه های شی گرا می باشد)! نظریه ای مورد نظر این است که یک تولید کننده کد پلیمورفیسم را به توزیع patchfinder اضافه کنیم که برای هر سیستمی که روی آن نصب می شود، image های باینری متفاوتی برای کد هسته ای patchfinder را ایجاد کند. این تولید بایستی بر اساس یک عبارت (که به آن pass-phrase می گوییم) که مدیر سیستم آنرا در زمان نصب ارائه می کند انجام شود. هنوز خط مشی پلیمورفیسم را پیاده سازی نکرده ایم، اما در انجام اینکار مصمم هستیم...

دیگر راه حل ها

تکنیک ارائه شده، قضیه ای از یک خط مشی کلی جهت تشخیص rootkit های مبتنی بر هسته می باشد. مشکل اصلی در چنین عملیاتی این است که ما می خواهیم از هسته جهت کمک در عملیات تشخیص کد مخرب استفاده کنیم که این کد مخرب کنترل کاملی روی هسته ما دارد. در حقیقت ما به هسته نمی توانیم اعتماد کنیم، اما از طرف دیگر می خواهیم مقداری اطلاعات مفید و قابل اعتماد نیز از آن دریافت کنیم.

Debug کردن مسیر اجرایی از فراخوانی های سیستمی، احتمالا تنها راه حل برای این مشکل نیست. ما قبلا نیز patchfinder را پیاده سازی کرده بودیم، اما روی تکنیک دیگری کار می کردیم که سعی در اکسپلویت کردن تفاوت ها در زمان اجرای چند فراخوانی سیستمی می کرد. آزمایشات انجام شده دقیقا همان هایی هستند که با patchfinder ضمیمه این مقاله شده اند. اما ما از دستور پروسه ای 'rdtsc' جهت محاسبه گردش های (cycles) در اجرای یک قطعه کد معین استفاده کردیم. این عملیات روی پردازنده های بالاتر از ۵۰۰ مگاهرتز بخوبی کار می کند.

متاسفانه، هنگامی که برنامه را روی یک پردازنده ۱,۷ گیگاهرتزی امتحان کردیم، فهمیدیم که زمان اجرای یک کد مشابه می تواند از یک آزمایش نسبت به آزمایش دیگر متفاوت باشد. این تفاوت بسیار زیاد بود و سبب بروز False Positive های بسیاری می گشت. اما مهم اینجاست که این تفاوت ها بواسطه محیط چندوظیفه ای (multitasking environment) سبب نمی شدند، بلکه عمیقا به معماری میکروسکوپیکی پردازنده های مدرن مربوط می شود. هم اکنون به عنوان گام بعدی پروژه سعی در طراحی راه حلی برای این مشکل نیز هستیم...

اگرچه تکنیک های بسیاری در یافتن rootkit های در هسته وجود دارد، اما به نظر می آید که خط مشی کلی بایستی اکسپلویت پلیمورفیسم باشد، همان طور که این خط مشی احتمالا تنها راه جهت دریافت اطلاعات قابل اعتماد و اطمینان از هسته در معرض خطر می باشد.

ارجاعات

- **Halflife, "Abuse of the Linux Kernel for Fun and Profit",**
- **Cyberwinds, "Knark-2.4.3" (Knark 0.59 ported to Linux 2.4), 2001.**
- **Stealth, "Adore v0.42": http://spider.scorpions.net/~stealth, 2001.**
- **Silvio Cesare, "Runtime kernel kmem patching": http://www.big.net.au/~silvio, 1998.**
- **sd, devik, "Linux on-the-fly kernel patching without LKM":(SucKIT source code), Phrack 58, 2001.**
- **palmers, "Sub proc_root Quando Sumus (Advances in Kernel Hacking)": (prrf source code), Phrack 58, 2001.**
- **Guillaume Pelat, "Grsecurity problem – modifying: 'read-only kernel'": http://securityfocus.com/archive/1/273002, 2002.**
- **"IA-32 Intel Architecture Software Developer's Manual", vol. 1-3: www.intel.com, 2001.**

ضمیمه: کد منبع برنامه PatchFinder

در این قسمت کد منبع مربوط به برنامه PatchFinder را می بینید که یک PoC برای تکنیک توصیف شده در مقاله نیز می باشد. این برنامه پلیمورفیسم را پیاده سازی نمی کند. برای اجرای این برنامه به پشتیبانی LKM نیاز است. اگر در خلال آزمایش متوجه عملیات ناشناسی شدید، احتمالا به این معنی خواهد بود که شخصی سیستم شما را تحت کنترل دارد (یا به اصطلاح آنرا root کرده است). از جنبه دیگر احتمالا وجود یک باگ نیز می رود. به هر حال با بررسی های بیشتر می تواند منشا پیدایش مشکل را بیابید.

