

Sicherheitsrelevante Programmierfehler (Teil 6)

[Thomas Biege](#)

Inhaltsverzeichnis

- [System Limits \(nicht auf C beschränkt!\)](#)
- [Signale \(nicht auf C beschränkt!\)](#)
- [Interval Timer](#)
- [Terminals und Escape-Sequenzen \(nicht auf C beschränkt!\)](#)
- [Handhabung sensibler Daten \(nicht auf C beschränkt!\)](#)
- [Sonstiges](#)

System Limits (nicht auf C beschränkt!)

Es gibt verschiedene Systemressourcen, die durch den Administrator über *ulimit(1)* oder durch den Programmierer mit Hilfe von *setrlimit(2)* begrenzt werden können. Durch Limitierung, z.B. von Filedeskriptoren oder Memorygrößen, ist es möglich, sogenannte *Denial-of-Service Attacks* zu verhindern. Wenn die Größe von *Core-Files* (Core-Dateien enthalten den Speicherabzug eines Programmes kurz bevor es abgestürzt ist) auf 0 gesetzt wird, kann sogar die ungewollte Preisgabe von Informationen verhindert werden. Als Beispiel ist ein Bug im FTP Daemon zu nennen, der vor ein paar Jahren im Betriebssystem *Solaris* aufgetreten ist. Der FTP Server hat den Inhalt der Datei */etc/shadow*, die ein normaler Benutzer nicht lesen darf, in den Speicher geladen, um den Benutzer zu authentifizieren. Der Angreifer konnte den Prozeß durch Senden von Signalen dazu bringen, abzubrechen und eine *Core-Datei* zu schreiben. Aus der *Core-Datei* war es dem Angreifer möglich, die Daten der geschützten Datei */etc/shadow* zu extrahieren.

Beispiel für das Setzen von Systemlimits:

```
[...]
extern int setlimits(sl_limit slim)
{
    struct rlimit rst;

    rst.rlim_cur = 0;

    rst.rlim_max = slim.fsize;
    if(setrlimit(RLIMIT_FSIZE, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.data;
    if(setrlimit(RLIMIT_DATA, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.stack;
    if(setrlimit(RLIMIT_STACK, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.core;
    if(setrlimit(RLIMIT_CORE, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.rss;
```

```

    if(setrlimit(RLIMIT_RSS, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.nofile;
    if(setrlimit(RLIMIT_NOFILE, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.nproc;
    if(setrlimit(RLIMIT_NPROC, &rst) < 0)
        return(-1);

    rst.rlim_max = slim.memlock;
    if(setrlimit(RLIMIT_MEMLOCK, &rst) < 0)
        return(-1);

    return(0);
}
[...]
```

Als negatives Beispiel für die Verwendung von Systemlimits soll hier eine alte Version von *su(1)* genannt werden. *su(1)* versuchte, die Paßwortdatei zu öffnen. Wenn das nicht ging, wurde ohne Paßwortabfrage eine Kommandoshell mit root-Rechten geöffnet, da *su(1)* annahm, daß das Filesystem kaputt sei. Wenn nun aber ein böartiger Benutzer die maximale Anzahl offener Filedesktoren auf 0 setzt, dann kann er *su(1)* austricksen um root-Zugriff zu bekommen. Bei Fehlern sollten sich sicherheitsrelevante Programme beenden und nicht versuchen, den Fehler zu interpretieren und zu beheben.

Signale (nicht auf C beschränkt!)

Ein Programmierer sollte einiges über dieses Thema wissen, damit seine Programme stabil und sicher laufen und nicht durch das Senden von Signalen gestört werden können.

Folgende Bedingungen müssen erfüllt sein, damit ein Prozeß Signale von einem anderen Prozeß akzeptiert (*BSD Kernel*):
(Sender S, Empfänger E)

- die reale *UID* von S ist die von root
- die reale *UID* von S und E sind gleich
- die effektive *UID* von S und E sind gleich
- die reale *UID* von S ist gleich der effektiven *UID* von E
- die effektive *UID* von S ist gleich der reale *UID* von E
- S und E gehören der selben Session ID an

Die *UID* kann über *setuid(2)* bzw. *seteuid(2)* und die Session ID mit Hilfe von *setsid(2)* geändert werden.

Bevor ein Programm kritische Codeabschnitte bearbeitet, sollte es immer alle Signale blocken. Auch eigene Signale, also nicht die, die von einem Benutzer geschickt wurden, können gefährlich werden.

Beispiel:

```

[...]
```

```

extern int sigprotection(u_int toggle, sigset_t *sp_blockmask)
{
    static sigset_t sp_savedmask;
    static u_int sp_status = SP_OFF;

    switch(toggle)
```

```

{
    case SP_ON:
        if(sp_status != SP_ON)
        {
            if(sp_blockmask == NULL)
                return(-1);

            sp_status = SP_ON;
            if(sigprocmask(SIG_BLOCK, sp_blockmask,
                &sp_savedmask) < 0)
                return(-1);
        }
        break;
    case SP_OFF:
        if(sp_status != SP_OFF)
        {
            sp_status = SP_OFF;
            if(sigprocmask(SIG_SETMASK, &sp_savedmask,
                NULL) < 0)
                return(-1);
        }
        break;
    default:
        return(-1);
}

return(0);
}
[...]
```

Als Beispiel für die Relevanz von Signalen ist hier ein Bug zu nennen, der in dem Programm *ping* aufgetreten ist. *Ping* sendet mit Zeitverzögerung *ICMP* Echo-Request Nachrichten über das Netz, um die Existenz einer anderen Netzwerkkomponente und deren Erreichbarkeit festzustellen. Ein Angreifer konnte nun in kurzen Abständen das Signal *SIGALRM* senden und dadurch die Zeitverzögerung nahezu aufheben, um so das Netz mit *ICMP*-Paketen zu überfluten.

