Port knocking from the inside out

Martin Krzywinski

www.portknocking.org

1. Introduction

Leaving a port open to the public an invitation for an intruder. Unfortunately, most services such as HTTP or SMTP need to be open for everyone to see. However, some of the more critical services may be accessible only when required. Here's where port knocking comes in.

In this article, I will cover

- what port knocking is, how it works and what it's used for,
- how to write the simplest implementation of port knocking,
- how to set up the Doorman program for advanced port knocking,
- how port knocking can be detected and compromised.

If you are running strongly-authenticating network services (e.g. SSH) on your server, it is likely that you're using the service's authentication and encryption process to discriminate between legitimate and illegitimate users. The legitimate user has a password and the illegitimate user does not. In order for the service to establish this fact, the illegitimate user is offered a chance to interact with the service.

Given that password guessing methods offer little chance of penetrating a well-maintained system, an intruder will try to bypass the authentication component of the service and try to elevate their privileges by exploiting a known bug, such as a buffer overflow. Since a zero-day exploit that targets your service may appear any day, maintaining a network service is not a passive activity. You must monitor vulnerability advisories and keep an eye out for patches to plug new security holes. But let's face it – reading advisories is boring, thankless and 99.99% of the time uninformative. It may be a year or two, or never, before an exploit for your version of a service is found and reported. So what can you do?

First, consider the fact that services with an finite user base do not need to have their ports open at all times. Unlike public services such as SMTP or HTTP, which need to receive connections from anyone and anywhere and usually do not require authentication, SSH is one such service which permits only password-bearing users in. Now, imagine being able to keep the SSH port (tcp/22) closed, thus making the service inaccessible and protected from exploits, until the service is actually requested by one of your legitimate users.

But how is this service requested if ports are closed and no connections are possible? This is where *port knocking* comes in. Port knocking permits a user to request for a port to be opened in front of a network service. This request takes the form of a passive sequence of authentication packets across closed ports on the server (see Inset *Port knock – information across closed ports*). It is possible to send information across closed ports,

The port knock – information across closed ports

Contrary to common opinion, you do not need open ports to transmit data. Even if connections are denied by an IP filter, these attempts are logged and therefore can be mined for information.

In the simplest case, consider ports A and B which are both closed, with no associated listening application. If the IP filter is configured to monitor packets arriving at these ports then it is straightforward to parse out the filter logs and identify the port sequence (e.g. *ABAAABBAA*) associated with a client IP. This port sequence can be used to encode information. The encoding could be a map between a specific port sequence and piece of information (e.g. *ABA = open port 22 for 15 minutes*, *ABB = close port 22*, *BAA = close port 22 and do not permit additional connections*). The encoding could also be binary with *A*/*B* representing 0/1.

even if the ports are closed and no network services are listening, because your firewall or another utility like tcpdump can be configured to monitor all incoming packets, even if they never reach an application like SSH.

1.1 Limitations of IP filtering

One way to limit the cross-section of your network service is to use an IP filter (see Inset *IP filters*), like netfilter/iptables. The filter would be configured to only allow connections from IP addresses from which your user base is connecting. This list would include remote offices and homes.

This approach does well to limit the number of sources from which connections are allowed, but is limited in its ability to deal with mobile users or offices/homes whose external IP addresses are dynamically assigned. If your users frequently change computers or networks, it may not be practical to maintain a changing list of IP addresses from which connections are permitted. This is especially true when users try to connect from notoriously untrustworthy locations like Internet cafes or university laboratories.

IP filters

The role of an IP filter is to control the passage of packets through the TCP/IP stack, based on the content in their headers. Thus, an IP filter may prevent packets originating from specific MAC or IP addresses from travelling further up the stack, denying the delivery of the packet to an application.

An IP filter (e.g. iptables) operates in OSI layer 3 of the TCP/IP stack (for explanation of OSI layers, see Inset *Seven layers of OSI* in the Article *Attacks on layer two of the OSI model* in the current issue of *hakin9*) and controls the flow of packets based on content in their IP headers, which includes source and destination IP address, protocol (TCP, UDP, etc.), and TTL. Most IP filters today, including iptables, reach lower (layer 2) and higher (layer 4), and can filter packets by lower-level information (MAC address) as well as higher-level information found in protocol headers (TCP or UDP), which includes source and destination ports and, for TCP, additional information like sequence number and flags (SYN, ACK, FIN and others).

