

Sicherheitsrelevante Programmierfehler (Teil 4)

[Thomas Biege](#)

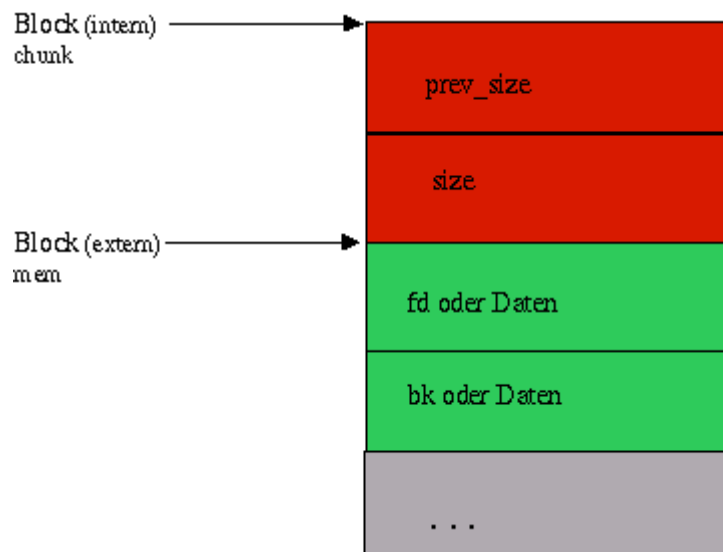
Inhaltsverzeichnis

- [Dynamischer Speicher und free\(3\)](#)
- [Chroot-Umgebungen \(nicht auf C beschränkt!\)](#)

Dynamischer Speicher und free(3)

Kurz nachdem der sicherheitsrelevante Einfluß von Format-Bugs gezeigt wurde, kam ein weiterer interessanter Bug ans Tageslicht. Für ein besseres Verständnis muß erst die Funktionsweise von *[m,c,re]alloc(3)* und *free(3)*, wie es in den meisten Linux-Systemen implementiert ist, (grob) erläutert werden.

Der Speicher wird mit Hilfe von Speicherblöcken in einer doppelt verketteten Liste verwaltet. Ein Block sieht folgendermaßen aus:



Intern arbeitet *malloc(3)* mit dem Zeiger *chunk*. Der Programmierer bekommt den Zeiger *mem = ((char *) chunk) + 8*, der direkt auf den dynamisch allozierten Speicher zeigt.

Wenn ein Listenelement bzw. Speicherblock (*chunk*) unbenutzt ist, also nicht auf allozierten Speicher verweist, dann werden *fd* und *bk* als Zeiger für die Listenelemente benutzt, ansonsten enthalten sie Daten.

Die Größe des Vorgängerblocks wird durch *prev_size* wieder gegeben, *size* enthält die Größe des aktuellen Blocks.

Wird nun *free(3)* benutzt, um den Speicher wieder freizugeben, muß zunächst der Zeiger *chunk* durch simple Subtraktion (8 Byte auf 32 Bit Architekturen) errechnet werden. Dieser Zeiger wird an die interne Funktion *chunk_free()* übergeben.

In dieser Funktion wird der Block wieder in die doppelt verkettete Liste eingefügt, nachdem sichergestellt wurde, daß der Vorgänger und Nachfolger des aktuellen Blocks ebenfalls nicht benutzt wird. Ob ein Block benutzt wird oder nicht, kann durch eine logische *UND-Verknüpfung* zwischen dem Element *size* und der Konstanten *PREV_INUSE* festgestellt werden.

Um den Block wieder in die Kette einzubinden, werden die Zeiger *fd* und *bk* des Vorgängerblocks und Nachfolgerblocks benötigt. Für das Verbiegen der Zeiger wird das Makro *unlink()* verwendet.

```
#define unlink(P, BK, FD) \
{ \
    BK = P->bk; \
    FD = P->fd; \
    FD->bk = BK; \
    BK->fd = FD; \
}
```

Ist es einem Angreifer möglich, das Argument für *free(3)* frei zu definieren, dann ist er in der Lage über das Makro *unlink()* den Speicherinhalt zu verändern. Durch geschicktes Setzen der Zeiger *fd* und *bk* können Funktionszeiger, Rücksprungadressen, Variableninhalte usw. modifiziert werden, um die Sicherheit eines Systems zu kompromitieren.

