

Sicherheitsrelevante Programmierfehler (Teil 8)

[Thomas Biege](#)

Inhaltsverzeichnis

- [Perl](#)
 - [Temporäre Dateien](#)
 - [Gefährliche Befehle und Eigenschaften von Perl](#)
 - [Signale](#)
- [Tools](#)
- [Quellen](#)

Perl

Perl wird i.d.R. für kleine bis mittlere Aufgaben benutzt. Der große Vorteil von Perl ist die Verarbeitung von Zeichen mit Hilfe von regulären Ausdrücken. Der Aufgabenbereich von Perlskripten ist also ähnlich dem von Shellskripten; im CGI Bereich werden sie sogar wesentlich häufiger als Shellskripte eingesetzt. Zusätzlich können viele Aufgaben, wie Netzwerkkommunikation (Client/Server), mit Hilfe von Modulen ebenfalls von Perl erledigt werden. Perl ist also aufgrund seiner Vielseitigkeit einem breiten Spektrum an Gefahrenquellen ausgeliefert.

Temporäre Dateien

Perl bietet keine dedizierte Möglichkeit, temporäre Dateien anzulegen. Leider wird dafür häufig der `open()` Befehl benutzt, der ein Perl Skript angreifbar macht (*Link Attacken, Race Conditions*).

Eine sichere Möglichkeit bietet die Verwendung von eigenen Verzeichnissen, wie im Shellskript-Abschnitt beschrieben, oder die Funktion `sysopen()`. `sysopen()` wird genauso wie der Systemaufruf `open(2)` benutzt.

Beispiel:

```
[...]  
use Fcntl qw(O_RDWR O_CREAT O_EXCL);  
[...]  
  
sysopen(TMPFILE, "/tmp/myfile.666",  
        O_RDWR|O_CREAT|O_EXCL, 0600);  
[...]
```

Das Problem und dessen Lösung entsprechen also genau dem von C.

Gefährliche Befehle und Eigenschaften von Perl

Viele Perl Befehle haben unangenehme Eigenschaften, die es einem Angreifer leicht machen können, Sicherheitslücken zu finden.

Nachfolgend eine Auflistung aller sicherheitsrelevanten Befehle und Eigenschaften.

Grundsätzlich sollte man versuchen, zur Erfüllung seiner Aufgaben die Befehle zu benutzen, die ihrerseits nicht auf die Shell zurückgreifen. Wenn nicht vertrauenswürdige Daten auf die Shellebene gelangen, kann für nichts garantiert werden.

system() und **exec()** verhalten sich recht ähnlich und benutzen zudem beide die Shell, wenn sie mit nur einem Argument aufgerufen werden. Kann ein Angreifer dieses Argument definieren, dann ist es ihm möglich, über Shell-Metazeichen ein beliebiges Kommando zu starten.

Fehler:

```
system("ls -al /home/$username");
```

Gibt der Angreifer als \$username *"evil;cat /etc/shadow"* an, dann wird die Datei */etc/shadow* ausgegeben.

Korrektur:

```
system("ls", "-al", "/home/$username");
```

Leider ist es dem Angreifer immer noch möglich, über *../../../../../../* eine Datei seiner Wahl benutzen zu lassen.

Am besten benutzt man für Perl, wie für Shell, einen Filter, der nur erlaubte Zeichen durchläßt.

```
unless($userinput =~ tr/[a-zA-z0-9@]//)
{
    print "Nice try, pal!\n";
    exit(1);
}
```

Die Funktionen **glob()**, **<>** und **Backticks** benutzen ebenfalls die Shell und sollten nicht direkt mit Benutzerdaten versorgt werden.

Perl's **open()** Befehl kann mit Hilfe des Pipesymbols Programme wie Dateien benutzen.

Beispiel:

```
[...]
open(LPD, "| lpd");
[...]
```

In diesem Beispiel wird der BSD Druckerspooler-Client zum Schreiben geöffnet, um eine Datei zu drucken.

