öS 62,– sfr 7,50 **DM 7,50** H 10554 Februar 1998

2

iX

MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

Mail-Viren-Check mit Linux

Anschluß für die Firma:
Internet Provider

Getestet:

AS/400 als Web-Server

IBMs Midrange-System im Internet

Für Mac, OS/2, Unix, Windows:
Netzweite Fax-Server

Für AppleTalk, IPX und SMB:
TotalNET Advanced Server

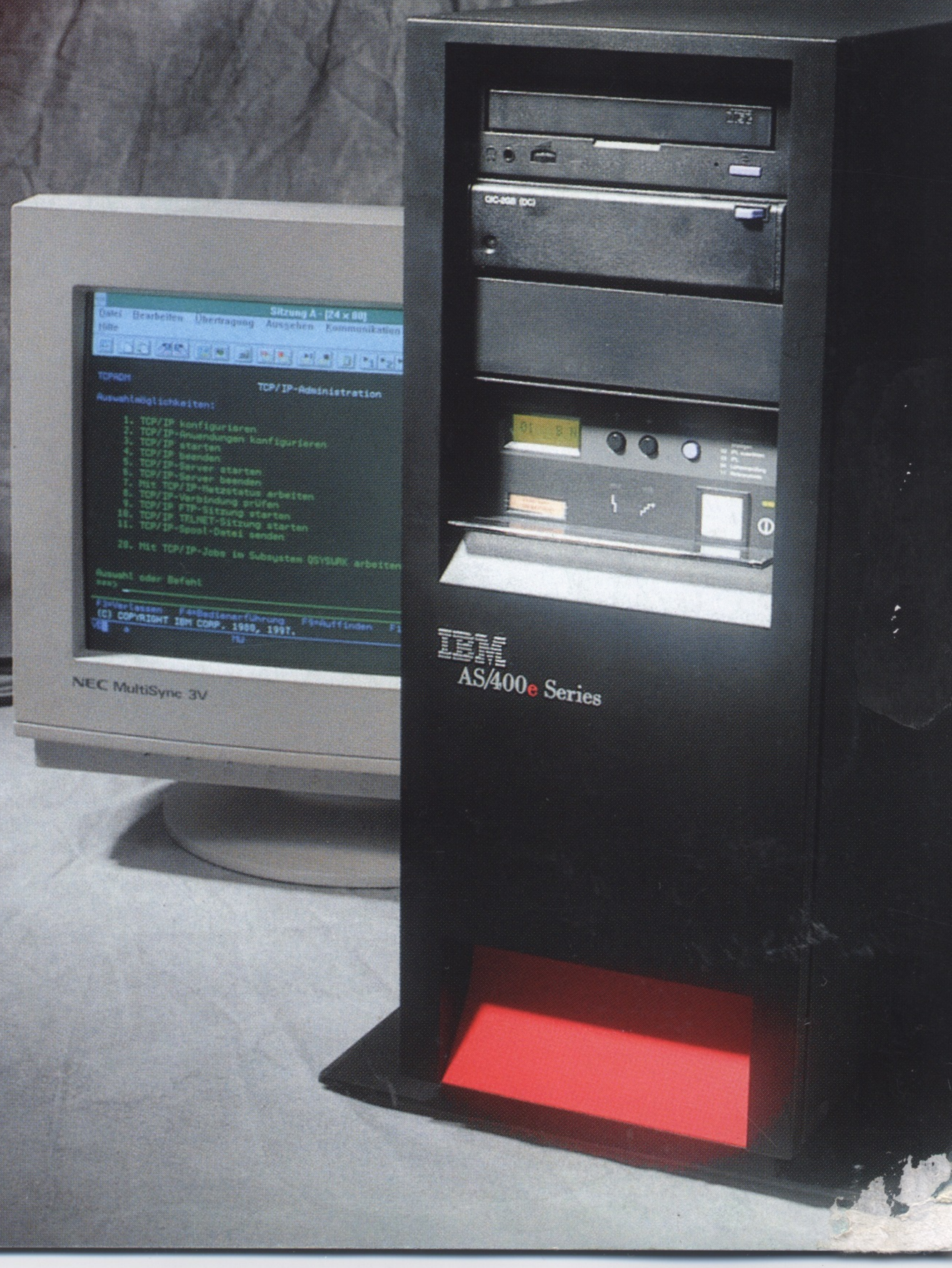
Apple in der Intel-Welt:
Rhapsody für PCs

