

[UNIX] Linux Kernel do_brk() Vulnerability (Explained)

Source: <http://www.derkeiler.com/Mailing-Lists/Securiteam/2003-12/0020.html>

From: SecuriTeam (*support_at_securiteam.com*)

Date: 12/07/03

To: list@securiteam.com

Date: 7 Dec 2003 10:36:27 +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 do_brk() Vulnerability (Explained)

SUMMARY

Abstract:

A critical security bug has been found in the Linux kernel 2.4.22 (and earlier) memory management subsystem. This bug has been silently fixed for the 2.4.23 as well as in the 2.6.0-test6 release without any notice to the open source community.

While performing a regular audit of the Linux kernel we have found the same bug at the end of September 2003 and quickly realized its serious impact on the kernel security. Shortly after we were ready with a simple proof-of-concept exploit code.

The following paper presents the technical details of the do_brk() bug found and the results of our research done while writing the exploit code. It also describes the numerous techniques we have used to create a very effective exploit code that leads to privilege escalation even on systems running a kernel secured with various security patches.

DETAILS

Securiteam: [UNIX] Linux Kernel do_brk() Vulnerability (Explained)

Linux memory management:

The physical memory of a x86 machine running one of the recent Linux kernels is managed in a simplified flat virtual memory model. This means that each user process may address its virtual memory ranging from 0 up to 4GB on 32-bit architectures which is usually much more than the real physical memory installed.

Virtual memory is a linear address space divided into 4kB size pages. These pages are mapped into the physical memory pages using appropriate page table on a per process basis. The process's page table contains additional attributes for each mapped page including the page protection attributes.

The virtual memory of a process is divided into two regions. `TASK_SIZE` is a kernel constant that defines the upper limit of the accessible memory for the code working at the lowest privilege level.

Its value is usually set to `0xc0000000` on systems with less than 1GB of physical memory (all examples included in this article refer to this value). The memory above this limit contains the kernel code with its data structures and is not directly accessible to the user due to the page protection mechanism. It can be accessed only by privileged (kernel) code.

The user accessible memory region below the `TASK_SIZE` limit is furthermore divided into multiple logical sections. Each section is described by its virtual address range and protection attributes. Each section performs a different purpose. The section named `.text` contains the executable code of the binary loaded, the `.data` section contains the readable and writable data and `.rodata` contains the read-only data and so on.

A typical memory layout of a user process may look like:

```
bash$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 03:02 207935 /bin/cat
0804c000-0804d000 rw-p 00003000 03:02 207935 /bin/cat
0804d000-0804e000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:02 213752 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:02 213752 /lib/ld-2.3.2.so
40016000-40017000 rw-p 00000000 00:00 0
40020000-40021000 rw-p 00000000 00:00 0
42000000-4212f000 r-xp 00000000 03:02 319985 /lib/tls/libc-2.3.2.so
4212f000-42132000 rw-p 0012f000 03:02 319985 /lib/tls/libc-2.3.2.so
42132000-42134000 rw-p 00000000 00:00 0
bffc000-c0000000 rwxp fffd000 00:00 0
```

The memory sections are also known in the Linux kernel as the virtual memory areas (VMAs). The kernel keeps tracks and manages a list of all virtual memory areas for each process in order to provide proper memory management (swapping, demand loading and protection fault handling). Each virtual memory area is described by `vm_area_struct` as defined in `<linux/mm.h>`. Most important members of this structure are:

```
struct vm_area_struct {
```

```
unsigned long vm_start;
unsigned long vm_end;
pgprot_t vm_page_prot;
/* ... */
}
```

The virtual memory areas of a process are linked in the memory descriptor structure (mm_struct) which is referenced inside the process's descriptor (task_struct) by the mm member variable with roughly following structure:

```
struct mm_struct {
    struct vm_area_struct * mmap; /* list of VMAs */
    /* ... */
    int map_count; /* number of VMAs */
    /* ... */
    unsigned long start_brk, brk, start_stack;
    /* ... */
}
```

More details of Linux memory management are out of the scope of this article and can be found in [3].

The bug:

The do_brk() is an internal kernel function which is called indirectly to manage process's memory heap (brk) growing or shrinking it accordingly. The user may manipulate his heap with the brk(2) system call which calls do_brk() internally. The do_brk() code is a simplified version of the mmap(2) system call and only handles anonymous mappings for uninitialized data.