Diese Eigenschaft von **open()** hat zur Folge, daß ein Angreifer einfach ein Pipesymbol mit anschließenden Programmnamen als Argument für **open()** angeben muß, um dieses Programm zur Ausführung zu bringen.

Wir können mit folgendem Code ein sicheres **popen(3)** emulieren:

```
[...]
```

```
open(PIPE, "-|") || exec("/bin/ls", $userdate);
print while <PIPE>;
[...]
```

Hier besteht immer noch die Gefahr, daß der Benutzer über `../` im Verzeichnisbaum zurückgeht.

Um Perl Befehle dynamisch während der Laufzeit eines Perl Skriptes auszuführen, kann der Befehl **`eval()`** oder der Modifier `/e` für reguläre Ausdrücke verwendet werden, der einen Ausdruck verarbeitet, bevor er ausgewertet wird. Werden also Benutzereingaben an **`eval()`** oder `/e` weitergegeben, so kann ein Angreifer Perl Code nach seiner Façon ausführen lassen. Hier hilft auch nicht unbedingt ein Zeichenfilter.

Da Perl häufig externe Programme über die Shell aufruft, besteht die Gefahr eines Angriffs über die **Umgebungsvariablen** `$PATH`, `$IFS` etc. Sollte das Perlskript mit erhöhten Rechten laufen, so empfiehlt es sich, die Programmumgebung zu säubern und komplett neu zu setzen bzw. sollten die vollen Pfadnamen benutzt und die Shell vermieden werden.

Beispiel:

```
[...]
$ENV{PATH} = join ':' => split(" ", << ' __PATHEND__ ');
    /bin/
    /sbin/
    /usr/bin/
    /usr/sbin/
__PATHEND__
[...]
```

Eine Besonderheit spielt noch die `@INC` Variable. Diese Variable gibt den Pfad zu den Perl-Modulen an. Setzt der Angreifer sie auf einem Pfad, der seine Module enthält, dann kann er seinen Code ausführen lassen. Bevor also Module über `use` eingebunden werden, sollte die `@INC` Variable auf einen sicheren Wert gesetzt werden.

Das sog. **Poison NUL Byte** (der Begriff wird auch für One-Byte-Bufferoverflows verwendet) beschreibt eine Eigenschaft von Perl im Zusammenhang mit C. Perl betrachtet das Zeichen `0x00` nicht als Ende einer Zeichenkette, C hingegen schon. Diese Eigenschaft kann sich ein Angreifer zunutze machen, indem er an geeigneter Stelle in seiner Eingabe ein `0x00` Zeichen benutzt.

Beispiel:

```
[...]
$username = param("username");
[...]
```

```
open(PROFILE, "-|") || exec("/usr/local/bin/my_txt2html",
                           "/etc/$username.profile");

print while <PROFILE>;
[...]
```

Dieses CGI Skript ermöglicht es, Benutzerprofile abzurufen, die in `/etc` liegen (sehr unwahrscheinlich). Gibt der Angreifer nun `profile.pl&username=shadow%00` an, dann öffnet `my_txt2html` nicht `/etc/shadow0x00.profile`, sondern `/etc/shadow`. Hier hilft auch wieder nur ein Filter für erlaubte Zeichen.

Es ist wichtig, daß Backslashes in der Benutzereingabe entfernt werden. Backslashes in der Benutzereingabe neutralisieren nachfolgende Backslashes, die beispielsweise von dem eigenen Perlskript eingefügt wurden, um gefährliche Eingaben zu entschärfen. Ähnliche Gefahren durch andere Zeichen sind ebenfalls denkbar.

Um es nochmal deutlich zu machen: Es sollten nicht die vermeintlich gefährlichen Zeichen ausgefiltert, sondern einfach nur die erlaubten durchgelassen werden.