حتما به خاطر داشته باشید که پس از آزمایشات، ماژول مربوط به patchfinder را حذف کنید.

```
<++> ./patchfinder/Makefile
MODULE_NAME=patchfinder.o
PROG_NAME=patchfinder

all: $(MODULE_NAME) $(PROG_NAME)

$(MODULE_NAME) : module.o traps.o
	ld -r -o $(MODULE_NAME) module.o traps.o

module.o : module.c module.h
	gcc -c module.c -I /usr/src/linux/include

traps.o : traps.S module.h
	gcc -D__ASSEMBLY__ -c traps.S


$(PROG_NAME): main.o tests.o libpf.o
	gcc -o $(PROG_NAME) main.o tests.o libpf.o

main.o: main.c main.h
	gcc -c main.c -D MODULE_NAME='"$(MODULE_NAME)"'\
		-D PROG_NAME='"$(PROG_NAME)"'
tests.o: tests.c main.h
libpf.o: libpf.c libpf.h
```

```
clean:
	rm -fr *.o $(PROG_NAME)
<--> ./patchfinder/Makefile
<++> ./patchfinder/traps.S
/*                                          */
/*          The Kernel PatchFinder version 0.9          */
/*                                          */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                          */

#include <linux/linkage.h>
#define __KERNEL__
#include "module.h"

tsk_ptrace  = 24                        # offset into the task_struct

ENTRY(PF_system_call)
	pushl %ebx
	movl $-8192, %ebx
	andl %esp, %ebx                         # %ebx <-- current

	testb $PT_PATCHFINDER,tsk_ptrace(%ebx)
	je continue_syscall
	pushf
	popl %ebx
	orl $TF_MASK, %ebx              # set TF flag
	pushl %ebx
	popf

continue_syscall:
	popl %ebx
	jmp *orig_system_call

ENTRY(PF_debug)
	incl PF_traps
	iret


<--> ./patchfinder/traps.S
<++> ./patchfinder/module.h
/*                                          */
/*          The Kernel PatchFinder version 0.9          */
/*                                          */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                          */

#ifndef __MODULE_H
#define __MODULE_H

#define PT_PATCHFINDER          0x80        /* should not conflict with PT_xxx
                                               defined in linux/sched.h */

#define TF_MASK                 0x100       /* TF mask in EFLAGS */

#define SYSCALL_VECTOR          0x80
#define DEBUG_VECTOR            0x1

#define PF_START     0xfee
#define PF_GET                  0xfed
#define PF_QUERY    0xdefaced
#define PF_ANSWER  0xaccede

#define __NR_patchfinder 250


#endif

<--> ./patchfinder/module.h
<++> ./patchfinder/module.c
```

```
/*                                                  */
/*          The Kernel PatchFinder version 0.9          */
/*                                                  */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                                  */

#define MODULE
#define __KERNEL__
#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include "module.h"

#define DEBUG 1

MODULE_AUTHOR("Jan Rutkowski");
MODULE_DESCRIPTION("The PatchFinder module");

asmlinkage int PF_system_call(void);
asmlinkage int PF_debug (void);
int (*orig_system_call)();
int (*orig_debug)();
int (*orig_syscall)(unsigned int);
extern void *sys_call_table[];
int PF_traps;

/* this one comes from arch/i386/kernel/traps.c */
#define _set_gate(gate_addr,type,dpl,addr) \
do { \
  int __d0, __d1; \
  __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
            "movw %4,%%dx\n\t" \
            "movl %%eax,%0\n\t" \
            "movl %%edx,%1" \
            :"=m" (*((long *) (gate_addr))), \
             "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
            :"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
             "3" ((char *) (addr)),"2" (__KERNEL_CS << 16)); \
} while (0)

struct idt_gate {
        unsigned short  off1;
        unsigned short  sel;
        unsigned char   none, flags;
        unsigned short  off2;
} __attribute__ ((packed));

struct idtr {
        unsigned short  limit;
        unsigned int    base;
} __attribute__ ((packed));

struct idt_gate * get_idt () {
            struct idtr idtr;
        asm("sidt %0" : "=m" (idtr));
            return (struct idt_gate*) idtr.base;
}

void * get_int_handler (int n) {
