Techniken, um über geschlossene Ports zu kommunizieren

Bitte anklopfen

Unsichtbare Hintertüren in standfesten Außenmauern verbinden Sicherheit und Freiheit beim Kommunizieren: Wer die Tür und den Code nicht kennt, bemerkt nicht mal, dass es sie gibt. Jens-Christoph Brendel

Die Internetnutzer haben sich eingemauert. Die Ausstattung mit Firewalls nähert sich der Sättigungsgrenze, so gut wie jede firmen- oder behördeneigene Netzparzelle umgeben heute virtuelle Brandmauern.

Das hat ohne Zweifel seine Berechtigung, aber auch eine Kehrseite: Je höher und stabiler die Wallanlagen sind, desto mehr behindern sie auch die Kommunikation mit erwünschten Partnern. Außenstellen, Heimarbeiter, Partnerfirmen oder Fernwarter sind zunächst ebenfalls ausgesperrt.

Horch, was kommt ...

Doch es gibt Auswege aus dem Dilemma. Neben den üblichen Ansätzen – beispielsweise Tunnel für verschlüsselten Nachrichtenaustausch – existiert eine Reihe ungewöhnlicher und interessanter Techniken, die gebetenen Besuchern eine Geheimtür öffnen. Einen eigenen Lösungsversuch stellt der Autor in diesem Beitrag vor.

Unter verschiedenen Lösungsansätzen für solche Hintertüren wird häufiger das so genannte Port-Knocking diskutiert, mit dem sich Botschaften durch geschlossene Ports übermitteln lassen. Der Server befindet sich dabei hinter einer Firewall, die keinerlei Verbindungswünsche aus der Außenwelt akzeptiert. Eine spezielle Client-Applikation versucht es aber trotzdem und testet vorher vereinbarte Andockstellen in einer bestimmten Reihenfolge.

Alle diese Versuche müssen und sollen natürlich fehlschlagen, aber sie hinterlassen Einträge im Logfile der Firewall. Das Logfile überwacht ein Daemon. Erkennt er in den Meldungen über abgewiesene Verbindungsversuche das verabredete Muster, löst er eine bestimmte Aktion aus. Beispielsweise könnte er für den Anklopfer, der sich durch das Portnummern-Passwort authentifiziert hat, den SSH-Zugang öffnen.

Es gibt inzwischen eine ganze Anzahl Implementierungen der Klopfzeichen-Technik. Eine komfortabel installierbare Perl-Version [2] wurde in diesem Sommer auf der Defcon-Konferenz in Las Vegas vorgestellt. Eine andere Variante [3] horcht direkt an der Netzwerkschnittstelle statt am Logfile.

Für einen Hacker, der in einem größeren Netzbereich mit einem Portscanner nach verwundbaren Servern fischt, erscheint ein so geschütztes System als wenig interessantes Angriffsziel, denn es bietet scheinbar keine Schwachpunkte. Ein ernsthafter Angreifer wird sich auf diese Weise aber kaum abschrecken lassen. Ist er in der Lage den Netzverkehr mitzuschneiden, könnte er die geheime Sequenz der Verbindungsanfragen erkennen und in einer Replay-Attacke nutzen.

Leicht zu knacken

Das ist auch eines der Argumente der Klopfzeichen-Kritiker [4]. Eine ihrer Mindestforderungen – die Verschlüsselung des geklopften Passworts – erfüllen inzwischen die meisten Versionen. Allerdings ist auch dabei Vorsicht geboten: Beschränkt man die Anzahl der Klopfsignale und der möglichen Ports aus praktischen Gründen, dann entspricht auch das verschlüsselte Klopfzeichen nur einem Passwort mit sehr geringer Schlüssellänge, das relativ leicht zu brechen ist. Die Verschleierung des Mechanismus (Security through Obscurity) wendet sich dann unter Umständen gegen ihren Benutzer, der sich in trügerischer Sicherheit wiegt.

Dem lässt sich entgegenhalten, dass ein Überwinden der Port-Knocking-Barriere noch nicht den Zugriff auf den Server ermöglicht, sondern nur einen Dienst aktiviert (etwa SSH oder VPN), der seinerseits durch ein starkes Passwort und ein Public-Key-Verfahren geschützt ist. Besser wäre, wenn bereits die äußere Hülle eines mehrstufigen Sicherheitskonzepts die Mängel einfacher Port-Knocking-Implementierungen von vornherein vermeidet. Die vom Autor entwickelte Variante versucht genau dies zu leisten.

