SHELLCODE

Security Research Team – September 2006

Registration Weakness in Linux Kernel's Binary formats

Polluting sys_execve() in kernel space without depending on the sys_call_table[]

Table of Contents

Acknowledgements: **Ignacio Marambio Catán, Jonathan Sarba**

[Disclaimer]

The information exposed in this document comes with NO warranty. Neither SHELLCODE nor the authors of this document will be responsible or liable for any damages resulting from the missuse of the material disclosed in this document.

[Introduction]

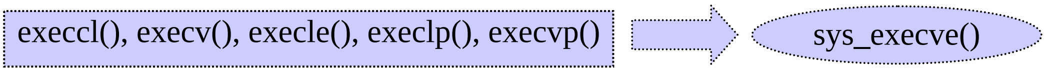
The present document aims to demonstrate a design weakness found in the handling of simply linked lists used to register binary formats handled by Linux kernel, and affects all the kernel families (2.0/2.2/2.4/2.6), allowing the insertion of infection modules in kernel-space that can be used by malicious users to create infection tools, for example rootkits.

Although, the use of the technique presented in the present document could to protect, prevent or control the execution, and be used to audit binaries, it can also be used to do just the opposite

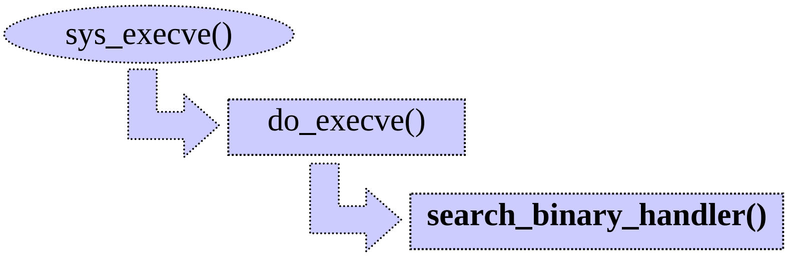
While the algorithm implemented in the kernel is ok, the context in which it is used is not so and can lead to problems such as the one discussed here.

[Description] exec() implementation

During the creation of new processes, the exec() family of functions is the one in charge of replacing the image of a process by another one; when a process invokes one of the following functions, the binary specified as argument is initialized through it's entry-point in the image already replaced.



The many functions of the exec() family are small wrappers to the system call sys_execve()[1]; this small function in turn calls the function do_execve()[1] which initializes the structure linux_binprm, creates the address space and credentials of the process, eventually it also invokes the function search_binary_handler()[2], and this is the function we will analyze throughout the rest of this document.



Perhaps it is surprising for more than one to know than kernel is the one in charge to recognize the different binary types existing in the system, identifying them with the first 4 (four) bytes of the file.

To do this, register_binfmt()[2] inserts each new binary format at the beginning of the global simply linked list, named formats[2], the linked list is implemented using pointers in a linux_binfmt[3] structure. This function has only suffered small changes since the 2.0 days and they were only SMP related changes.

[1] REF[ia32]: /arch/i386/kernel/process.c
[2] /fs/exec.c
[3] /include/linux/binfmt.h

/fs/exec.c

```
int register_binfmt(struct linux_binfmt * fmt)
{
    struct linux_binfmt ** tmp = &formats;
    if (!fmt)
        return -EINVAL;
    if (fmt->next)
        return -EBUSY;
    write_lock(&binfmt_lock);
    while (*tmp) {
        if (fmt == *tmp) {
            write_unlock(&binfmt_lock);
            return -EBUSY;
        }
        tmp = &(*tmp)->next;
    }
    fmt->next = formats;
    formats = fmt;
    write_unlock(&binfmt_lock);
    return 0;
}
```

[bold] Only kernel 2.4/2.6

It is important to highlight that the function only verifies if the new format exists or not in the list, therefore iterating through the list is not a problem in the proposed solution, the current function is already O(n)

[Analysis] The Binary Format list

From the analysis of the linux_binfmt[4] structure, we will focus on the load_binary[5] function, the one in charge actually execute the program originally specified in any of the exec() functions. The rest of the function pointers are only usefull to some of the binary formats.

Each binary type has its own load_binary[5] function which will be executed inside search_binary_handler()[5] in the following iteration.

