

# Sicherheitsrelevante Programmierfehler (Teil 2)

[Thomas Biege](#)

## Race Conditions (nicht auf C beschränkt!)

*Race Conditions* können überall da entstehen, wo **nicht-atomare** Aufrufe für sicherheitsrelevante Programmabschnitte verwendet werden. Hier soll nur eine klassische und häufig auftretende Art von *Race Conditions* beschrieben werden. Grundsätzlich können *Race Conditions* in den verschiedensten Facetten auftreten und beschränken sich nicht nur auf das Filesystem.

Beispiele:

Wenn ein privilegiertes Programm eine Datei öffnet, die dem Benutzer gehört, der das Programm aufruft, muß vor dem Öffnen der Datei überprüft werden, ob der Benutzer das überhaupt darf.

Fehler:

```
[...]
if(access("/home/evil_ed/RythmStick", W_OK) == 0)
{
    /* Benutzer darf schreiben */
    if((fd = open("/home/evil_ed/RythmStick", O_WRONLY)) < 0)
    {
        fprintf(stderr, "Ich darf die Datei nicht
                                öffnen!\n");
        exit(-1);
    }
}
[...]
```

Erst müssen die verschiedenen benutzerbezogenen IDs von Unix erklärt werden:

### User ID

Jeder Benutzer erhält eine eindeutige Nummer. Die UID 0 gehört root. Wer mit der UID 0 arbeitet, wird von keiner Sicherheitsschranke des Systems aufgehalten. Normale Benutzer erhalten in der Regel UID ab einen bestimmten Wert, z.B. 100 oder 500. Alle darunter liegenden UID's gehören Systemprozessen.

### Group ID

Unix-Systeme haben mehrere Gruppen. Die Administratoren können beispielsweise der Gruppe root oder wheel zugeordnet sein. Benutzer sind meistens in der Gruppe user oder werden Gruppen für ihren Tätigkeitsbereich zugeordnet, z.B. fb4, wwwadmin, students, accounting etc. Ein Benutzer kann 1...NGROUPS\_MAX (32) Gruppen angehören.

### SetUID und SetGID

Damit ein Programm eine Aufgabe erledigen kann, für die es höhere Rechte benötigt, kann entweder ein Benutzer mit diesen Rechten das Programm ausführen, oder dem System wird gesagt, daß dieses Programm die benötigten Rechte beim Ausführen zu bekommen hat. Der erste

Fall ist sehr unbequem und unsicher, da entweder jeder das Paßwort kennen müßte oder sich die Person, die das Paßwort weiß, in das System einloggen muß, um eine Aufgabe für einen anderen Benutzer zu erledigen. Im Falle eines Drucker- oder Fax-Spoolers wäre das nicht vertretbar. Unix hat hierfür extra zwei Flags in der Filesystemimplementierung. Das **SetUID-Flag** gibt dem Programm beim Ausführen die (effektive) UID des Benutzers, dem das Programm gehört und nicht die UID des Benutzers, der das Programm ausführt (reale UID). Das gleiche gilt für das **SetGID-Flag**.

### reale UID/GID und effektive UID/GID

Die reale UID ist die UID des Benutzers. Führt Benutzer Thomas mit der UID 543 beispielsweise ein SetUID Programm vom Benutzer root aus, so hat das Programm die effektive UID 0 und die reale UID 543. Da für die Zugangsüberprüfung auf Systemobjekte in Unix-Systemen die effektive UID benutzt wird, können die UIDs getauscht oder sogar die effektive UID verworfen werden. Ist die reale UID nicht 0, so kann eine verworfene UID nicht zurückgeholt werden, eine getauschte UID hingegen schon.

### saved UID/GID (POSIX)

Die saved IDs sind nur implementiert, wenn `_POSIX_SAVED_IDS` gesetzt ist, was bei modernen Unix-Derivaten der Fall sein sollte. Die saved UID wird von den `exec*(3)` Funktionen gesetzt und ermöglicht einen Wechsel der UID. Ist die reale und effektive UID beispielsweise 543 und die saved UID 0, dann ist ein Wechsel, obwohl die effektive UID nicht 0 ist, möglich. Die saved UID wird nur verworfen, wenn `setuid(2)` aufgerufen wird und die effektive UID 0 ist. Zudem müssen SetUID Applikationen, deren SetUID ungleich 0 ist, `setreuid(getuid(), getuid())` benutzen, um ihre Privilegien loszuwerden; ein einfaches `setuid(2)` hilft nicht. Dasselbe gilt natürlich auch für die GID.

Der `access(2)` Aufruf benutzt die **reale UID** und **reale GID** zum Prüfen der Rechte. Das heißt, daß die **effektive UID/GID** von **SetUID/-GID** Programmen nicht zum Tragen kommt. Bei der Zugriffüberprüfung mit `open(2)` hingegen wird die **effektive UID/GID** benutzt. Dieser Umstand wäre nicht weiter schlimm - ja, sogar erwünscht - wenn `access(2)` und `open(2)` eine atomare Funktion, also ein Systemaufruf, wäre. Zwischen `access(2)` und `open(2)` besteht aber nun ein Zeitfenster, in dem das Programm verwundbar ist. Der Angreifer kann nun die Datei `/home/evil_ed/RythmStick` löschen und durch einen Link ersetzen, der beispielsweise auf `/etc/security/shadow` zeigt. Der `open(2)` Aufruf öffnet also dann nicht `/home/evil_ed/RythmStick`, sondern über den Verweis `/etc/security/shadow` und verarbeitet die Daten, an die der Benutzer eigentlich nicht Hand anlegen darf.