The do_brk() function lacks of any bound check of its parameter and may be exploited to create an arbitrary large virtual memory area exceeding the user accessible memory limit. Under normal circumstances the heap is a part of the process's virtual memory space and spans some kilobytes to megabytes of memory below the mentioned TASK_SIZE limit. It is usually used for keeping dynamically allocated data mostly through the malloc() library call. The missing bound check inside the do_brk() kernel function enables the expansion of the heap area above the TASK_SIZE limit.

Thus the kernel memory management subsystem can be tricked into believing that the protected kernel memory belongs to the user process's heap. This trick doesn't provide direct access to the kernel memory yet because the kernel pages are protected by the MMU unit of the CPU. However it is now possible to use other system calls to operate on the oversized VMA to disturb the protection of kernel pages.

Exploitation:

1) Attack vectors

The faulty do_brk() function is called inside the ELF and a.out binary format loaders as well as from the corresponding uselib() binary format handlers. Together with the sys_brk() call these are the three different

Securiteam: [UNIX] Linux Kernel do_brk() Vulnerability (Explained)

vectors which may be used to exploit the do_brk() bug. For the purpose of this article we are going to focus on the sys_brk() system call only.

2) Heap expansion

The heap may be expanded only if there is no other mapping in the requested address range. The regular process's stack is usually placed at the top of the process's memory right below the TASK_SIZE address, therefore it must be moved somewhere else before the exploitation can take place at all. Another step is to ensure that the heap is also the last section in the process's memory layout.

Now we may use the brk(2) system call to expand the heap to span the kernel memory. This must be done by calling brk multiple times, each time expanding the heap by a relative small amount of bytes. This is because we need to bypass a kernel limit on the virtual memory that may be mapped at once using do_brk() function.

After these three steps our heap may look like:

```
080a5000–ffff0000 rwxp 00000000 00:00 0
```

Unfortunately if our process is now terminated (exited or killed) in this state all VMA regions belonging to the process are cleaned, the memory pages unmapped and released to the kernel memory management. Thus parts of kernel memory may become inaccessible for all kernel control paths leading to system instability or immediate reboot.

3) Kernel memory protection

After expanding the heap region the pages above the 0xc0000000 boundary still cannot be directly accessed by the user process because all kernel memory pages are marked with the supervisor bit.

This unprivileged access to the pages is still prevented by the hardware MMU unit. The indirect access through ptrace(2) system call however could now be possible but we decided not to use this mechanism that is disabled on many or if not most Linux systems.

Therefore the kernel pages must be unprotected before accessing them. We need to make the kernel pages user readable and writable. Fortunately after short research we have discovered that the mprotect(2) system call works perfectly for kernel pages if the right VMA is present in the process's memory descriptor. And so we are able to selectively change protection of almost any page in the kernel.

However on x86 like processors with page size extensions (PSE) enabled the kernel code page size is equal to 4MB for performance reasons. The mprotect(2) system call doesn't handle such a big pages at all causing immediate crash. It may only be used on pages of 4kB size. Such pages are used by the kernel memory kmalloc() and vmalloc() allocators. The vmalloc() function is used i.e. to allocate memory for kernel modules.

Securiteam: [UNIX] Linux Kernel do_brk() Vulnerability (Explained)

With all the above information we are able to write anything to kmalloc'ed or vmalloc'ed kernel memory. Two main questions arise: what to write and where to write it to?

4) Kernel structures

We can use the kernel memory allocator to allocate some data structure that stays in the kernel memory for a while. We must find such a structure that would allow an easy privilege escalation after we modify its content.

The process's local descriptor table (LDT) holds an array of segment descriptors each of them describing segment limits and access privileges. This array is allocated through the vmalloc() allocator for each process that writes LDT entries using the modify_ldt(2) system call. The LDT stays in memory as long as the process is not terminated. The kernel provides limited ability to write entries into the LDT array. It protects against a misuse of LDT to prevent user process from gaining so called ring0 privileges. Thus if we were able to write into LDT array any arbitrary LDT entry we could escalate our privileges easily.

The Kernel memory layout varies from system to system. It depends on the kernel configuration as well as the compiler and the compilation options used. Address values returned by memory allocators are mostly unpredictable. So far this seems to be the hardest part of the exploit.