Es ist also sehr darauf zu achten, daß Aufrufe von *free(3)* sich immer nur auf tatsächlich allozierte Speicherbereiche beziehen!

Chroot-Umgebungen (nicht auf C beschränkt!)

Mit dem *chroot(2)* Systemaufruf kann die Sicht eines Prozesses auf das Dateisystem verändert werden.

Aus Sicherheitsaspekten werden Daemonprozesse (Server) häufig (leider nicht häufig genug) in ein sog. *Chroot-Jail* verbannt, d.h. daß von privilegierten Prozessen die *UID* und *GID* geändert wird, der Prozeß in ein speziell prepariertes Verzeichnis wechselt, und dieses Verzeichnis das neue Wurzelverzeichnis (Root Directory) wird.

Wird nun ein Softwarebug in diesem Serverprogramm ausgenutzt, dann kann der Angreifer nur mit den verringerten Benutzerrechten und auch nur in dem *Chroot-Verzeichnis* agieren. Der Zugriff auf darüberliegende Verzeichnisstrukturen wird ihm verwehrt.

Da der Daemonprozeß aber zur Laufzeit auf bestimmte Ressourcen im Filesystem zugreifen muß, müssen Shared Libraries (werden nicht benötigt, wenn das Programm statisch übersetzt wurde) Geräte-, System- und Konfigurationsdateien in das *Chroot-Verzeichnis* kopiert werden.

Bei Aufsetzen von *Chroot-Umgebungen* werden nicht selten schwerwiegende Fehler gemacht.

Sensible Daten

Einige Programme, z.B. FTP Server, benötigen die Paßwortdatei, um die Besitzer und Gruppen von Dateien korrekt anzuzeigen. Aus diesem Grund wurde früher häufig die Datei */etc/passwd* einfach nach *~fptd/etc/* kopiert. Ein Angreifer konnte sich nun über anonymes FTP einfach einloggen, die *passwd* Datei auf seinen Rechner übertragen und versuchen, die Paßwörter zu knacken.

Falls nicht unbedingt nötig, sollten niemals sensible Daten in die Chroot-Umgebung gelangen.

Gerätedateien (Special Files)

Wenn das Serverprogramm Gerätedateien benötigt, sollte man genau auf die Art der Geräte achten. Geräte wie `/dev/zero` und `/dev/null` sind unproblematisch, sind aber Special Files vorhanden, die Zugriff auf den Speicher (z.B. `/dev/kmem`) oder die Festplatte (z.B. `/dev/sdb3`) und dergleichen erlauben, dann kann ein Angreifer, je nach Zugriffsrechten, sensible Daten lesen, neue Accounts einrichten oder den Kernel des Betriebssystems verändern und somit weiteren Zugang zum System erlangen.

Privilegien

Werden die Rechte des Servers nicht komplett verworfen, läuft das Programm also noch mit Rootrechten, ist ein Ausbruch aus dem *Chroot-Jail* kein Problem.

Viele Administratoren und Programmierer fühlen sich sicher, wenn sie ihrem Server als neue *UID/GID nobody* o.ä. zuweisen. Das Problem entsteht, wenn diese IDs nicht explizit von diesem Server, sondern auch von anderen Programmen des Systems verwendet werden.

Der Angreifer kann nun über Signale die anderen Programme stören, anhalten oder beenden. Er kann sogar mit Hilfe von *ptrace(2)* Systemaufrufe mitlesen und verändern, um so geheime Informationen, wie Paßwörter, zu erlangen oder außerhalb der *Chroot-Umgebung* Kommandos ausführen.

Zudem wird oft vergessen, die zusätzlichen GID's zu verwerfen, oder die Reihenfolge der Systemaufrufe ist falsch.

```
[...]
setpwent();
pwd_ptr = getpwnam(<user>);
if(chroot(<jail>) || chdir("/"))
    [EXIT]

if(setgroups(0, 0) < 0) // drop supplementary groups
    [EXIT]

if(setgid(pwd_ptr->gid) || setuid(pwd_ptr->uid)
    [EXIT]
endpwent();
[...]
```

Das *endpwent(3)* ist nötig, um den offenen Filedeskriptor für die Passwortdatenbank wieder zu schließen. (s. nächster Abschnitt) Für das Protokollieren über Syslog muß natürlich vor dem *chroot(2) openlog(3)* aufgerufen werden.

