Report
System Software
Wintersemester 2009
CA644
Brian Stone

Hacking Linux Applications by implementing a new Syscall

cand. Dipl. Inf. Tobias Müller <muellet2@>, 59212333
BSc. Hugh Nowlan <nowlanh2@>, 59213060

20th November 2009

Abstract

This paper shows how to build a recent Linux kernel from scratch, how to add a new system call to it and how to implement new functionality easily. The chosen functionality is to retrieve the stack protecting canary so that mitigation of buffer overflow attacks can be circumvented.

Contents

1 Introduction

During the CA644 module in the winter semester of 2009, students were asked to implement a new linux system call (syscall) to give a normal user (i.e. non root) functionality that was unavailable beforehand. Without any prior knowledge of Kernel development, we implemented a new syscall which enables a user to obtain the stack protecting canary [Kuperman et al., 2005] of any process. While this does not necessarily have useful applications for a normal user, a hacker might want to know the canary value for a given process to overflow a buffer and thus overwrite the return address[1]. We will show how we have implemented this functionality, what pitfalls are there to avoid and how to finally use the gained ability.

Implementing a new system call (syscall) for the Linux kernel is interesting from a security as well as operating system point of view. Not only tightens it the understanding of how operating systems work with the environment such as processes running in userland or the machine it runs on but it also gives an idea what needs to be changed by an attacker in order to implement a dangerous attack. During the implementation of the system call in this paper it be came apparent how easily an attacker could insert code to implement their own system calls or other functions that could modify the behaviour of the system. While exploring the use of the `task_struct`[2] it was noted that changing the user IDs for the a process is a matter of changing a stored `int` value.

Having said that, the rest of this paper is structured as follows: Section 2 describes how to actually build a linux kernel from scratch. While section 3 shows how to generally

[1]for details on a buffer overflow attack cmp. [Aleph1, 1996]
[2]holding properties of the task, cmp. http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/include/linux/sched.h#L994

add a system call to the Linux kernel, section 4 illustrates how to add our desired functionality to the kernel. Finally, we conclude our findings in section 5.

2 Building a Kernel

There is plenty literature about Linux and its development and most of it includes a section on how to modify and compile the kernel as well. But as development of the Linux kernel is incredibly fast[Kroah-Hartman et al., 2008], the information is mostly outdated. The structure of the files, for example, has changed between the version Silberschatz uses in [Silberschatz, 2009] and the latest kernel version available to us[3]. The macros used in [Jones, 2007] do not exist anymore so that the examples do not work out of the box. A good tutorial in [macboypro, 2009] covers everything that is needed to add a new system call to the Linux kernel, but unfortunately, he does not mention how to implement syscalls that take an argument. As it turns out, the necessary `syscall` function is variadic so that one it replaces the `__syscall1` style macros mentioned in older literature.

To actually build a working Linux kernel, one need to follow a simple, but sometimes tedious recipe:

1. Obtain Kernel source code.
   The source code of the Linux kernel is kept on `http://www.kernel.org`. To receive a package with the source code, simply download it with, e.g. `wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.6.tar.bz2`
   In order to unpack the source code, use `tar xf linux-2.6.31.6.tar.bz2`

2. Install build dependencies.
   These are the tools one needs to actually compile the kernel, which is a huge C program. On a Debian based Linux distribution[Ubuntu Community, 2009], this can be done with the command `apt-get build-dep linux`

3. Configure the kernel.
   To select the features and driver that shall be compiled with the kernel. Since Linux can run on many platforms, ranging from tiny embedded devices to high performance multi-processor server, there are many options to choose from. As different people have different needs, you can select virtually any feature to be built or not. This can be confusing and time consuming since rebuilding a kernel, booting it, testing whether the configuration was correct can take a long time. `make menuconfig` provides an interactive menu to configure the Linux kernel. Also, a mechanism to use an old configuration (i.e. the one that the currently running kernel was built with) is available via `make oldconfig`

4. Booting the kernel.
   To check whether the hardware was detected correctly or, if the system call is already implemented (described in section 4), to activate the added syscall. To make

[3] 2.6.31 at the time of writing

the new kernel known on a system running GRUB as bootloader, it is sufficient to run `make install`

2.1 Pitfalls

Compiling a Linux kernel is fraught with pitfalls. When first compiling the kernel using `make oldconfig` the kernel did not include many of the options the stock Debian kernel included and therefore the system refused to boot, unable to find the SCSI hard drive. Once recompiled with the relevant flags and installed, the system booted but without network access. The problem here had been forgetting to install the compiled modules. This was fixed with a simple `make modules_install` and a reboot.