UDP-Geheimnisse

Bei dieser Technik ist auf der Serverseite ebenfalls ein Daemon im Spiel, der diesmal aber kein Logfile beobachtet, sondern sein Ohr an einen bestimmten UDP-Port hält. Da UDP ein verbindungsloses Protokoll ist, kann auch ein Port-

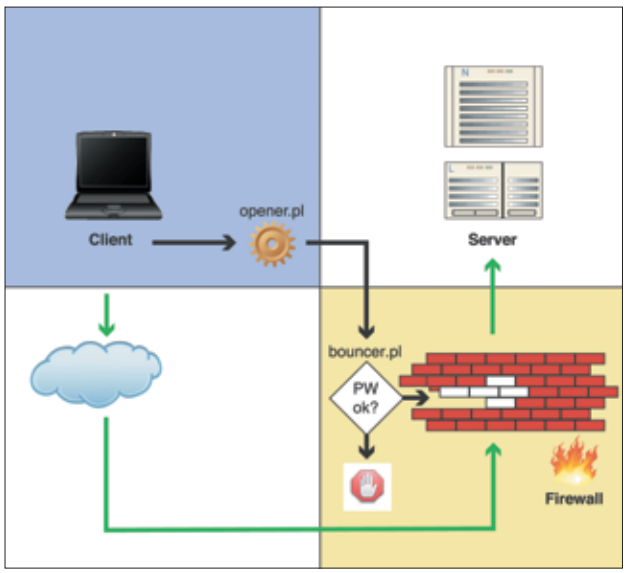

Abbildung 1: Nachdem Opener eine Geheimbotschaft an Bouncer versandt hat, verifiziert dieses Skript das enthaltene Passwort und öffnet für autorisierte Clients eine Hintertür in der Firewall.

scanner nicht testen, wie sein Zielsystem auf die ausgestreckte Hand einer Verbindungsanfrage reagiert. Stattdessen wertet er eine Fehlermeldung aus, die das Betriebssystem verschickt, wenn der kontaktierte Port seiner Meinung nach keinem Dienst zugeordnet ist: die ICMP-Message »Destination unreachable, Port unreachable«.

Auch im vorliegenden Fall wird diese Meldung als Antwort auf jedes Paket an den überwachten Port verschickt, denn der Daemon bearbeitet die dort ankommenden Nachrichten zwar, informiert aber sein Betriebssystem nicht darüber. Dieses beantwortet eintreffende Pakete in dem Glauben, der Port sei unbenutzt, mit der Unerreichbar-Meldung. Der Port bleibt für Außenstehende so unsichtbar wie beim klassischen Port-Knocking.

Ein Client weist sich dem Wächter-Daemon gegenüber als berechtigter Nutzer aus, indem er via UDP ein Paket mit einem Passwort und dem Code für eine Aktion verschickt. Wenn der Daemon das Passwort akzeptiert, führt er die dem Code zugeordnete Aktion aus.

Im Unterschied zu einer starren Klopfsequenz ist das Passwort bei dieser Variante aber jedes Mal ein anderes und wird außerdem zusammen mit dem Aktionscode verschlüsselt. Der Krypto-Algorithmus (wahlweise DES oder Idea), die Länge des Passworts und des Schlüssels sind konfigurierbar.

Daemon und Client sind Perl-Skripte. Der Client (Auszug in Listing 1, vollständig auf [6]) ist relativ einfach gestrickt. Die Voreinstellungen für Zielsystem, Port und Passwort können der Einfachheit halber am Anfang des Skripts fest verdrahtet und sonst auf der Kommandozeile übergeben werden. Für einen an Port 51324 lauschenden Server, der Aktion 21 starten soll, sieht das so aus:

```
./opener.pl -v -p 51324 -d server.xyz.com ➥
  -f 21
```

Der Aufruf »opener -h« erklärt kurz die Optionen.

Minuten-Passwort

Nach jedem Start erzeugt ein Generator ein neues Passwort. Dazu initialisiert er Perls Zufallszahlengenerator mit einem Startwert (Seed), den er aus der Uhrzeit ableitet. Das Skript »bouncer.pl« der Gegenseite – es muss mit Root-Rechten laufen – macht nach dem Empfang eines Pakets das Gleiche.