Offene Filedeskriptoren

Wenn der Angreifer es schafft, von dem Serverprozeß eigene Programme abzuspalten (z.B. über Speicherüberläufe), kann er auf offene Filedeskriptoren zugreifen. Er kann Daten außerhalb des *Chroot-Jails* lesen, falls die Filedeskriptoren auf Dateien zeigen, die sich nicht im *Chroot-Verzeichnis* befinden, oder er kann sogar über *fchdir(2)* aus dem *Chroot-Jail* ausbrechen, wenn die Deskriptoren auf Verzeichnisse verweisen (*opendir(3)*).

Sollten die Filedeskriptoren auf Sockets zeigen, dann ist es dem Angreifer möglich, je nach Art des Sockets alle Netzpakete der Collisiondomain, alle Pakete an den Rechner oder alle Pakete für den Server zu lesen.

Mit Hilfe eines Socketdeskriptors ist der Angreifer u.U. sogar in der Lage, den Server zu personifizieren, Sicherheitslücken in den Clients auszunutzen oder wenigstens den Betrieb zu stören.

Über offene Diskriptoren, die auf TTY's zeigen, ist es mit *ioctl(2)* möglich, Befehle in den Eingabepuffer des TTY's schreiben.

Netzwerk API

In der *Chroot-Umgebung* kann ein Angreifer ohne weiteres eigene netzwerkfähige Programme benutzen, um so Restriktionen von Paketfiltern oder *TCP/UDP*-Wrappern zu umgehen, indem er die *Loopback-Schnittstelle* für die Kommunikation benutzt oder einfach aufgrund seiner IP-Adresse weniger restriktiv von den *Access Control Lists (ACLs)* gehandhabt wird. Natürlich kann der Angreifer auch seinen eigenen Server aufsetzen.

Directory Links

Wenn im *Chroot-Verzeichnis* Links existieren, die auf Verzeichnisse außerhalb des "Gefängnisses" verweisen, dann ist ein Ausbruch über *cd(1)* ebenfalls ein Kinderspiel.

Zugriffsrechte

In das *Chroot-Verzeichnis* sollte nur der "eingesperrte" Prozeß schreiben dürfen, da sonst u.U. die Bedingungen (s.o.) von "außen" geändert werden könnten, die das Verlassen des *Chroot-Jails* vereinfachen.

Neben den Zugriffsmöglichkeiten auf Dateien und Verzeichnisse sollten auch die Rechte von *IPC* Strukturen, z.B. *Shared Memory*, so restriktiv wie möglich sein.

/proc Filesystem

Ist das */proc* Dateisystem im *Chroot-Jail* gemounted, dann ist es mit einem einfachen *cd /proc/1/root/* o. ä. möglich, den Rest des Dateisystems zu erreichen.

Die Sicherheit von Chroot-Umgebungen stützt sich also auf die korrekte Programmierung und auf das fehlerfreie Einrichten des Verzeichnisses. Aufbau einer Chroot-Umgebung:

```
[...]
#ifdef OPEN_MAX
    static long maxopenfd = OPEN_MAX;
#else
    static long maxopenfd = 0L;
#endif
[...]


/* drop privileges */
if(setgid(<special GID>) < 0)
    [Exit]
if(setuid(<special UID>) < 0)
    [Exit]

/*
** We set the Close-on-Exec flag on all open filedescriptors.
** If an attacker spawns a new process due to a buffer overflow
** s/he won't be able to access our open file/network
** handles.
*/
if(maxopenfd = 0)
    if((maxopenfd = sysconf(_SC_OPEN_MAX)) < 0)
        maxopenfd = OPEN_MAX_LINUX;

for (i = 3; i <= maxopenfd; i++)
{
    oldval = fcntl(i , F_GETFD, 0L);
    (void) fcntl(i, F_SETFD, (oldval != -1) ? (oldval | FD_CLOEXEC) :
                FD_CLOEXEC);
}
```

```
if(chroot(<special Dir>) < 0)
    [Exit]
if(chdir("/") < 0)
    [Exit]

[...]
```

Die Einrichtung des Verzeichnisses hängt stark von der Applikation ab und wird deswegen hier nicht gezeigt. 

LinuxKP.org 31.05.2001