Using an IP filter makes the assumption that behind a trusted IP address, or network address, are only trusted users. This is clearly not always true, especially when the trusted IP address is a gateway of a large internal network. It's easy to imagine a case in which attacks and legitimate connections originate from the same IP.

Port knocking provides a mechanism with which these issues can be addressed. With port knocking, the correspondence between an IP address and individual user is no longer necessary. The users can identify themselves using their authentication tokens without requiring any ports to be open on the server. Port knocking therefore allows a specific user to

connect from any IP, rather than any user to connect from a specific IP. This is an important distinction, as the number of individual devices from which a user can establish a connection is generally increasing (office computer, home computer, laptop, PDA, cell phone, etc.).

1.2 Requirements

Chances are, if you're reading this you have everything you need to protect your systems with port knocking. First, you'll need an addressable firewall, like iptables. Next, you'll need a sufficient number of unused ports that you can allocate for monitoring for knocks. If authentication is done through a sequence of port knocks, theoretically all you need is two ports, However, the more ports you have (e.g. 1,024 or even 16,355), the more complex, and therefore flexible, authentication data can be encoded into the same number of ports. If authentication is done through a packet bearing a payload, you only need a single port.

There are a large number of implementations of port knocking around and we will profile Doorman in this article. First, let's review some TCP/IP fundamentals to better understand what is going on.

2. Network Communication

When data is sent between two computers it is processed by a number of software and hardware layers to ensure specific delivery to the appropriate application. There are many ways of sending data over the Internet, and we'll briefly focus on TCP and UDP here, which are two common protocols. We'll first look at the anatomy of an Ethernet packet and then briefly review how TCP communication takes place. Knowing about packet structure will help us understand how port knocking works. Some implementations, for example, incorporate authentication tokens inside the packet in a non-standard way.

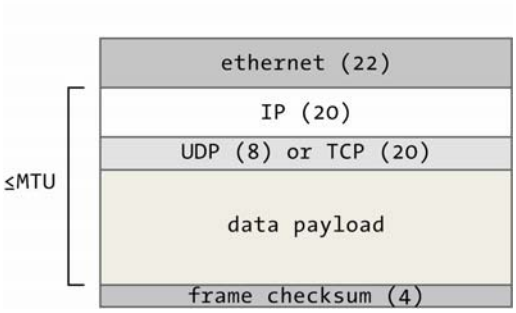

Figure 1. A packet sent using TCP or UDP protocol contains three headers: Ethernet (22 bytes), IP (20 bytes) and TCP or UDP (8 or 20 bytes), a variable-length data payload section and a final frame checksum which is added at the network access layer

In IP transmission, data is encapsulated into packets (see Inset *Packets and datagrams*) with several levels of headers (see Figure 1). At three of the four TCP/IP stack layers, another header is added. The Ethernet header is added at the data link layer (layer 2), the IP header at the network layer (layer 3) and protocol-specific header (e.g. TCP or UDP) at the transport layer (layer 4). In each case the header of a layer further up the stack is part of the data payload of the previous layer.

The Ethernet standard (IEEE 802.3) permits the data payload at the network access layer to be up to 1500 bytes (Maximum Transfer Unit, or MTU). With recent advancements in data transfer speeds, it is commonly thought that the MTU is too small, causing too much