Automatisch übersetzen:
EMail-Translator

Programmieren:
Linux Device Driver Kit
CORBA-Java-Anbindung
Perl remote einsetzen
Tcl-GUI-Builder

Netze:
Gigabit Ethernet
SNMP Version 3
NT-Login im Detail

Mainframe-Ersatz:
SPARC-Server E10000



4 399148 507509 02

Perl-Programme remote einsetzen

Schnatter, schnatter

Michael Schilli

Java hat es vorgemacht – doch Code auf fremde Maschinen zu laden, um ihn dort auszuführen, ist seit neuestem auch mit Perl möglich. Die Penguin-Release 3.0 von Felix Gallo bietet ein skalierbares Sicherheitsmodell, das auf PGP basiert.

Perl-Programme müssen nicht immer auf dem Rechner laufen, auf dem die Scripts liegen. Penguin stellt eine Bibliothek von Funktionen zur Verfügung, die Code- und Datenstücke zwischen Rechnern mit aktiviertem Perl-Interpreter im Netzwerk übermitteln und eintreffenden Perl-Code in sicheren Umgebungen ausführen.

Entwickler Felix Gallo stellt sich die ideale Anwendung seines frei verfügbaren Produkts als ein Heer im Internet hin und her wuselnder Informationsagenten vor – einer Schar schnatternder Pinguine nicht unähnlich.

iX-TRACT

- Das Penguin-Modul enthält Funktionen zum Übertragen und Ausführen von Perl-Programmen auf entfernten Rechnern.
- Im einfachsten Fall schickt es ein Script zum entfernten Rechner, der seine Authentizität mit Hilfe von PGP überprüft. Für jeden Benutzer läßt sich einstellen, welche Funktionen seine Scripts ausführen dürfen.
- Umgekehrt kann ein Rechner Perl-Programme von einem anderen laden und lokal in einem 'Sandkasten' ausführen. PGP sorgt auch dabei für die nötige Sicherheit.

Trau, schau, wem!

Führte Penguin jedoch einfach wahllos eintrudelnde Perl-Scripts aus, geriete das System schnell in Schwierigkeiten, denn wer will schon wildfremden Leuten erlauben, Dateien zu durchstöbern oder die Festplatte zu formatieren? Penguin stellt über das *Safe*-Modul von Malcolm Beattie und Tim Bunce sogenannte Compartments zur Verfügung, in denen Programme nur in einem fest umrissenen 'Sandkasten' herumtoben dürfen – den geringsten Versuch, diesen zu verlassen, ahndet der Interpreter mit sofortigem Abbruch des betreffenden Scripts. Dabei ist bestimmten Benutzer mehr erlaubt als anderen. Als Ausweis gilt die digitale PGP-Unterschrift (siehe Kasten: 'Tür auf oder Riegel vor?'), mit der der Absender empfängerseitig auszuführenden Code signiert.

Im Gegensatz zu Java, das in einem starren 'Sandkasten' läuft, bietet der Perl-Penguin über das *Safe*-Modul eine fein abgestufte Rechteverwaltung. So, wie man im richtigen Leben manchen Leuten den Hausschlüssel anvertraut und mit anderen lieber nur schriftlich verkehrt, läßt Penguin unterschiedliche Vertrauensstufen zu. Höchst zweifelhafte Naturen dürfen beispielsweise nur festgelegte Funktionen aufrufen, ohne auch nur in eine Schleife einzutreten – denn schließlich könnten sie so CPU-Zeit verbraten. Vertrauenswürdigeren Personen darf man auf der anderen Seite für den gesamten Perl-Befehlsumfang Tür und Tor öffnen.