So erhalten beide die gleichen Pseudo-Zufallszahlen und darüber das gleiche Passwort. Damit das gut funktioniert, gleichen beide Seiten ihre Uhren am besten regelmäßig mit einem Zeitserver ab. Der Benutzer kann allerdings eine maximale Zeitdifferenz vorgeben – das Skript probiert dann alle Passwörter, die sich aus Uhrzeit plus/minus Toleranzwert ergeben. Der Daemon protokolliert alle Kontaktversuche und deren Ergebnis im Syslog (Listing 2). ▶

Listing 1: Auszug aus »opener.pl«

```
02 # Passwort-Generator
03 sub generate_pw {
04     my ( $sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst ) =
05       localtime( time() );
06     my $seedval = ( $yday * 24 + $hour ) * 60 + $min;
07     srand($seedval);
08     my @KeyElem = ( ( 'a' .. 'z' ), ( 'A' .. 'Z' ), ( 0 .. 9 ) );
09     my $pw = '';
10     for ( 1 .. $Prefs{l} ) {
11         my $i = int( rand(61) );
12         $pw .= $KeyElem[$i];
13     }
14     if ( $Prefs{v} ) { print "Password $pw generated with seedval
   $seedval \n"; }
15     return $pw;
16 }
18 # DES-Verschluesselung
19 sub encrypt_pw {
20     my ( $plaintext, $deskey ) = @_;
21     my $cipher = Crypt::CBC->new( $Prefs{k}, "Crypt::DES" );
22     my $ciphertext = encode_base64( $cipher->encrypt($plaintext) );
23     return $ciphertext;
24 }
27 # ===--> M A I N <---==
29
30 getOptions();
31
32 # Create a new socket
33 my $Socket = new IO::Socket::INET->new(
34     PeerPort => $Prefs{p},
35     Proto    => 'udp',
36     PeerAddr => $Prefs{d}
37 );
38
39 # Send messages
40 my $pw = generate_pw();
41 $pw = $pw . ":" . $Prefs{f};
42 my $key = encrypt_pw( $pw, $Prefs{k} );
43 if ( $Prefs{v} ) { print "Decrypted password + function $Prefs{f} = $key
   \n"; }
44
45 $Socket->send($key);
```

Im Unterschied zu einem richtigen, absolut sicheren One-Time-Pad ist das so gewonnene Passwort nicht wirklich zufällig. Das darf es auch gar nicht sein, denn der Wächter-Daemon muss, um es prüfen zu können, zur selben Zeit die gleiche Kombination erzeugen.

Gesetz des Zufalls

Aber es ergibt sich immerhin eine sinnlos wirkende Zeichenfolge, die minütlich wechselt – also weder erraten noch einfach wiederholt werden kann – und die im nächsten Schritt zusätzlich verschlüsselt wird. Für diese Verschlüsselung bemüht der Client das Perl-Modul Crypt::CBC, das seinerseits mit DES oder Idea arbeiten kann. Er chiffriert damit Passwort und Aktionscode, verpackt beide in ein UDP-Paket und schickt es an die Zieladresse.

Dort angekommen nimmt es der Daemon »bouncer.pl« im Stillen an. Er entschlüsselt den Inhalt, extrahiert das Passwort, vergleicht mit Hilfe eines identischen Passwortgenerators, ob er um diese Zeit zum selben Ergebnis gekommen wäre, und führt nach erfolgreicher Prüfung die Aktion aus, die der mitgeschickte Code ihm zuweist.

Aktionsplan

Das Daemon-Skript (Auszug siehe Listing 3, vollständig auf [6]) benötigt eine Reihe Perl-Module. Fehlende installiert man via CPAN (am einfachsten interaktiv: »perl -MCPAN -eshell«), denn dabei werden eventuelle Abhängigkeiten automatisch aufgelöst.

Der Wächter braucht mehr Einstellungen, unter anderem bis zu 100 Codes für beliebige Aktionen (die übrigens ihrerseits wieder Skript-gesteuert sein können). Er liest deshalb nach dem Start ein eigenes Konfigurationsfile. Diese Datei enthält Einträge nach dem Schema Option = Wert und könnte in einem einfachen Fall etwa so aussehen wie das Beispiel in Listing 4. Die Vorgaben darf der