            struct idt_gate * idt_gate = (get_idt() + n);
            return (void*)((idt_gate->off2 << 16) + idt_gate->off1);
}

static void set_system_gate(unsigned int n, void *addr) {
            printk ("setting int for int %d -> %#x\n", n, addr);
```

```
	_set_gate(get_idt()+n,15,3,addr);
}

asmlinkage int sys_patchfinder (int what) {
	struct task_struct *tsk = current;

	switch (what) {
			case PF_START:
				tsk->ptrace |= PT_PATCHFINDER;
				PF_traps = 0;
				break;
			case PF_GET:
				tsk->ptrace &= ~PT_PATCHFINDER;
				break;
			case PF_QUERY:
				return PF_ANSWER;
			default:
				printk ("I don't know what to do!\n");
				return -1;
	}
	return PF_traps;
}

int init_module () {

	EXPORT_NO_SYMBOLS;

	orig_system_call = get_int_handler (SYSCALL_VECTOR);
	set_system_gate (SYSCALL_VECTOR, &PF_system_call);

	orig_debug = get_int_handler (DEBUG_VECTOR);
	set_system_gate (DEBUG_VECTOR, &PF_debug);

	orig_syscall = sys_call_table[__NR_patchfinder];
	sys_call_table [__NR_patchfinder] = sys_patchfinder;

	printk ("Kernel PatchFinder has been succesfully"
			"inserted into your kernel!\n");
#ifdef DEBUG
	printk (" orig_system_call : %#x\n", orig_system_call);
	printk (" PF_system_calli  : %#x\n", PF_system_call);
	printk (" orig_debug       : %#x\n", orig_debug);
	printk (" PF_debug         : %#x\n", PF_debug);
	printk (" using syscall    : %d\n", __NR_patchfinder);

#endif
	return 0;
}

int cleanup_module () {
	set_system_gate (SYSCALL_VECTOR, orig_system_call);
	set_system_gate (DEBUG_VECTOR, orig_debug);
	sys_call_table [__NR_patchfinder] = orig_syscall;

	printk ("PF module safely removed.\n");
	return 0;
}



<--> ./patchfinder/module.c
<++> ./patchfinder/main.h
/*                                                  */
/*          The Kernel PatchFinder version 0.9          */
/*                                                  */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                                  */
```

```
#ifndef __MAIN_H
#define __MAIN_H

#define PF_MAGIC     "patchfinder"
#define M_GENTTBL    1
#define M_CHECK          2
#define MAX_TESTS    9
#define TESTNAMESZ   32

#define WARN_THRESHOLD        20
#define ALERT_THRESHHOLD 500
#define TRIES_DEFAULT             200


typedef struct {
        int t;
        double ft;
        char name[TESTNAMESZ];
        int (*test_func)();
} TTEST;

typedef struct {
        char  magic[sizeof(PF_MAGIC)];
        TTEST test [MAX_TESTS];
        int   ntests;
        int   tries;
} TTBL;

#endif

<--> ./patchfinder/main.h
<++> ./patchfinder/main.c
/*                                                  */
/*          The Kernel PatchFinder version 0.9              */
/*                                                  */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                                  */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include "main.h"
#include "libpf.h"

void die (char *str) {
        if (errno) perror (str);
        else printf ("%s\n", str);
        exit (1);
}

void usage () {
 printf ("(c) Jan K. Rutkowski, 2002\n");
 printf ("email: jkrutkowski@elka.pw.edu.pl\n");
 printf ("%s [OPTIONS] <filename>\n", PROG_NAME);

 printf ("  -g save current system's characteristics to file\n");
 printf ("  -c check system against saved results\n");
 printf ("  -t change number of iterations per each test\n");
 exit (0);

}

void write_ttbl (TTBL* ttbl, char *filename) {
        int fd;
        fd = open (filename, O_WRONLY | O_CREAT);
        if (fd < 0) die ("can not create file");
        strcpy (ttbl->magic, PF_MAGIC);
```

```
	if (write (fd, ttbl, sizeof (TTBL)) < 0)
				die ("can not write to file");
	close (fd);
}

void read_ttbl (TTBL* ttbl, char *filename) {
	int fd;
	fd = open (filename, O_RDONLY);
	if (fd < 0) die ("can not open file");
	if (read (fd, ttbl, sizeof (TTBL)) != sizeof(TTBL))
				die ("can not read file");
	if (strncmp(ttbl->magic, PF_MAGIC, sizeof (PF_MAGIC)))
			die ("bad file format\n");
	close (fd);
}