Bevor ein Perl-Interpreter ein Script ausführt, wandelt er es in eine Reihe von Opcodes um, den internen Perl-Bytecode. Das *Safe*-Modul erlaubt es nun zusätzlich, eine Operatormaske anzugeben, das heißt Operatoren zuzulassen oder auszuschließen. Das von *Safe* intern verwendete Opcode-Paket legt für jeden der über 300 Perl-Opcodes einen Namen fest, Beispiele sind der +-Operator (*add*), der Eintritt in eine Schleife (*enterloop*) oder Systemfunktionen wie *unlink*. Die Abstufung kann soweit gehen, jedem Penguin-Benutzer eine Auswahl an Perl-Operatoren zuzuordnen, doch in der Praxis bedient man sich von Opcode bereitgestellter Tags, die unter einem symbolischen Namen sinnvolle Gruppen von Perl-Opcodes zusammenfassen (siehe 'Wichtige Opcode-Gruppen-Tags'). Die *Opcode.pm*

beiliegende Manual-Seite (kommt mittels *perldoc Opcode* zum Vorschein) gibt detailliertere Informationen dazu.

Die Datei *.rightsfile* im Heimatverzeichnis des Empfängers ordnet den einzelnen Benutzern, die sich über PGP-Signaturen eingesandter Perl-Scripts ausweisen, bestimmte Rechte zu:

```
#Der Gute
[Michael Schilli <mschilli1@aol.com>]
:base_core :base_loop print
# Der Böse
[Boeser Willi <bgates@microsoft.com>]
:base_core
# Nicht ausgewiesener Benutzer
[default]
:default
```

Penguin implementiert die PGP-Funktionen keineswegs selbst. Es bedient sich Phil Zimmermanns Programm *pgp*, das einschließlich eines Keyring (einer Sammlung von Public Keys bekannter Benutzer) auf Sender- und Empfängersystem vorhanden sein muß (siehe Kasten 'Installation').

Pinguine im Einsatz

Um zwei Anwendungsfälle geht es diesmal:

- ein lauschender Server nimmt Perl-Code von Clients entgegen, um ihn auszuführen,
- ein Client meldet sich beim Server, holt ein Programm ab und führt es aus.

Für den ersten Fall enthält die Penguin-Distribution bereits zwei fertige Module, *Penguin::Trivial::Server* und *Penguin::Trivial::Client*. Listing 1 zeigt den Client *uclient.pl*, der eine Schwierigkeit offenbart: Zum Codesignieren muß er sich gegenüber dem *pgp*-Programm identifizieren – schließlich braucht er Informationen aus dem geschützten privaten Keyring. Das Kennwort im Klartext in ein Perl-Script zu verpacken, ist schlechter Stil, da so der PGP-Keyring samt privatem Schlüssel offenliegt. Sicherer, wenngleich umständlicher, fragt man das Paßwort bei Scriptstart am Terminal blind ab (Zeilen 10 bis 14).

Die Zeilen 16 bis 18 lesen die Datei *codefile* ein, die den auf der Zielmaschine auszuführenden Perl-Code enthält. In Zeile 20 setzt der Konstruktoraufruf alle notwendigen Variablen für das Client-Objekt. Die *run*-Methode in Zeile 27 überträgt das verpackte und signierte Codepaket an den Server, der es wiederum auspackt, überprüft, ausführt und den Rückgabewert des Codestücks an den Client schickt.

Listing 2 zeigt den zugehörigen Server *userver.pl*. Er benötigt kein Paßwort, da er nur die digitale Signatur eines ankommenden Codestückchens verifizieren muß und der private Schlüssel verfahrensbedingt hierbei keine Rolle spielt. Er muß mit *userver.pl&* gestartet worden sein, bevor man *uclient.pl* ausführen lassen kann.

Zeile 3 bindet das *Penguin::Trivial::Server*-Modul ein, dessen Konstruktoraufruf in den Zeilen 7 bis 9 steht. Neben der Nummer des verwendeten Ports erhält er mit dem *Share*-Parameter die Referenz einer Liste erlaubter Serverfunktionen. Diese dienen dazu, den strikten Rechte-Mechanismus des Penguin-Moduls serverseitig zu umgehen: Die ankommenden Perl-Snippets dürfen die explizit angegebenen Funktionen unabhängig von den in der Rechtedatei zugelassenen Opcodes benutzen.