Falls ein Perl Skript wider alle Vernunft **SetUID** oder **SetGID** gesetzt sein sollte, dürfen die besonderen Privilegien nur dann aktiviert werden, wenn sie wirklich gebraucht werden, und diese Abschnitte sollten so klein wie möglich sein.

Beispiel:

```
[...]
# drop privileges
$) = $( # eGID = rGID
$> = $< # eUID = rUID
[...]

# gain back higher privileges
$( = $) # rGID = eGID
$< = $> # rUID = eUID
[...]
```

Perl Skripte mit besonderen Aufgaben und Privilegien sollten im **Taint-Mode** ausgeführt werden. Dazu muß *perl* lediglich die Option *-T* benutzen. Natürlich kann der Taint-Mode nur einen geringen Teil der Sicherheitsrisiken von Perl verhindern; er kann dem Programmierer nicht das Denken abnehmen.

Signale

Perl ist bei Signalen denselben Gefahren ausgeliefert wie C und Shell-Sprachen.

Um Signale zu blocken oder abzufangen, kann man wie folgt vorgehen:

```
[...]
sub SayGoodBye
{
    print "Good Bye!\n\n";
    exit(0);
}
[...]

$SIG{'HUP'} = 'SayGoodBye';
$SIG{'INT'} = 'SayGoodBye';
$SIG{'QUIT'} = 'SayGoodBye';
$SIG{'ILL'} = 'SayGoodBye';
$SIG{'TRAP'} = 'SayGoodBye';
$SIG{'ABRT'} = 'SayGoodBye';
$SIG{'UNUSED'} = 'SayGoodBye';
$SIG{'FPE'} = 'SayGoodBye';
$SIG{'KILL'} = 'SayGoodBye';
$SIG{'USR1'} = 'SayGoodBye';
$SIG{'SEGV'} = 'SayGoodBye';
$SIG{'USR2'} = 'SayGoodBye';
$SIG{'PIPE'} = 'SayGoodBye';
$SIG{'ALRM'} = 'SayGoodBye';
$SIG{'TERM'} = 'SayGoodBye';
$SIG{'STKFLT'} = 'SayGoodBye';
$SIG{'IO'} = 'SayGoodBye';
$SIG{'XCPU'} = 'SayGoodBye';
$SIG{'XFSZ'} = 'SayGoodBye';
$SIG{'VTALRM'} = 'SayGoodBye';
$SIG{'PROF'} = 'SayGoodBye';
[...]
```

Tools

- Brain + Experience ;-))
<http://127.0.0.1/schweinehundueberwinden/weg/zum/erfolg.html>
- SecProgLib
<http://www.suse.de/~thomas>
- ITS4
<http://www.rstcorp.com/its4>
- PSCAN
<http://www.striker.ottawa.on.ca/~aland/pscan/>
- frc-scanner
<http://www.notatla.Daemon.co.uk/SOFTWARE>
- ssc
<http://packetstorm.securify.com/UNIX/misc/sscc.tar.gz>
- initd_
http://www.securityfocus.com/data/tools/initd_.tar.gz
- LibSafe
<http://www.bell-labs.com/org/11356/libsafe.html>
- StackGuard
- StackShield
- Openwall Patches
<http://www.openwall.com>
- lclint

Quellen

- Bugtraq@securityfocus.com
- Secprog@securityfocus.com
- Security-Audit@*ferret*uk
- Aleph One; Smashing the Stack for Fun and Profit; Pharck49-7
- W. Richard Stevens; Advanced Programming in the UNIX® Environment; Addison Wesley
- W. Richard Stevens; UNIX® Network Programming; Prentice Hall
- Trutz Eyke Podschun; Das Assembler-Buch; Addison Wesley
- Matt Bishop; Unix Security: Writing Secure Programs; SANS '96
- Gerhard Willms; Das C-Grundlagen Buch; Data Becker
- perlsec(1) Man Page 