3 Adding a system call

3.1 The call code

The system call code can be added in an arbitrary location. There are a variety of places within the kernel source tree that would be a suitable home for an added system call, depending on the purpose it serves. In the case of this project it was decided that a folder would be created in the root of the source tree for clarity. The path of the new folder was added to the "core-y" make rule and a Makefile was added to the directory to ensure the new code was linked into the kernel. The code was denoted as being C code by the addition of the `asmlinkage` flag.

3.2 Defining the call in the kernel

Once written, the call needs to be defined in system call tables. Assuming being in the directory of the unpacked kernel source code (cmp. section 2), the system call number is defined in `./arch/x86/include/asm/unistd_32.h` by adding a line like `#define __NR_writelog 333`, where 333 is one number larger than the current highest system call number. Then the system call's prototype should be added to `./arch/x86/include/asm/syscalls.h`, within the `X86_32` if statement. This is not vital but will reduce compilation warnings and make the code more well integrated. The linking of call name to the symbol table is done in `./arch/x86/kernel/syscall_table_32.h`. When all the correct modifications have been made the call is ready to be tested and used.

4 Implementing new syscall(s)

4.1 Writelog

It was decided to implement a system call that logs every write of a given process to the kernel log. The idea is similar to the one stated more than ten years ago in [halflife@infonexus.com, 1997] but instead of snooping on a terminal, our write logger should monitor any writes from any given process. Our system call is thus named sys_writelog and the necessary code to implement the desired functionality is shown in figure 1.

In order to see whether our implementation actually works, we wrote a small programm that simply calls the new syscall from userland.

The structure of the test program shown in figure 2 is simple: It gets its own process id and calls the new systemcall with it. The program then writes a secret string to a file (stdout in this case) and resets the logging functionality.

Sadly, the syscall turned out to be not working. It does not run beyond line 60 most probably due to a read-only mapped system call table.

Since the kernel has almost full control over the underlying hardware it must be possible to remap the relevant memory segment. So instead of stating the obvious, we decided to explore other features of the kernel and moved on to attempt the construction of a PID changing system call.

4.2 PID Changer

Initially, the model was to change any given process ID to an arbitrary value but we later decided to set it so that the selected process could change its own PID to another value. This does most probably not have any useful applications but rather a cosmetic effect. An attacker might want to set the IDs of her processes to a "cool" value such as 31337, 2342 or -1. It also allows to study how the kernel manages unexpected modifications to its data structures including possibly non unique process IDs.

The call was named sys_pidchanger and the code implementing the function is shown in figure 3. Another function to change the PID of any given process directly via the task_struct datastructure was implemented but this was found to result in the process disappearing from the process list. This is most likely due to the PID of a process being an internal and external value. When it is changed in the program, commands such as ps do not know how to gather information about the process. The process continues to run but the new ID is hidden from process lists. In theory this could be exploited by an attacker to hide malicious programs, although software such as rkhunter [rkh, 2009] can reveal the existence of programs with aberrant process/resource relations via the proc file system and other methods.

In order to test the syscall, a program was written that invokes the system call upon itself. The program runs in an infinite loop, printing out its own process ID, calling the sytem call and writing that ID to the standard output. The __NR_getpid system call was used after the first getpid() as, during development, we noted that getpid() caches the result of the first call as in most cases, it is not expected that a process ID will change.

Since we did not want to remove any caching, we moved on to implement yet another system call.

```
extern void *sys_call_table[];
static int snoopedpid = 0;
static void *original_write;

asmlinkage long
new_write(unsigned int fd, const char __user *buf, size_t count) {
  int pid = current->pid;

  if(pid == snoopedpid) {
    printk(KERN_INFO "fd: %d buf: %s count: %d\n", fd, buf, count);
  }
  /* restore in order to write intercepted data */
  sys_call_table[__NR_write] = original_write;
  ssize_t result = sys_write(fd, buf, count);
  sys_call_table[__NR_write] = new_write;

  printk(KERN_EMERG "Finished write cycle\n");

  return result;
}

/* Sets up logging of write syscalls */
asmlinkage long
sys_writelog(int pid) {
  printk(KERN_EMERG "PID is %d\n", pid);

  if(original_write == NULL) {
    original_write = sys_call_table[__NR_write];
    printk(KERN_EMERG "Got original write\n");
  }

  snoopedpid = pid;

  if(pid == 0) {
    /* disable writelog, restore original write */
    printk(KERN_EMERG "Disabling writelog\n");
    sys_call_table[__NR_write] = original_write;
    printk(KERN_EMERG "Writelog disabled\n");
    return 1;
  } else if (pid > 0) {
    printk(KERN_EMERG "Performing new_write\n");
    sys_call_table[__NR_write] = new_write;
    printk(KERN_EMERG "Made it past the new_write\n");
    return 0;
  } else {
    printk(KERN_CRIT "PID of %d - no action taken\n", pid);
    return -1;
  }
}
```