Our goal is of course not to guess anything. We want to find a way to determine the exact address of the mapped LDT array in the kernel memory. This part took most of time spent coding the exploit... and so we came over the Linux signal handling code.

If a signal is delivered to a process with a custom signal handler installed the signal handling routine receives information about the signal they caught, like the sender of the signal and the reason why the signal has been sent. The SIGSEGV signal is sent each time an user process tries to read or write to memory that is inaccessible from within process's context. Each page fault is handled by the do_page_fault() kernel function. One of its arguments is an error_code that is provided by the CPU.

This argument describes the exact reason of the page fault and is necessary to handle the page fault properly like loading the faulty page on demand, performing a copy-on-write or killing the process with the SIGSEGV signal in case of an invalid memory reference.

In the case of the SIGSEGV signal the kernel's do_page_fault() routine leaks its error_code value (un)intentionally to the signal handler. There are two error_code values that we are interested in:

- * a page fault occurred because the page was not mapped into memory
- * a page fault occurred because the page protection doesn't allow to access it

Thus the error_code value is suitable to determine whether an address above the TASK_SIZE limit has an underlying page mapped into the kernel

Securiteam: [UNIX] Linux Kernel do_brk() Vulnerability (Explained)

address space despite the fact that the page is not directly accessible to the user! This condition may be checked for each page above the

TASK_SIZE limit using for example the `verr` assembler instruction creating an exact map of the kernel memory. More details about the Intel instruction set can be found in [1].

If we create two maps of the kernel memory, the first before allocating and the second after allocating the kernel memory for the LDT array we can easily compare these maps and the result would be the exact address of the allocated kernel structure.

5) Privilege escalation

After finding our LDT array in kernel memory we can create there a call gate descriptor which enables privilege level transition from the user to the kernel privilege level.

An i386 call gate contains a code segment selector and an entry point to the gate code as well as descriptor privilege level. The code segment selector decides about the privilege level at which the code executed by the call gate is being run. On the other hand the descriptor privilege level decides about the necessary privilege level of calling code.

Call gates work in a similar way the `int $0x80` system call mechanism works which switches a regular process into kernel mode. The main difference to the system call interrupt is that with a user-space writable LDT we can just store there the address of an arbitrary routine that would be called at CPL0 privilege level. Details about the Intel privilege levels can be found in [2].

We decided to setup a call gate in the LDT with descriptor privilege level of 3 and the code segment equal to `KERNEL_CS` (which is the kernel code descriptor for CPL0) pointing back into the process's address space below `TASK_SIZE` thus allowing a user mode task to directly call its own code at CPL0. To perform this task an assembler trampoline code has been created which basically computes the pointer to the current process and calls a high-level C function which contains the actual exploit code.

While running at the most privileged CPL0 level is possible to change any kernel structure. Changing process's credentials is a quite easy task. The only thing we have to do is to find `task_struct` somewhere within kernel memory and then change its UIDs, GIDs and capability set. See `<linux/sched.h>` in kernel sources for detailed description of `task_struct`. However for the sake of simplicity only the EUID and EGID of the process must be changed if we execute another binary right after gaining EUID=0 because the `execve()` system call will reenable full process's capabilities if called with EUID=0.

6) Cleanup problem

After the privilege escalation into the CPL0 execution ring takes place a clean up code must be run in order to prevent the system from crashing and

Securiteam: [UNIX] Linux Kernel do_brk() Vulnerability (Explained)

to allow the process to terminate cleanly. Our idea is just to scan the kernel memory in a heuristic way for `vm_area_struct` structures that were expanded over `TASK_SIZE` limits. All those structures are changed to hold range up to `TASK_SIZE` and are sufficient to leave the system in a stable state.

ADDITIONAL INFORMATION

References:

[1] Intel Architecture Software Developer's Manual Volume 2 "Instruction Set Reference"

[2] Intel Architecture Software Developer's Manual Volume 3 "System Programming Guide"

[3] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel"

The information has been provided by <mailto:paul@isec.pl> Paul Starzetz independently discovered the `do_brk()` bug, <mailto:cliph@isec.pl> Wojciech Purczynski invented and provided numerous techniques to automatically and efficiently exploit the bug.

=====

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.