/fs/exec.c

```
int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
{
        int try,retval;
        struct linux_binfmt *fmt;
        [..]
                for (fmt = formats ; fmt ; fmt = fmt->next) {
                        int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt-
>load_binary;
                        if (!fn)
                                continue;
[...]
```

The said iteration will continue as long as load_binary()[5] returns -ENOEXEC. New binary formats are declared using a static linux_binfmt[4] structure and are added on top of the formats list as discussed earlier.

[4] /include/linux/binfmt.h
[5] /fs/exec.c

/fs/exec.c (search_binary_handler)

```
[...]
                        if (retval != -ENOEXEC || bprm->mm == NULL)
                                break;
                        if (!bprm->file) {
                                read_unlock(&binfmt_lock);
                                return retval;
[...]
```

When declaring and inserting a new format, we must do it with its corresponding load_binary function, which will be always called each time we run sys_execve()6 before any other as long as out format is the last to be inserted in the list; it is only important for the return value of this function to be -ENOEXEC so that the originally executed program does it successfully; this will be the case since search_binary_handler7 will continue to search the list for the proper format when it sees the -ENOEXEC.

[6] REF[ia32]: /arch/i386/kernel/process.c
[7] /fs/exec.c

[POC] Inserting a new binary format

The following proof of concept shows the insertion of a new binary format using a kernel module. This code demonstrates how easy the creation of rootkits using this technique is.

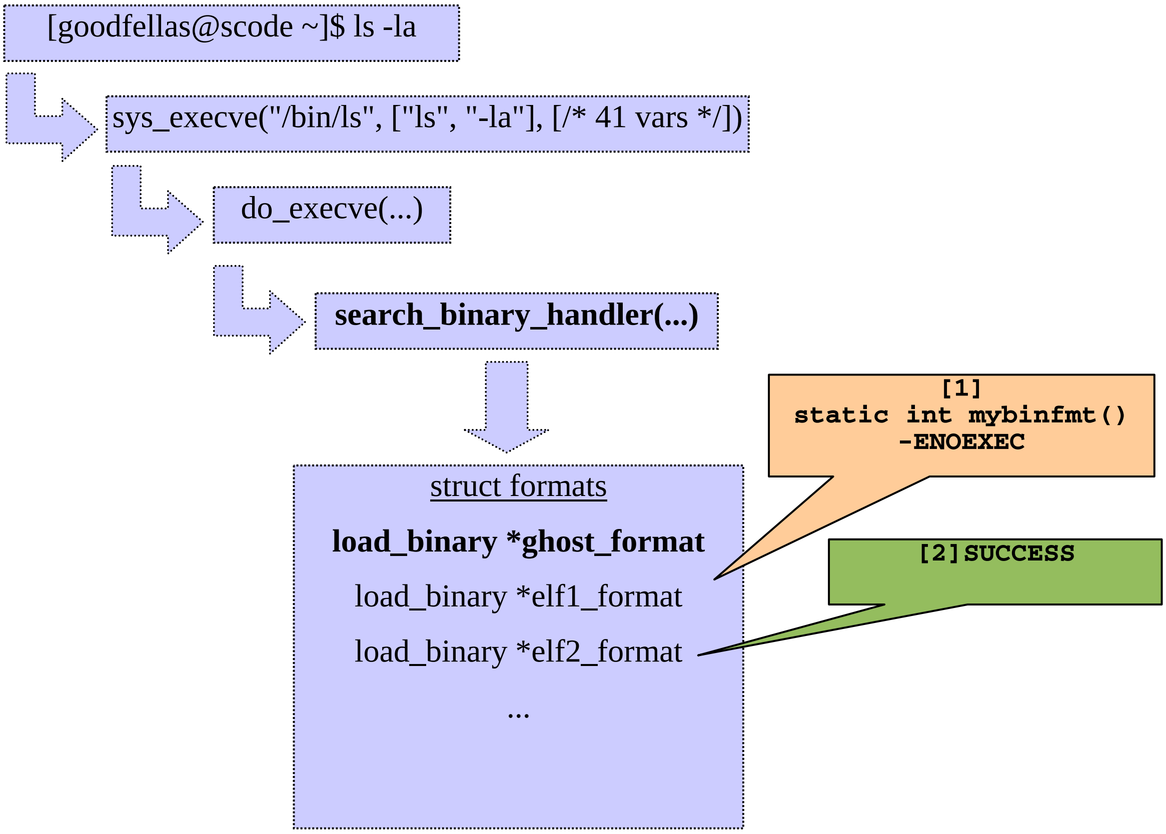
MODULE: ghost.c

```
#define DRIVER_AUTHOR "SHELLCODE Security Research Team"
#define DRIVER_DESC   "Registration Weakness in Kernel's Binary format"
MODULE_LICENSE("GPL");
static int mybinfmt(struct linux_binprm *bprm,struct pt_regs *regs){
    printk(KERN_ALERT "Hello not printable characters!\n");
    return -ENOEXEC;
}
static struct linux_binfmt ghost_format = {
        .module         = THIS_MODULE,
        .load_binary    = mybinfmt,
};
static int __init ghost_init(void){
    printk("Loading ghost\n");
    return register_binfmt(&ghost_format);
}
static void __exit ghost_exit(void){
    printk(KERN_ALERT "Unloading ghost\n");
    unregister_binfmt(&ghost_format);
}
module_init(hello_3_init);
module_exit(hello_3_exit);
```