Die *serve*-Methode in Zeile 13 lauscht am definierten Port und handelt alles Notwendige ab: Sie nimmt an-

WICHTIGE OPCODE-GRUPPEN-TAGS

base_core	Grundlegendes, Stringmanipulation, Listen, Subroutinen et cetera
base_mem	Anonyme Hashes/Listen (Speicherattacken möglich)
base_loop	Schleifen (CPU-Attacken durch Endlosschleifen möglich)
base_io	Ein-/Ausgabe über vordefinierte File-Handles
default	:base_core :base_mem :base_loop :base_io :base_orig
filesys_read	Lesender Zugriff aufs Dateisystem
sys_db	Systemressourcen wie Paßwortdatei, DNS et cetera
browse	:default :filesys_read :sys_db
filesys_write	Schreibender Zugriff aufs Dateisystem
subprocess	Shell- und Prozeßzugriff (system(), fork(), et cetera)

Tür auf oder Riegel vor?

Wenn ein Penguin-Partner A ein Script an B schickt, sollte B folgendes sicherstellen, bevor das Programm irgendwelche Sonderrechte erhält:

- Der Absender ist tatsächlich A. Er muß hierzu B gegenüber seine Identität beweisen.
- Das Script liegt B genau so vor, wie A es abgeschickt hat.

Penguin prüft dies mit Phil Zimmermanns frei verfügbaren Programm PGP. Es arbeitet nach dem Public-Key-Verfahren mit zwei Schlüsseln: Dem privaten (Private Key), der geheim bleibt, und einem öffentlichen (Public Key).

Eine mit dem öffentlichen Schlüssel kodierte Nachricht knackt nur der private Schlüssel. Möchte A eine verschlüsselte Nachricht an B senden, benutzt er deshalb Bs öffentlichen Schlüssel. Lesen kann diesen Datensalat anschließend nur der Besitzer von Bs privatem Schlüssel –im Normalfall B selbst.

Umgedreht ergibt sich eine interessante Anwendung: Eine mit dem privaten Schlüssel geschützte Nachricht läßt sich mit dem öffentlichen lesbar machen. So kann man einen Text dekodieren, den nur einer so verschlüsseln konnte, wie er verschlüsselt ist: Der Besitzer des Private Key.

Zusammen mit einer 'Message-Digest'-Funktion läßt sich so ein digitales Unterschriftsverfahren realisieren. Solche Funktionen erzeugen aus einem beliebig langen Dokument einen Stempel fester Länge. Dabei kommen für verschiedene Dokumente unterschiedliche Stempel heraus. Wenn sich in einem Text nur ein einziger Buchstabe ändert, ist der Stempel danach nicht wiederzuerkennen. Zum Beispiel liefert das MD5-Verfahren:

```
"Betrag: DM 1000.00" MD5->
  fed3722604fdcead9f77bdb9de67ccd7
"Betrag: DM 1001.00" MD5->
  41acb1c152699d2064d334185d864551
"Betrag: DM 1000.01" MD5->
  6ed217eddb63de76adf6b447516fb3cd
```

Eine digitale Unterschrift ist nun nichts anderes als der Message-Digest-Stempel, verschlüsselt mit dem privaten Schlüssel des Absenders. Jeder, der im Besitz des öffentlichen Schlüssels ist, kann den Stempel entschlüsseln und das darüberstehende Dokument selbst dem MD5-Algorithmus unterwerfen, um dann beide Werte zu vergleichen. Sind sie identisch, wurde das Dokument offensichtlich vom Besitzer des privaten Schlüssels so erstellt, falls nicht, hat sich jemand am Dokument oder dem Stempel zu schaffen gemacht.

kommende Perl-Snippets entgegen, führt sie aus und liefert für jede Anfrage ein Datenpaket mit deren Rückgabewert als Inhalt zurück.

Der eine darf, der andere nicht ...