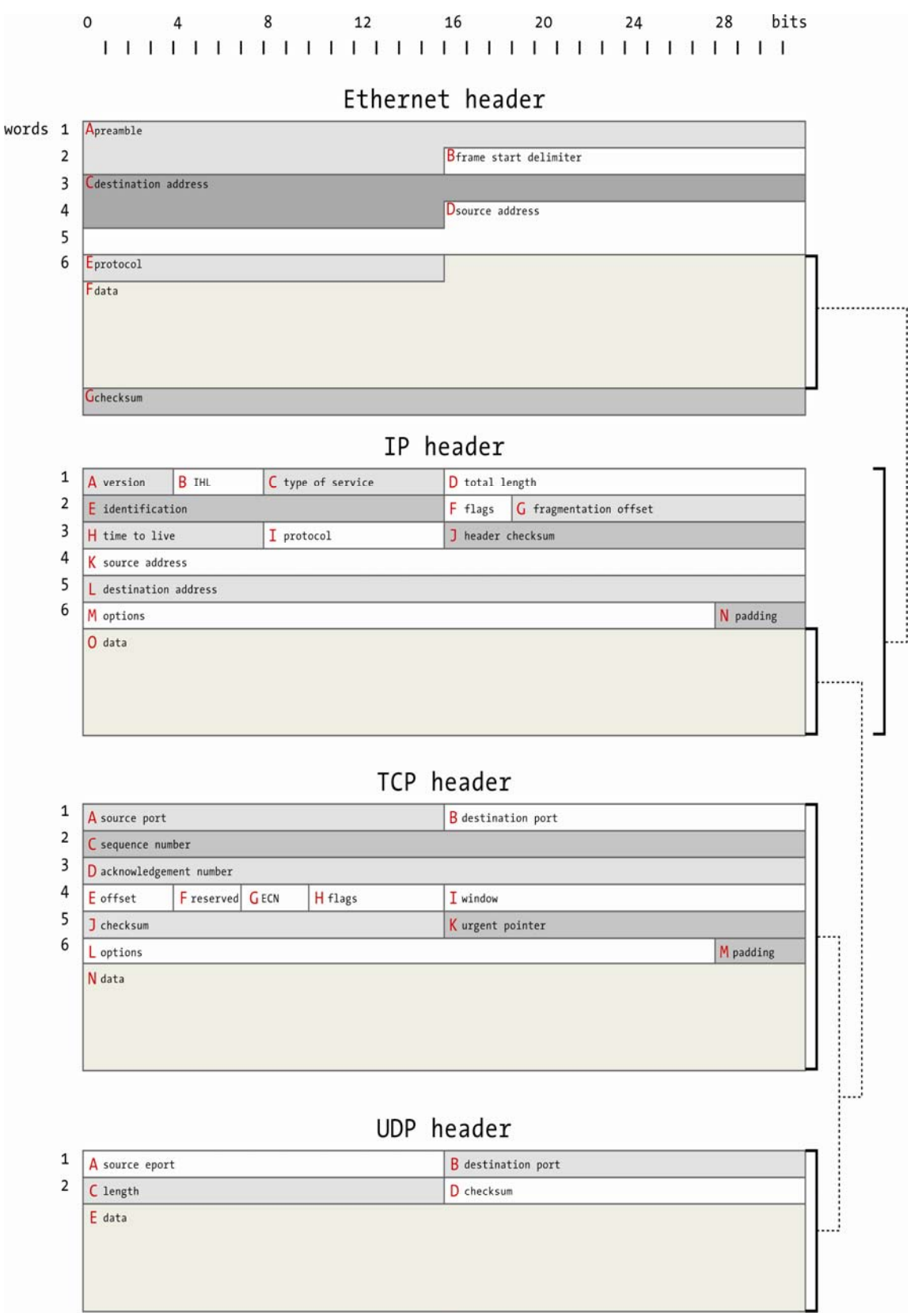

Figure 2. Ethernet, IP, TCP and UDP headers all contain information critical to successful transport of the data from their respective network layers.

to include support for larger MTUs (12 kb for a 100 Mbit link, for example).

3. Packets and datagrams

A *packet* is a block of data that contains all the information necessary to deliver it (see Figure 1). Think of a postal letter. As packets travel across the Internet, various network devices examine and manipulate the packets on their way from destination to source.

A *datagram* is a specific packet format defined by the IP protocol (see Figure 2). The datagram contains the IP header, protocol-specific header (TCP, UDP, etc.) and the data payload. In an Ethernet packet, the datagram is the data payload at the network layer (OSI layer 3).

Within each network layer header, a variety of information is stored (see Figure 2) to provide the hardware and software that operates at that layer to correctly deliver the data up or down the TCP/IP stack. Later, we'll examine a packet in detail and point out the location and form of some of the important fields in these headers.

There are generally two types of port knocking systems: those that use UDP and those that use TCP (see Inset *UDP or TCP for Port Knocking*)**.** A few implementations use ICMP, or combinations of protocols. TCP systems make use of a part of the three-way handshake in TCP (see Inset *TCP and the three-way handshake*), a preamble in the TCP conversation between two computers during which a connection is established and the initial value for a packet order counter is set. TCP port knocking systems may encode information in the destination port values in a sequence of packets, or in the header or data payload of a single packet. We will see an example of how to add data to a packet header to achieve this. UDP systems typically send a single packet with authentication data located in the data payload part of the packet.

Port states

A port is one component of a network socket, the others being the IP address and protocol. The IP address is specific to a network device, while the port is specific to an application that is designed to receive the packet. Ports are numbered and for TCP or UDP they range from 0 to 65,355. A given network device typically has a large number of sockets open for a variety of ports. SSH for example is conventionally assigned to port tcp/22.