Also nochmal zur Verdeutlichung:

Die Datei `/home/evil_ed/RythmStick` existiert und User EvilEd hat darauf Schreibrechte, das wird mit `access(..., W_OK)` geprüft. Jetzt löscht EvilEd die originale Datei und setzt eine Datei-Verknüpfung auf eine Datei, auf die er keine Schreibrechte hat. `Open(2)` folgt dem Link und öffnet die geschützte Datei.

Korrektur:

Man kann diesem Problem auf verschiedene Art und Weisen begegnen.

### *faccess()*

Leider existiert `faccess()` nicht auf allen System, z.B. nicht auf *Linux*, *OpenBSD*, *Solaris* und *AIX*.

```
[...]
fd = open("datei", O_WRONLY);
if(faccess(fd, W_OK) != 0)
{
```

```

    fprintf(stderr, "Netter Versuch!\n");
    exit(-1);
}
[...]
```

Ein weiterer Nachteil von *faccess()* ist, daß vorher ein *open(2)* gemacht werden muß. Wenn *open(2)* auf Gerätedateien (Unix) angewandt wird, kann je nach Implementierung des Gerätetreibers bereits eine Aktion mit dem entsprechenden Gerät durchgeführt werden. Als Beispiel sind Magnetbänder zu nennen, die zurückgespult werden, wenn die Geräte-datei geöffnet wird. Bei iterativen Backups kann dies zum Verlust der alten Backupdaten führen.

## ***O\_NOFOLLOW***

Die *open(2)* Option *O\_NOFOLLOW* verbietet es *open(2)*, **symbolischen Links** zu folgen. Sie kann aber mit **Hard-Links** ausgetrickst werden und nützt deswegen in diesem Fall wenig.

## ***setegid(2)* und *seteuid(2)***

Mit *seteuid(2)* und *setegid(2)* lassen sich für den Zeitraum, in dem *open(2)* ausgeführt wird, die erhöhten Privilegien des Programms verwerfen und die Rechte des Nutzers können angenommen werden. Nach *open(2)* können die alten Rechte wieder hergestellt werden.

Die *sete(u|g)id(2)* bzw. *setre(u|g)id(2)* Funktionen sind konform zu *BSD4.3* und sind im erweiterten *POSIX* Standard enthalten, wenn *\_POSIX\_SAVED\_IDS* definiert ist. Das Betriebssystem muß zusätzlich zu den **realen** und **effektiven IDs** also auch **saved IDs** unterstützen. Dabei ist darauf zu achten, daß die **Gruppen ID vor der User ID** herabgesetzt wird, da andernfalls bei bereits verringerter UID die GID nicht mehr geändert werden kann. Die Tatsache wird leider oft vergessen, und das Programm enthält dann immer noch ein Sicherheitsloch. Zudem sollte immer der Rückgabewert von *setuid(2)* und *setgid(2)* überprüft werden, um auf unliebsame Ereignisse reagieren zu können.

```

[...]
```

```

uid_t  euid, ruid;
gid_t  egid, rgid;

euid = geteuid();
egid = getegid();
ruid = getuid();
rgid = getgid();

if(setegid(rgid) < 0)
    [Exit]
if(seteuid(ruid) < 0)
    [Exit]

open("...", ...);

if(setegid(egid) < 0)
    [Exit]
if(seteuid(euid) < 0)
    [Exit]
[...]
```

## ***fork(2)***

Der einzig portable und sicherste, aber auch umständlichste Weg besteht darin, einen **Child-Process** mit *fork(2)* zu erzeugen, die Privilegien dauerhaft zu verwerfen, die Datei zu öffnen und nach der Rückgabe des Filedeskriptors an den **Parent-Process** den **Child-Process** zu beenden.

```

[...]
```

```

pid_t  child_pid;
[...]
```

```

if((child_pid = fork()) < 0)
    [Exit]
else if(child_pid > 0) // Parent
    [Get Filedescriptor and Wait on Child]
else // Child
{
    int fd;

    if(setgid(getgid()) < 0)
        [Exit]
    if(setuid(getuid()) < 0)
        [Exit]
    /*
    ** Wenn die EUID des Prozesses 0 ist, dann werden
    ** mit set(u|g)id(2) _alle_ IDs (real, effektiv,
    ** saved) geändert!
    ** Andernfalls wird _nur_ die effektive ID gesetzt!
    */

    if( (fd = open("userfile", O_WRONLY)) < 0)
        [Exit]

    /*
    ** Nun kann der Filedeskriptor dem Parent-Process
    ** übergeben werden. Leider gibt es dafür keine
    ** standardisierte Funktion. SystemV und BSD Unix
    ** bieten unterschiedliche Möglichkeiten.
    ** SVR4:
    ** - ioctl(I_SENDFD)
    ** - Unix Domain Sockets
    ** 4.3BSD:
    ** - sendmsg(2) and recvmsg(2)
    ** - Unix Domain Sockets
    */
    [...]
}
[...]
```