Interval Timer

Es gibt drei Interval-Timer:

<code>ITIMER_REAL</code>	zählt die tatsächlich laufende Zeit und liefert <i>SIGALRM</i> wenn die Null erreicht wurde.
<code>ITIMER_VIRTUAL</code>	zählt nur, wenn der Prozeß ausgeführt wird und liefert <i>SIGVTALRM</i> , wenn die Null erreicht wurde.
<code>ITIMER_PROF</code>	zählt sowohl, wenn der Prozeß Rechnerzeit beansprucht, als auch, wenn das System für den Prozeß tätig ist. Zusammen mit <i>ITIMER_VIRTUAL</i> , kann man so ermitteln, wie lange eine Applikation sich im User- und im Kernelprogrammraum aufhält. Wenn der Zähler abgelaufen ist, wird <i>SIGPROF</i> geliefert.

Da diese Timer an den *Child-Prozeß* vererbt werden, sollten sicherheitskritische Programme diese Timer ignorieren, damit ihr Ablauf nicht durch ungewollte Signale gestört wird.

Die Interval-Timer können über *setitimer(2)* und *getitimer(2)* manipuliert und abgefragt werden.

Terminals und Escape-Sequenzen (nicht auf C beschränkt!)

Terminals, also nicht nur die alten *Video Terminals* (VT's), die über serielle Schnittstellen an *Mainframes* angeschlossen sind, sondern auch die entsprechenden Emulatoren, die zur Betreuung vom Kommandoshells nötig sind, können nicht nur Zeichen darstellen, sondern bieten auch die Möglichkeit, über *Escape-Sequenzen* den Cursor zu positionieren, Vorder- und Hintergrundfarbe zu verändern, Tasten neu zu belegen oder sogar Kommandos auszuführen.

Diese Eigenschaften können von einem Angreifer dazu benutzt werden, um das Terminal unbrauchbar zu machen, Informationen zu verfälschen oder sogar Befehle ausführen zu lassen.

Um diese Gefahr zu unterbinden, sollten Programme, die auf das Terminal eines Benutzers schreiben, z.B. Mail-Clients, Chat-Programme, Web-Browser etc, das *Escape-Zeichen* ausfiltern (auf *vtXXX Terminals* ist das: 0x00 bis 0x1F und 0x7F bis 0x9F), oder am besten nur Buchstaben, Zahlen und Interpunktionszeichen passieren lassen.

Handhabung sensibler Daten (nicht auf C beschränkt!)

Sensible Daten, wie Paßwörter, Personaldaten etc., sollten am besten sofort nach Verwendung wieder aus dem RAM gelöscht werden und im Idealfall von einem extra Prozeß bearbeitet werden, der klein - also gut zu auditieren - ist, nur über sichere *IPC* (z.B. *Pipes*) mit dem Anwenderprogramm kommuniziert und völlig autark (eigene *UID*, *GID*, *Session ID*) im Prozeßuniversum existiert. In einigen extremen Fällen ist der Einsatz von Kryptographie nicht verkehrt.


Häufig werden für sensible Daten gepufferte I/O-Funktionen verwendet. Das kann vom Anwendungsprogrammierer gewollt sein oder ist Teil einer Bibliothek, auf die der Programmierer keinen Einfluß hat. In solchen Fällen befinden sich die sensiblen Informationen in Speicherbereichen, die nicht einfach gelöscht werden können. Hier sollte man den Puffer für den Stream mit *setbuf(3)* löschen, indem man *setbuf(fstream, NULL)* aufruft.

Sollen Daten auf der Festplatte (oder RAM) sicher überschrieben werden, also so, daß sie selbst durch *Magnetic Force Microscopy* nicht wieder restauriert werden können, empfiehlt es sich, das Paper von *Peter Gutmann* zu lesen. (Peter Gutmann; Secure Deletion of Data from Magnetic and Solid-State Memory; USENIX 6) Nach jedem Schreiben sollte man jedoch beachten *fflush(3)* und anschließend *sync(2)* aufzurufen, um die Daten physikalisch auf das Medium zu schreiben. Von der Verwendung von *fsync(3)* sollte abgesehen werden, da es im Gegensatz zu *sync(2)* nicht blockiert und somit das physikalische Schreiben der speziellen Bitmuster möglicherweise inkonsistent wird.

Sonstiges

Im Grunde gibt es nicht viel, was man beim Linken falsch machen kann. Man sollte jedoch immer darauf achten, daß die Pfadangaben für die *Shared Libraries* absolut sind. **Relative Pfade** zu *Shared Libraries* können ausgenutzt werden, um eigene Libraries zur Laufzeit zu laden. Privilegierte Prozesse

würden somit die Sicherheit verletzen, in dem sie Funktionen, die vom Angreifer verfaßt wurden, ausführen.

Aufrufe, die die Kommandoshell benutzen, wie *system(3)* oder *popen(3)* (und wie bereits erwähnt, *getcwd(3)* unter alten SunOS) sollten unbedingt in privilegierten Applikationen und Netzwerkdiensten vermieden werden. Selbst wenn die Benutzereingaben gefiltert wurden, kann es immer noch unangenehme Interaktionen mit der Shell geben, zum Beispiel mit den Umgebungsvariablen ENV und BASH_ENV (s. *bash(1)*). 

LinuxKP.org 11.06.2001