Enthält die Datei *codefile* auf Client-Seite etwa die Zeilen

```
#!/usr/bin/perl

print 'Der Server arbeitet ...\n';
return 'Rückgabewert';
```

fördert der Aufruf von *uclient.pl* Unterschiedliches zutage: Falls für den in Listing 1 in Zeile 6 festgeschriebenen Benutzer in der Serverrechtedatei der *print*-Opcode (respektive das Sammel-Tag *:base_io*) zugelassen ist, lautet das Ergebnis

```
Penguin antwortet: Rückgabwert
```

und der Server gibt

```
Der Server arbeitet ...
```

aus. Läßt das serverseitige *.rightsfile* hingegen kein *print*-Kommando zu, überschreitet der Client mit dem übermittelten Snippet seine Kompetenzen und *uclient.pl* meldet

```
Penguin antwortet: print trapped \
by operation mask at (eval 7) line 3.
```

Anders liegt der Fall jedoch, wenn der Client explizit freigegebene Funktionen des Servers nutzt. Steht also in der Datei *codefile*

```
#!/usr/bin/perl
server_function();
```

gibt es keinerlei Schwierigkeiten, obwohl *server_function()* in Listing 2 letztendlich auf eine *print*-Funktion zurückgreift. Es zählt nur, daß der Server *server_function()* in Listing 2 explizit freigegeben hat, und das Ergebnis ist entsprechend

```
Penguin antwortet: Erlaubt!
```

Java im Browser, Perl im Interpreter

Umgekehrt arbeiten die Listings 3 und 4, *dclient.pl* und *dserver.pl*. Clients laden bei dieser Anwendung Perl-Code vom Server und führen ihn aus – ganz wie ein Java-fähiger Browser Applets von einem Web-Server holt und startet.

Server und Client bedienen sich einer Reihe von Penguin-Funktionen, die die bereits vorgestellten Trivial-Pakete intern schon benutzen. Hier zeigt sich das Penguin-Design im Detail: Kanäle zwischen Rechnern transportieren Frames, die mittels 'Wrappern' eingepackte und eventuell signierte Code- und Datenpakete enthalten. Die Zeilen 3 bis 9 in *dclient.pl* ziehen die notwendigen Penguin-Module, *Penguin::Rights* für die Rechteverwaltung, *Penguin::Frame::Code* für übermittelte Codepakete. *Wrapper::PGP* handelt PGP-signierte Frames ab, *Channel::TCP::Client* bietet eine simple TCP-Verbindung zum Server, *Penguin::Compartment* implementiert Quarantänebereiche für auszuführende Perl-Scripts.

Zeilen 17 bis 21 initialisieren und öffnen den Kanal zum Server, Zeile 23 holt den (erwarteten) Code-Frame, den

Penguin installieren

Penguin ist frei auf dem CPAN erhältlich, in Deutschland unter anderem über

ftp://ftp.archive.de.uu.net/pub/programming/perl/CPAN/

ftp://ftp.gmd.de/packages/CPAN/

ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/

Die aktuelle Version 3.0 liegt relativ dazu in *modules/by-module/Penguin/Penguin-3.00.tar.gz*.

Installiert wird nach dem Auspacken wie üblich mit

```
perl Makefile.PL
make install
```

Empfohlen sei der aktuellste Stand der Perl-Distribution (5.004_04), dieser enthält bereits die von Penguin benötigen Module Safe und IO.

Zum digitalen Signieren benutzt Penguin die Kommandozeilenversion von PGP. Für den nicht-kommerziellen Gebrauch gibt's diese gebührenfrei zum Beispiel unter

ftp://ftp.iks-jena.de/pub/mitarb/lutz/crypt/software/pgp/pgp263ii.tar.gz

Installiert wird folgendermaßen:

```
mkdir pgpinstall
cd pgpinstall
gzip -dc pgp263ii.tar.gz | tar xfv -
cd src
```

Das anschließende *make* benötigt noch ein Kürzel des verwendeten Betriebssystems, eine Linux-Version von *pgp* beispielsweise erstellt

```
make linux
```

Vor dem Einstieg in die Welt der digitalen Signaturen benötigen die Teilnehmer eine Kombination aus privatem und öffentlichem Schlüssel.

```
pgp -kg
```

führt durch ein Fragenmenü, an dessen Ende schließlich beide Schlüssel in $HOME/.pgp liegen. Dieses Verzeichnis muß bereits existieren, sonst steigt das Programm mit einer Fehlermeldung aus.