main (int argc, char **argv) {
	TTBL current, clear;
	int tries = 0, mode = 0;
	int  opt, max_prio, i, j, T1, T2, dt;
	char *ttbl_file;
	struct sched_param sched_p;

	while ((opt = getopt (argc, argv, "hg:c:t:")) != -1)
			switch (opt) {
				case 'g':
					mode = M_GENTTBL;
					ttbl_file = optarg;
					break;
				case 'c':
					ttbl_file = optarg;
					mode = M_CHECK;
					break;
				case 't':
					tries = atoi (optarg);
					break;
				case 'h':
				default :
					usage();
			}

	if (getuid() != 0)
			die ("For some reasons you have to be root");

	if (!mode) usage();

	if (patchfinder (PF_QUERY) != PF_ANSWER) {
			printf (
				"\n                                        ---== ALERT! ==--\n"
				"It seems that module %s is not loaded. "
				"However if you are\nsure that it is loaded,"
			"then this situation means that with your\n"
				"kernel is something wrong! Probably there is "
				"a rootkit installed!\n", MODULE_NAME);
			exit (1);
	}

	current.tries = (tries) ? tries : TRIES_DEFAULT;
	if (mode == M_CHECK) {
			read_ttbl (&clear, ttbl_file);
			current.tries = (tries) ? tries : clear.tries;

	}

	max_prio = sched_get_priority_max (SCHED_FIFO);
	sched_p.sched_priority = max_prio;
	if (sched_setscheduler (0, SCHED_RR, &sched_p) < 0)
			die ("Setting realtime policy\n");
```

```
	fprintf (stderr, "* FIFO scheduling policy has been set.\n");

	generate_ttbl (&current);

	sched_p.sched_priority = 0;
	if (sched_setscheduler (0, SCHED_OTHER, &sched_p) < 0)
			die ("Dropping realtime policy\n");
	fprintf (stderr, "* dropping realtime schedulng policy.\n\n");

	if (mode == M_GENTTBL) {
			write_ttbl (&current, ttbl_file);
			exit (0);
	}

	printf (
	"    test name      | current | clear | diff  | status \n");
	printf (
	"-----------------------------------------------------\n");

	for (i = 0; i < current.ntests; i++) {
			if (strncmp (current.test[i].name,
                                        clear.test[i].name, TESTNAMESZ))
                                die ("ttbl entry name mismatch");

                        T1 = current.test[i].t;
                        T2 = clear.test[i].t;
                        dt = T1 - T2;
                        printf ("%-18s | %7d| %7d|%7d|",
                                current.test[i].name, T1, T2, dt);

                        dt = abs (dt);
                        if (dt < WARN_THRESHOLD) printf ("  ok ");
                        if (dt >= WARN_THRESHOLD && dt < ALERT_THRESHHOLD)
                                printf ("  (?) ");
                        if (dt >= ALERT_THRESHHOLD) printf (" ALERT!");

                        printf ("\n");
	}

}



<--> ./patchfinder/main.c
<++> ./patchfinder/tests.c
/*                                              */
/*          The Kernel PatchFinder version 0.9          */
/*                                              */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                              */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <assert.h>
#include "libpf.h"
#include "main.h"

int test_open_file () {
	int tmpfd, T = 0;

	patchfinder (PF_START);
	tmpfd = open ("/etc/passwd", 0, 0);
	T = patchfinder (PF_GET);
```

```
        close (tmpfd);
        return T;
}

int test_stat_file () {
        int T = 0;
        char buf[0x100];            /* we dont include sys/stat.h */

        patchfinder (PF_START);
        stat ("/etc/passwd", &buf);
        T = patchfinder (PF_GET);

        return T;
}

int test_read_file () {
        int fd, T = 0;
        char buf[0x100];

        fd = open ("/etc/passwd", 0, 0);
        if (fd < 0) die ("open");

        patchfinder (PF_START);
        read (fd, buf , sizeof(buf));
        T =  patchfinder (PF_GET);

        close (fd);
        return T;
}

int test_open_kmem () {
        int tmpfd;
        int T = 0;

        patchfinder (PF_START);
        tmpfd = open ("/dev/kmem", 0, 0);
        T = patchfinder (PF_GET);

        close (tmpfd);
        return T;
}