The fate of a packet destined on a specific port largely depends on the IP filter on the server that receives the packet. In iptables, a port can be found in three conditions:

- A port is *OPEN* when the IP filter permits packets to pass up the TCP/IP stack and reach the application. Whether the application is running or not is optional and not of concern to the IP filter. The filter simply permits the packet to reach the application layer.
- If the port status is *REJECT* then the server returns an ICMP error packet back to the connecting client, politely telling the client that the connection has been refused. In this mode, the client receives confirmation of the server's existence. Rejected packets do not reach the application layer.
- Finally, when a port is set to *DROP* (or *DENY* in ipchains) connections, the server silently ignores connection attempts and does not return an error packet. In this mode the client has no confirmation of the server's existence.

Pioneering efforts

Two prior efforts that implemented a variant of port knocking, even before the term was coined, are cd00r and SAdoor. cd00r by FX of Phenoelit was created to provide access to a remote machine that did not advertise open ports. It is a minimal C implementation and initiates and inetd daemon when TCP SYN packets are detected at a specific, fixed sequence of ports. SAdoor by CMN of Darklabs was influenced by cd00r. It relies on authentication by a sequence of specifically formatted key packets, followed by a final command packet which stores an encrypted command to be executed on the server within its payload.

4. Port Knocking in UDP and TCP

The decision to use TCP or UDP is up to the programmer. Both have benefits and neither has significant disadvantages. Port knocking implementations that use TCP make use of the SYN packet to authenticate the client, although the SYN flag isn't strictly required. Since the client is free to send sequences of packets, SYN or otherwise, to any remote socket, the combination of ports to which these packets are sent can be used to encode data. Since the packets are not acknowledged by the server (port knocking is passive), some implementations choose to use UDP as the protocol, which is a more natural choice of protocol when acknowledgement is not required.

The benefits of TCP are that the TCP header is larger and can accommodate more authenticating information (sequence and acknowledgement numbers provide storage room for 4 bytes each). If a lot of TCP packets are sent, it's important for the server to be able to reconstruct their initial order, which can be achieved by using the sequence identification number in each packet.

UDP, on the other hand, has a much smaller header (8 bytes versus 20 bytes) and requires that data be placed in the payload of the packet. It can be argued therefore that UDP is a more practical and elegant way to send information in a single, unacknowledged packet. A single UDP packet is always more stealthy than a characteristic sequence of TCP packets to various ports.

Regardless whether UDP or TCP is used, a port knocking server allocates a set of closed ports (see Inset *Port states*) and monitors them for specially formatted packets which form the port knock. The knock plays the role of a personalized trigger – each user may have their own individual knock constructed from their authentication tokens, or other personal information.

Port knocking is a unique authentication system because the client sends his authentication tokens across a closed ports without acknowledgement of receipt. The

TCP and the three-way handshake

TCP is a *connection-oriented*, *byte-stream*, *reliable* protocol. Each of these terms has a specific definition in the context of data transmission. Aspects of TCP communication can be subverted in a port knocking system to permit authentication across a closed port.

The TCP connection first begins by the server opening a socket and listening passively for incoming connections. A socket is a combination of IP address, protocol and port. A remote client wishing to connect to the server's socket, which will typically be associated with a specific service (e.g. SSH), will send a TCP packet with its SYN flag (*synchronize sequence numbers*) set to indicate a request for connection. The purpose of the SYN packet is to ask for the initial value of an index incorporated into the TCP header (sequence number) to permit the client to reconstruct the initial order in which the packets were sent.

If the connection attempt is permitted by the IP filter, the TCP-enabled application will respond by sending back a SYN/ACK (ACK for *acknowledgement*) packet with the initial sequence number. The client will acknowledge receipt by sending an ACK packet, thus completing the handshake. The handshake preamble makes TCP a *connection-oriented* protocol.

TCP is also a *byte-stream protocol*, because it sends data in an ordered fashion, made possible by the sequence number. Thus, the client may receive the packets in any order and using the sequence number it can determine the intended order.

TCP is a *reliable* protocol because throughout the TCP connection the client periodically acknowledges that it has received data. If the sender does not receive acknowledgement of receipt after it has sent a mutually agreed upon amount of data within a prescribed time, it will send the data again.