Kernel: 2.6.x

- The technique may be used with root privileged only.
- The kernel patches like GRSEC8, and the kernel level security implementations like SELINUX9, do not contemplate the insertion of non valid binary formats nor the functions references in kernel-space for this list of binary formats.
- The technique was successfull on 2.4.x y 2.6.x kernels, the test cases were done using the following kernel versions: 2.4.33 y 2.6.17.11

The following is a small flow of execution once loaded the ghost.c module used during the tests of concept in our laboratory.


[goodfellas@scode ~]$ ls -la
sys_execve("/bin/ls", ["ls", "-la"], [/* 41 vars */])
do_execve(...)
search_binary_handler(...)
[1]
static int mybinfmt()
-ENOEXEC
struct formats
load_binary *ghost_format
[2]SUCCESS
load_binary *elf1_format
load_binary *elf2_format
...

[8] http://www.grsecurity.net/
[9] http://www.nsa.gov/selinux/

It is clear from this scheme that the load_binary function declared in our ghost format will be invoked before any other. This cannot be detected using tools of the type strace/ltrace, since the function acts in kernel-space, it is called by search_binary_handler()[10].

This affects not only those executed commands manually, it will as well be affecting all the processes of the system, considering that the call to sys_execve()[11] can be invoked quite frequently.

[10] /fs/exec.c
[11] /arch/i386/kernel/process.c

[Workaround] Kernel patching

Next we expose one of the possible kernel level solutions for 2,0, 2,2, 2,4 and 2,6 kernels, this solution requires the recompilation of kernel and system reboot with the new kernel image.

The modifications were directly made to register_binfmt()12, in the handling of the simply linked list called formats algorithm, the new elements would now be added to the list in the last position.

The presented differences were made only for the last kernel versions existing at the time of the conclusion of the present document, that is to say:

- **`Kernel 2.0.40`**
- **`Kernel 2.2.26`**
- **`Kernel 2.4.33`**
- **`Kernel 2.6.17.13`**

To apply the patch the file diff must be copied into the kernel source directory and then executing:

```
patch -p0 < file-diff.
```

It's possible that the following patches may be applied to previous versions of the same kernel familly (this should be checked diffing the files to ensure that the files have the same content)

[12] /fs/exec.c

PATCH: kernel 2.0.40

```
*** fs/exec.c   2006-09-26 18:17:52.000000000 -0300
--- fs/exec_new.c       2006-09-26 19:05:21.000000000 -0300
***************
*** 86,98 ****
                return -EINVAL;
        if (fmt->next)
                return -EBUSY;
!       while (*tmp) {
                if (fmt == *tmp)
                        return -EBUSY;
                tmp = &(*tmp)->next;
        }
!       fmt->next = formats;
!       formats = fmt;
        return 0;
  }

--- 86,102 ----
                return -EINVAL;
        if (fmt->next)
                return -EBUSY;
!       if (*tmp == NULL) {
!               formats = fmt;
!               return 0;
!       }
!       while ((*tmp)->next) {
                if (fmt == *tmp)
                        return -EBUSY;
                tmp = &(*tmp)->next;
        }
!       fmt->next = NULL;
!       (*tmp)->next = fmt;
        return 0;
  }
```