_syscall3(int, getdents, int, fd, struct dirent*, dirp, int, count)
int test_readdir_root () {
        int fd, T = 0;
        struct dirent de[1];

        fd = open ("/", 0, 0);
        if (fd < 0) die ("open");

        patchfinder (PF_START);
        getdents (fd, de,  sizeof (de));
        T = patchfinder (PF_GET);

        close (fd);
        return T;
}

int test_readdir_proc () {
        int fd, T = 0;
        struct dirent de[1];

        fd = open ("/proc", 0, 0);
        if (fd < 0) die ("open");

        patchfinder (PF_START);
        getdents (fd, de,  sizeof (de));
        T = patchfinder (PF_GET);
```

```
        close (fd);
        return T;
}

int test_read_proc_net_tcp () {
        int fd, T = 0;
        char buf[32];

        fd = open ("/proc/net/tcp", 0, 0);
        if (fd < 0) die ("open");

        patchfinder (PF_START);
        read (fd, buf , sizeof(buf));
        T = patchfinder (PF_GET);

        close (fd);
        return T;
}

int test_lseek_kmem () {
        int fd, T = 0;

        fd = open ("/dev/kmem", 0, 0);
        if (fd <0) die ("open");

        patchfinder (PF_START);
        lseek (fd, 0xc0100000, 0);
        T = patchfinder (PF_GET);

        close (fd);
        return T;
}

int test_read_kmem () {
        int fd, T = 0;
        char buf[256];

        fd = open ("/dev/kmem", 0, 0);
        if (fd < 0) die ("open");
        lseek (fd, 0xc0100000, 0);

        patchfinder (PF_START);
        read (fd, buf , sizeof(buf));
        T = patchfinder (PF_GET);

        close (fd);
        return T;
}

int generate_ttbl (TTBL *ttbl) {
        int i = 0, t;

#define set_test(testname) {                                    \
        ttbl->test[i].test_func = test_##testname;              \
    strcpy (ttbl->test[i].name, #testname);                     \
        ttbl->test[i].t = 0;                                    \
        ttbl->test[i].ft = 0;                                   \
        i++;                                                            \
}

        set_test(open_file)
        set_test(stat_file)
        set_test(read_file)
        set_test(open_kmem)
        set_test(readdir_root)
        set_test(readdir_proc)
        set_test(read_proc_net_tcp)
        set_test(lseek_kmem)
        set_test(read_kmem)
```

```
        assert (i <= MAX_TESTS);
        ttbl->ntests = i;
#undef set_test

        fprintf (stderr, "* each test will take %d iteration\n",
                                ttbl->tries);
        usleep (100000);
        for (i = 0; i < ttbl->ntests; i++) {
                        for (t = 0; t < ttbl->tries; t++)
                                ttbl->test [i].ft +=
                                        (double)ttbl->test[i].test_func();

                        fprintf (stderr, "* testing... %d%%\r",
                                        i*100/ttbl->ntests);
                        usleep (10000);
        }

        for (i = 0; i < ttbl->ntests; i++)
                        ttbl->test [i].t =
                        (int) (ttbl->test[i].ft/(double)ttbl->tries);

        fprintf (stderr, "\r* testing... done.\n");

        return i;

}


<--> ./patchfinder/tests.c
<++> ./patchfinder/libpf.h
/*                                      */
/*      The Kernel PatchFinder version 0.9      */
/*                                      */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                      */

#ifndef __LIBPF_H
#define __LIBPF_H

#include "module.h"

int patchfinder(int what);

#endif

<--> ./patchfinder/libpf.h
<++> ./patchfinder/libpf.c
/*                                      */
/*      The Kernel PatchFinder version 0.9      */
/*                                      */
/* (c) 2002 by Jan K. Rutkowski <jkrutkowski@elka.pw.edu.pl>  */
/*                                      */

#include <asm/unistd.h>
#include <errno.h>
#include "libpf.h"

_syscall1(int, patchfinder, int, what)


<--> ./patchfinder/libpf.c
```

ترجمه: سعید بیکی (cephexin@secumania.net)