Listing 2: Logmeldungen

```
01 Oct 12 21:46:31 hercules bouncer daemon[7028]: bouncer daemon started, PID: 7028
02 Oct 12 21:49:46 hercules bouncer daemon[7028]:
03 RECEIVED: XXX.XXX.XXX.XXX:63878 ->192.168.2.162:51324
04 Oct 12 21:49:46 hercules bouncer daemon[7028]: Time delta: 0
05 Oct 12 21:49:46 hercules bouncer daemon[7028]: seed: 411709 password: t8V7KLzK
06 Oct 12 21:49:46 hercules bouncer daemon[7028]: Received data: UmFuZG9tSVYfikabk2
07 8jcPGw32/ePuz23ZritNGwZNc=
08 Oct 12 21:49:46 hercules bouncer daemon[7028]: Estimated  pw: t8V7KLzK
09 Oct 12 21:49:46 hercules bouncer daemon[7028]: Decrypted  pw: t8V7KLzK:21
10 Oct 12 21:49:46 hercules bouncer daemon[7028]: ACCESS GRANTED - function 21
```

Listing 3: Auszug aus »bouncer.pl«

```
001 ...
002 # Redirect standard file descriptors from and to /dev/null
003 # Fork a child process, kill the parent
004 # Create a new session with the calling process as the
005 # leader of the session and the process group
006 # Detach the controlling terminal
007 sub daemonize {
008     chdir '/' or die "Can't chdir to /: $!";
009     open STDIN,  '/dev/null'  or die "Can't read /dev/null: $!";
010     open STDOUT, '>/dev/null' or die "Can't write /dev/null: $!";
011     defined( my $pid = fork ) or die "Can't fork: $!";
012     exit if $pid;
013     setsid or die "Can't start a new session: $!";
014     syslog( 'notice', 'bouncer daemon started, PID: ' . $$ );
015     umask 0;
016     open STDERR, '>&STDOUT' or die "Can't dup stdout: $!";
017 }
018
019 # callback function
020 sub analyzePackets {
021     my ( $user_data, $header, $packet ) = @_;
022
023     #   Strip ethernet encapsulation of captured packet
024     my $ether_data = NetPacket::Ethernet::strip($packet);
025
026     #   Decode contents of TCP/IP packet contained within
027     #   captured ethernet packet and then the UDP datagram
028     #   contained within the ip packet
029     my $ip  = NetPacket::IP->decode($ether_data);
030     my $udp = NetPacket::UDP->decode( $ip->{'data'} );
031
032     #   Print all out where its coming from and where its
033     #   going to
034     syslog( 'debug',
035             "RECEIVED: "
036           . $ip->{'src_ip'} . ":"
037           . $udp->{'src_port'} . " -> "
038           . $ip->{'dest_ip'} . ":"
039           . $udp->{'dest_port'} );
040
041     my $dat = $udp->{'data'};
042     for ( my $i = $Prefs{delta} * -1 ; $i <= $Prefs{delta} ; $i++ ) {
043         my $pw = generatePW($i);
044         syslog( 'debug', "Received data: $dat" );
045         syslog( 'debug', "Estimated  pw: $pw" );
046         my $cipher = Crypt::CBC->new( $Prefs{deskey}, "Crypt::DES" );
047         my $tmpkey = $cipher->decrypt( decode_base64($dat) );
048         syslog( 'debug', "Decrypted  pw: $tmpkey" );
049         my $key = substr( $tmpkey, 0, $Prefs{pwlen} );
050         my $fnc = substr( $tmpkey, $Prefs{pwlen} + 1, 2 );
051
052         if ( $key eq $pw ) {
053             syslog( 'alert', "ACCESS GRANTED - function $fnc" );
054             system $Prefs{$fnc};
055             last;
056         } else {
057             syslog( 'alert', "ACCESS  DENIED -  wrong password: " .
    $dat );
058         }
059     }
060 }
061
062 #          ==> M A I N <-==
063
064 getOptions();
065 readConfig();
```

Admin dort auch im laufenden Betrieb ändern. Dann muss er dem Daemon nur via »kill -HUP *Daemon-PID*« signalisieren, dass der sich dem File erneut widmen soll.

Dies bewerkstelligt der Daemon über einen eigenen Signalhandler, der eine kleine Besonderheit aufweist: Er muss via »POSIX::SigAction« implementiert sein, weil das eine sofortige Bearbeitung erzwingt. Geht man anders vor und hinterlegt zum Beispiel im »%SIG«-Hash den Verweis auf die Handler-Routine, dann passiert erst mal gar nichts. Denn alle neueren Perl-Versionen ab 5.7.3 verwenden Deferred Signals (also verzögerte Signale), die hier etwas kontraproduktiv wirken.