PATCH: kernel 2.2.26

```
*** fs/exec.c   2006-09-26 18:59:30.000000000 -0300
--- fs/exec_new.c       2006-09-26 19:00:54.000000000 -0300
***************
*** 94,106 ****
                return -EINVAL;
        if (fmt->next)
                return -EBUSY;
!       while (*tmp) {
                if (fmt == *tmp)
                        return -EBUSY;
                tmp = &(*tmp)->next;
        }
!       fmt->next = formats;
!       formats = fmt;
        return 0;
  }

--- 94,110 ----
                return -EINVAL;
        if (fmt->next)
                return -EBUSY;
!       if (*tmp == NULL) {
!               formats = fmt;
!               return 0;
!       }
!       while ((*tmp)->next) {
                if (fmt == *tmp)
                        return -EBUSY;
                tmp = &(*tmp)->next;
        }
!       fmt->next = NULL;
!       (*tmp)->next = fmt;
        return 0;
  }
```

PATCH: kernel 2.4.33

```
*** fs/exec.c   2005-01-19 11:10:10.000000000 -0300
--- fs/exec_new.c       2006-09-26 19:08:53.000000000 -0300
***************
*** 65,79 ****
        if (fmt->next)
                return -EBUSY;
        write_lock(&binfmt_lock);
!       while (*tmp) {
                if (fmt == *tmp) {
                        write_unlock(&binfmt_lock);
                        return -EBUSY;
                }
                tmp = &(*tmp)->next;
        }
!       fmt->next = formats;
!       formats = fmt;
        write_unlock(&binfmt_lock);
        return 0;
  }
--- 65,84 ----
        if (fmt->next)
                return -EBUSY;
        write_lock(&binfmt_lock);
!       if (*tmp == NULL) {
!               formats = fmt;
!               write_unlock(&binfmt_lock);
!               return 0;
!       }
!       while ((*tmp)->next) {
                if (fmt == *tmp) {
                        write_unlock(&binfmt_lock);
                        return -EBUSY;
                }
                tmp = &(*tmp)->next;
        }
!       fmt->next = NULL;
!       (*tmp)->next = fmt;
        write_unlock(&binfmt_lock);
        return 0;
  }
```

PATCH: kernel 2.6.17.13

```
*** fs/exec.c   2006-09-09 00:23:25.000000000 -0300
--- fs/exec_new.c       2006-09-26 19:10:52.000000000 -0300
***************
*** 76,90 ****
        if (fmt->next)
                return -EBUSY;
        write_lock(&binfmt_lock);
!       while (*tmp) {
                if (fmt == *tmp) {
                        write_unlock(&binfmt_lock);
                        return -EBUSY;
                }
                tmp = &(*tmp)->next;
        }
!       fmt->next = formats;
!       formats = fmt;
        write_unlock(&binfmt_lock);
        return 0;
  }
--- 76,95 ----
        if (fmt->next)
                return -EBUSY;
        write_lock(&binfmt_lock);
!       if (*tmp == NULL) {
!               formats = fmt;
!               write_unlock(&binfmt_lock);
!               return 0;
!       }
!       while ((*tmp)->next) {
                if (fmt == *tmp) {
                        write_unlock(&binfmt_lock);
                        return -EBUSY;
                }
                tmp = &(*tmp)->next;
        }
!       fmt->next = NULL;
!       (*tmp)->next = fmt;
        write_unlock(&binfmt_lock);
        return 0;
  }
```

[About]

SHELLCODE

We are experts in communication and IT security. We started our activities in 1999. Our team is built by experts with more than 10 years of experience working with technological complex structures and high performance networks.

We work on IT security with a structure dedicated to investigation and development, giving professional services and related solutions.

Shellcode Security Research Team

GoodFellas is our team in charge of research and development of new techniques and vulnerabilities, and the publication of new theories related to the existing platforms security in the IT world.

You can contact our team to send consults, comments and collaborations at:

GoodFellas@shellcode.com.ar.