Figure 1: Code to implement a write logger (without includes, full file attached to this PDF)

```
#define __NR_writelog   337    /* or whatever you set it in unistd.h */

int
main () {
  /* enable the writelog */
  syscall(__NR_writelog, getpid());

  fprintf(stdout, "%s", "My secret string\n");

  syscall(__NR_writelog, 0); /* Disable writelog */
  return 0;
}
```

Figure 2: Program which uses the new systemcall

```
/* Changes the PID of a process */
asmlinkage long
sys_pidchange(int pid) {
  printk(KERN_EMERG "PID %d\n", current->pid);

  current->pid = pid;
  printk(KERN_EMERG "PID changed to %d\n", current->pid);

  current->group_leader->pids[PIDTYPE_PID].pid = pid;
  printk(KERN_EMERG "PID also changed to %d\n",
                            current->group_leader->pids[PIDTYPE_PID].pid)
                                ;
  return current->pid;
}

asmlinkage long
sys_pidchange2(int oldpid, int newpid) {
  struct task_struct *thetask = find_task_by_vpid(oldpid);

  printk(KERN_EMERG "PID %d\n", thetask->pid);
  thetask->pid = newpid;
  printk(KERN_EMERG "PID changed to %d\n", thetask->pid);

  return newpid;
}
```

Figure 3: Code to implement a process ID changer (without includes, full file attached to this PDF)

```
#define __NR_pidchange          337

int main() {
  while(1) {
    pid_t mypid = getpid();
    printf("My PID is %d\n", mypid);
    syscall(__NR_pidchange, mypid, 31337);
    mypid = getpid();
    printf("Suddenly my PID is %d\n", mypid);
  }
  return 0;
}
```

Figure 4: Program which uses the PID changer system call

4.3 Stack Canary

When issues were encountered with the PID changing code, it was decided to attempt a more concrete but similarly security-minded problem. The stack canary is an optional feature of the Linux kernel, enabled via the CONFIG_CC_STACKPROTECTOR configuration option. If an attacker is attempting to access the stack via a buffer overflow vulnerability or other attack, the canary will most likely be accessed while enumerating memory. When this happens, the program in question is ended immediately, stating a segmentation fault.

The call was named `sys_getcanary` and the code implementing the function is shown in figure 5.

In order to test the syscall, a program shown in figure 6 was written that invokes the system call upon itself if not given an argument or upon another process ID if given as an argument. Figure 7 demostrates the results showing that it successfully obtains the stack of any given process, even if it is owned by the root user. With this information, an attacker would be able to successfully overflow a buffer in the targeted program, without any protective mechanism noticing it.

5 Conclusion

We have shown how to build a 2.6.31 Linux kernel and that it is nearly trivial to add malicious functionality. Although an attacker needs elevated privileges in order to install a malicious kernel, it does not mild the threat, since the described technique helps the attacker to keep his privileges and thus circumvent security machnisms. Although not every approach we have taken lead to success immediately, it is just a matter of knowledge to implement more sophisticated attacks than our successfully implemented syscall.

```c
/* Changes the PID of a process */
asmlinkage long
sys_getcanary (int pid) {

#ifdef CONFIG_CC_STACKPROTECTOR
  if (pid == 0) {
    return current->stack_canary;
  } else {
    struct task_struct *thetask = find_task_by_vpid(pid);
    /* printk(KERN_EMERG "Stack canary is at %x\n", thetask->stack_canary)
        ; */
    return thetask->stack_canary;
  }
#else
  /* printk(KERN_EMERG "Please overflow our buffers! No canary found!");
      */
  return -1;
#endif
}
```

Figure 5: Code to retrieve the canary of a process (without includes, full file attached to this PDF)