Damit das empfängerseitige *pgp*-Programm um den Absender weiß, muß dort dessen öffentlicher Schlüssel bekannt sein. Den Schlüssel des Benutzers Meyer gibt *pgp* mit

```
pgp -kxa Meyer file
```

in die Datei *file* aus. Nach deren Übertragung auf das Zielsystem bringt

```
pgp -ka file
```

dem Empfänger den öffentlichen Schlüssel des Senders bei. Anschließend muß *pgp* noch in ein allgemein bekanntes Verzeichnis wie */usr/local/bin*. Und der Reigen der Pinguine kann beginnen.

```
 1 #!/usr/bin/perl
 2
 3 use Penguin::Trivial::Client;
 4
 5 $codefile  = 'codefile';
 6 $name      = 'Michael Schilli <mschilli1\@aol.com>';
 7 $host      = 'localhost';
 8 $port      = 8118;
 9
10 $|=1;
11 print('Password: ');
12 system('stty -echo');
13 chop($password = <STDIN>);
14 system('stty echo');
15
16 open(CODEFILE, '<$codefile') || die 'Cannot open $codefile';
17 while(<CODEFILE>) { $codestring .= $_; }
18 close(CODEFILE);
19
20 $penguinclient = new Penguin::Trivial::Client
21                          Code     => $codestring,
22                          Host     => $host,
23                          Port     => $port,
24                          Password => $password,
25                          Name     => $name;
26
27 print 'Penguin antwortet: ', $penguinclient->run(), '\n';
```

Penguin-Client, der Codestücke an einen Server zur Ausführung sendet (Listing 1).

```
 1 #!/usr/bin/perl
 2
 3 use Penguin::Trivial::Server;
 4
 5 $port = 8118;                    # Portnummer
 6
 7 my $penguinserver = new Penguin::Trivial::Server
 8                          Port     => $port,
 9                          Share    => [ 'server_function' ];
10
11 print 'Penguin Server Up [$port]\n';
12
13 $penguinserver->serve();         # Lauschen …
14
15 ############################################################
16 sub server_function {
17 ############################################################
18     print 'Erlaubt!\n';
19     return 'Erlaubt!';
20 }
```

Lauschender Penguin-Server: Er wartet auf eintreffende Perl-Snippets, entpackt und führt sie aus (Listing 2).

```
 1 #!/usr/bin/perl
 2
 3 use Penguin;
 4 use Penguin::Rights;                 # Zugriff auf Rechtedatei
 5 use Penguin::Frame::Code;           # Code-Frames
 6
 7 use Penguin::Wrapper::PGP;          # PGP-Signierung
 8 use Penguin::Channel::TCP::Client;  # Kanal
 9 use Penguin::Compartment;           # 'Sandkasten'
10
11 $targethost = 'localhost';
12 $targetport = '8118';
13
14 $rightsdb = new Penguin::Rights;    # Rechtedatei einlesen
15 $rightsdb->get();
16
17 $mychannel = new Penguin::Channel::TCP::Client Peer => $targethost,
18                                                Port => $targetport;
19
20                                     # Verbindung aufnehmen
21 $mychannel->open or die 'No connection to Penguin';
22
23 $frame = $mychannel->getframe();   # Frame holen
24
25 die 'Invalid frame!' if $frame eq undef;
26
27                                     # Auspacken
28 ($title, $signer, $wrapmethod, $code) = $frame->disassemble();
29
30                                     # User-spezifische Rechte holen
31 $userrights = $rightsdb->getrights(User => $signer);
32
33                                     # 'Sandkasten' einrichten
34 $compartment = new Penguin::Compartment;
35 $compartment->initialize(Operations => $userrights);
36
37                                     #  Snippet ausführen
38 $result = $compartment->execute(Code => $code);
39
40 if ($@) { $result = $@; }          # Befugnisse überschritten??
41
42 $mychannel->close();               # Verbindung schließen
43
44 print 'Rückgabewert: $result\n';
```