Hört die Signale

Das Perl-Skript registriert dabei das Signal, schiebt dessen Verarbeitung aber aus Sicherheitsgründen auf, denn viele Perl-Funktionen sind nicht reentrant. Als geeigneten Zeitpunkt sieht es beispielsweise den Moment an, bevor ein neuer Perl-Opcode interpretiert wird. In den meisten Fällen ist eine solche Stelle ohne merkliche Verzögerung erreichbar. Für den lauschenden Daemon in seiner I/O-Schleife ergäbe sich die Gelegenheit aber erst nach dem Empfang eines neuen UDP-Pakets.

Eisblock

Die heimliche UDP-Kommunikation erhöht fühlbar die Kosten einer Attacke und wehrt als äußerer Festungsring das feindliche Fußvolk ab. „Festungen, welche der Verteidiger vor sich lässt“, schreibt der Stratege Clausewitz, „brechen wie Eisblöcke den Strom des feindlichen Angriffs.“ [5] ■

Infos

[1] Port-Knocking-Software: [http://www.portknocking.org]
[2] Port-Knocking-Tools: [http://www.cipherdyne.org/fwknop/]
[3] Port-Knocking-Daemon: [http://www.zeroflux.org/knock/]
[4] Kritik am Port-Knocking: [http://www.newsforge.com/article.pl?sid=04/08/02/1954253]
[5] Clausewitz online: [http://www.clausewitz.com/CWZHOME/VomKriege/VKTOC.htm]
[6] Listings dieses Beitrags: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/12/Portknocking/]

Listing 4: »bouncer.cfg«

```
01 #interface, key, password, passwprd length
02 iface  = eth0
03 deskey = 3gQMhzyy
04 pwlen = 805
06 # adresses
07 watch_ip   = remote.client.de
08 watch_port = 51324
09 # max timedifference (minutes)
10 delta = 2
11 # action codes (demo)
12 21 = /usr/local/bin/fw_open_script $ip->{'src_ip'}
13 22 = /usr/local/bin/fw_close_script
```

Listing 3: Auszug aus »bouncer.pl« (Fortsetzung)

```
066
067 # some hard coded preferences
068 my $snaplen = 256;     # read the first 256 byte per packet
069 my $promisc = 0;       # no promiscuous mode
070 my $to_ms   = -1;      # 0: capture packets until an error
071                        #-1: capture packets indefinitely.
072 my $err;               # pcap error messages
073 my $filter = 'udp and dst port ' . $Prefs{watch_port};
074 my $filter_c;          # compiled paket filter
075
076 # Initialize syslog
077 openlog( 'bouncer daemon', 'cons,pid,ndelay', 'daemon' );
078 setlogsock('unix');
079
080 # Use network device passed in the config file or if no
081 # argument is passed, try to determine an appropriate
082 # interface by lookupdev()
083 my $dev = $Prefs{iface};
084 unless ( defined $dev ) {
085     $dev = Net::Pcap::lookupdev( \$err );
086     if ( defined $err ) {
087         die 'Unable to determine network device for monitoring - ',
    $err;
088     }
089 }
090
091 # Check on bogus network device arguments that may be
092 # passed to the program as an entry in the configuration file
093 my ( $address, $netmask );
094 if ( Net::Pcap::lookupnet( $dev, \$address, \$netmask, \$err ) ) {
095     die 'Unable to look up device information for ', $dev, ' - ',
    $err;
096 }
097
098 # Create a packet capture object on the device
099 my $object = Net::Pcap::open_live( $dev, $snaplen, $promisc, $to_ms,
    \$err );
100 unless ( defined $object ) {
101     die 'Unable to create packet capture object on device ', $dev, ' -
', $err;
102 }
103
104 # Compile and set packet filter for packet capture object.
105 Net::Pcap::compile( $object, \$filter_c, $filter, 0, $netmask )
106   && die 'Unable to compile packet capture filter_c';
107
108 Net::Pcap::setfilter( $object, $filter_c )
109   && die 'Unable to set packet capture filter_c';
110
111 #          ==> D A E M O N I Z E <==
112
113 &daemonize;
114
115 use POSIX;
116 sigaction SIGTERM, new POSIX::SigAction sub {
117     Net::Pcap::close($object);
118     syslog( 'alert', 'Bye, bouncer daemon is exiting' );
119     closelog;
120     exit(1);
121 };
122
123 sigaction SIGHUP, new POSIX::SigAction sub {
124     syslog( 'alert', 'Reloading the config file' );
125     readConfig;
126 };
127
128 Net::Pcap::loop( $object, -1, \&analyzePackets, '' )
129   || die 'Unable to perform packet capture';
```