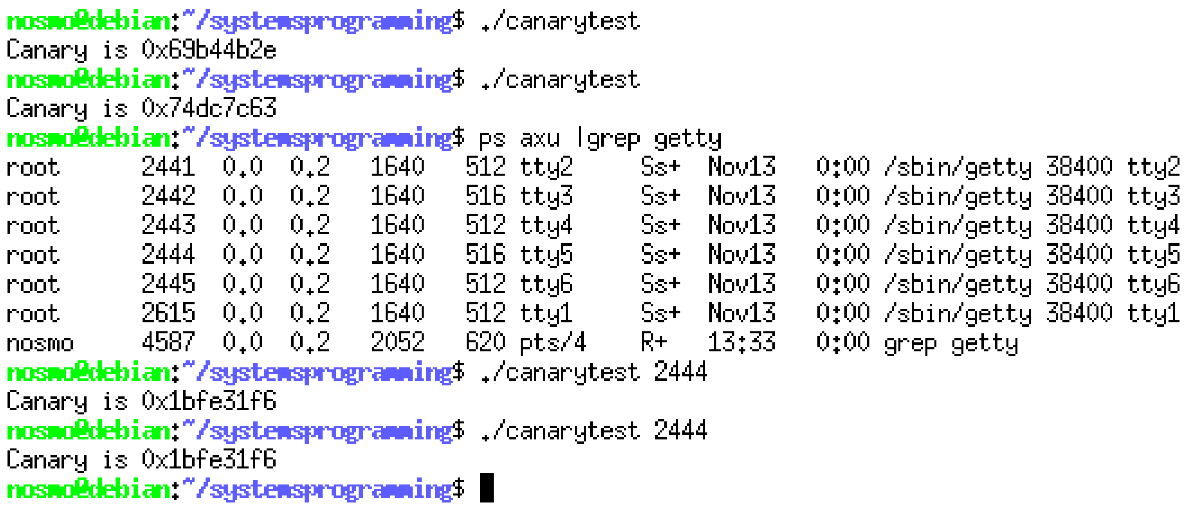
```c
#define __NR_getcanary          337

int main(int argc, char* argv[]) {
  int process = getpid();

  if(argc > 1)
    process = atoi(argv[1]);

  long value = syscall(__NR_getcanary, process);
  printf("Canary is 0x%x\n", value);
  return 0;
}
```

Figure 6: Program which uses the canary retrieving syscall



```
nosmo@debian:~/systemsprogramming$ ./canarytest
Canary is 0x69b44b2e
nosmo@debian:~/systemsprogramming$ ./canarytest
Canary is 0x74dc7c63
nosmo@debian:~/systemsprogramming$ ps axu |grep getty
root      2441  0.0  0.2   1640   512 tty2     Ss+  Nov13   0:00 /sbin/getty 38400 tty2
root      2442  0.0  0.2   1640   516 tty3     Ss+  Nov13   0:00 /sbin/getty 38400 tty3
root      2443  0.0  0.2   1640   512 tty4     Ss+  Nov13   0:00 /sbin/getty 38400 tty4
root      2444  0.0  0.2   1640   516 tty5     Ss+  Nov13   0:00 /sbin/getty 38400 tty5
root      2445  0.0  0.2   1640   512 tty6     Ss+  Nov13   0:00 /sbin/getty 38400 tty6
root      2615  0.0  0.2   1640   512 tty1     Ss+  Nov13   0:00 /sbin/getty 38400 tty1
nosmo     4587  0.0  0.2   2052   620 pts/4    R+   13:33   0:00 grep getty
nosmo@debian:~/systemsprogramming$ ./canarytest 2444
Canary is 0x1bfe31f6
nosmo@debian:~/systemsprogramming$ ./canarytest 2444
Canary is 0x1bfe31f6
nosmo@debian:~/systemsprogramming$ █
```

Figure 7: Example of the canary system call being used in an application. The canary changes for new applications but it shown to remain constant for the chosen getty process

References

[rkh, 2009] (2009). Rootkit hunter. http://www.rootkit.nl/projects/rootkit_hunter.html.

[Aleph1, 1996] Aleph1 (1996). Smashing the stack for fun and profit. *Phrack Magazine*, 7(49). http://www.phrack.org/issues.html?id=14&issue=49.

[halflife@infonexus.com, 1997] halflife@infonexus.com (1997). Abuse of the linux kernel for fun and profit. *Phrack Magazine*, 7(50). http://www.phrack.org/issues.html?issue=50&id=5.

[Jones, 2007] Jones, T. (2007). Kernel command using linux system calls. http://www.ibm.com/developerworks/linux/library/l-system-calls/.

[Kroah-Hartman et al., 2008] Kroah-Hartman, G., Corbet, J., and McPherson, A. (2008). Linux kernel development (April 2008). https://www.linuxfoundation.org/publications/linuxkerneldevelopment.php.

[Kuperman et al., 2005] Kuperman, B. A., Brodley, C. E., Ozdoganoglu, H., Vijaykumar, T. N., and Jalote, A. (2005). Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56.

[macboypro, 2009] macboypro (2009). Adding a custom system call to ubuntu linux in [C] « tech today. http://macboypro.wordpress.com/2009/05/15/adding-a-custom-system-call-to-the-linux-os/.

[Silberschatz, 2009] Silberschatz, A. (2009). *Operating system concepts*. J. Wiley & Sons, Hoboken NJ, 8th ed. edition. `http://www.os-book.com/`.

[Ubuntu Community, 2009] Ubuntu Community (2009). Kernel/compile - community ubuntu documentation. `https://help.ubuntu.com/community/Kernel/Compile`.

License