Penguin-Client, der ein Script vom Server holt und ausführt (Listing 3).

```
 1 #!/usr/bin/perl
 2 use Penguin::Frame::Code;
 3 use Penguin::Wrapper::PGP;
 4 use Penguin::Channel::TCP::Server;
 5
 6 $|=1;
 7 print('Paßwort: ');
 8 system('stty -echo');                    # Auf 'blinde' Eingabe schalten
 9 chop($password = <STDIN>);               # Paßwort einlesen
10 system('stty echo');                    # Terminal-Echo wieder an
11
12 $mychannel = new Penguin::Channel::TCP::Server Bind => 8118, Listen => 5;
13
14 while(1) {
15     $mychannel->open();
16
17     ($remotehost, $remoteport) = $mychannel->getinfo();
18
19     print 'Client: $remotehost\n';
20
21     $codeframe = new Penguin::Frame::Code Wrapper =>
22                                           'Penguin::Wrapper::PGP';
23
24     $codeframe->assemble(Password => $password,
25                          Text     => 'print 'Hier ist der Server!\n';',
26                          Name     => 'Der Gute Server');
27
28     $mychannel->putframe(Frame => $codeframe);
29 }
```

Dieser Penguin-Server schickt ein angefordertes Perl-Script zum Client (Listing 4).

wiederum Zeile 28 'auswickelt' und daraus einen (optionalen) Titel, den Absender, die Verpackungsmethode und schließlich das übermittelte Codefragment extrahiert. Stimmt etwas mit dem Frame oder der PGP-Signatur nicht, sei es, daß die Unterschrift gefälscht oder das Dokument manipuliert wurde, liefert *getframe* den Wert *undef* zurück.

Zeile 34 baut den Sandkasten auf, die nächste Zeile definiert die dortigen Spielregeln für den Absender des Frames anhand der vorher eingelesenen Rechtedatei. Zeile 38 führt den Code in sicherer Umgebung aus. Falls dort jemand seine Kompetenzen überschreitet, bricht die *execute*-Methode ab und setzt in *$@* eine aussagekräftige Fehlermeldung. Andernfalls enthält *$result* den Rückgabewert des Snippet.

Den zugehörigen Server zeigt Listing 4. Da er zum PGP-Signieren des gesendeten Frame Zugang zum Private Key benötigt, lesen die Zeilen 6 bis 10 die PGP-Paßphrase blind ein. Zeile 12 definiert einen Penguin-Kanal, der Server wird auf Kanal 8118 auf anfragende Clients warten und einen Eingabepuffer für 5 Kunden anlegen.

Ab Zeile 14 wartet die Endlosschleife mit *$mychannel->open()* blockierend auf hereinkommende Client-Anfragen. Zeile 17 liest nur für die Statusanzeige (beziehungsweise für ein zu implementierendes Logging) die IP-Adresse und Port des anfragenden Clients aus.

Zeile 21 bereitet den Code-Frame vor, in den Zeile 24 ein Stück Programm legt und den Namen sowie das PGP-Paßwort des Servers mitgibt, worauf die *assemble*-Methode das Ganze sauber verpackt und PGP-signiert. Zeile 28 schließlich sendet den Frame zum Client. Wie erwartet liefert der Aufruf von *dclient.pl* bei laufendem *dserver.pl* erst *Hier ist der Server!* und dann *Rückgabewert: 1*. (ck)

MICHAEL SCHILLI

arbeitet als Web-Engineer für AOL Productions, San Mateo. Er ist Autor von 'Effektives Programmieren mit Perl 5'.

Literatur

[1] Simson Garfinkel, Gene Spafford; Web Security & Commerce; O'Reilly 1997, ISBN 1-56592-269-7

[2] Dirk Fox; Beweismittel; Unterschriften auf der Datenautobahn, *iX* 12/97, S. 98 ff.