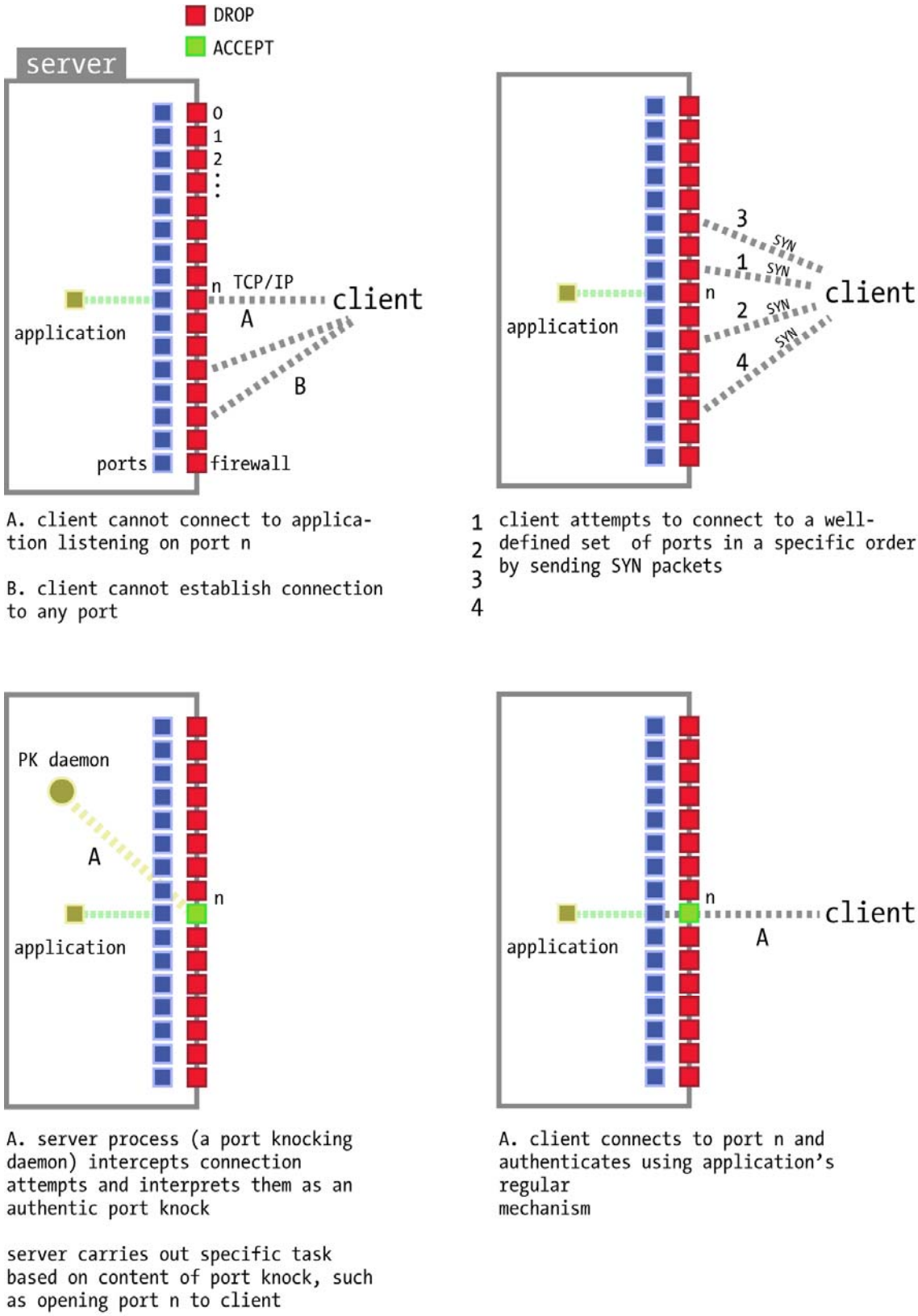

Figure 3. Phases in traditional port knocking using a sequence of ports address to by TCP SYN packets to encode authentication information.

place, or whether it is successful. This aspect of port knocking is what makes it much more difficult to detect and subvert by an intruder. A correctly formatted knock triggers the server to perform an action in accordance to instructions stored in the knock. The server may open or close a port for the client's IP address or perform any other action, such as send email, perform a backup or even shut down.

4.1 Traditional implementation

Port knocking was first formally described in *SysAdmin* magazine, although a couple of proof-of-concept projects were already in existence (see Inset *Pioneering efforts*). The initial port knocking specification described authentication through a series of SYN packets. The port sequence was used to encode the authentication tokens. One such example was an 8 port knock that contained the following bytes: IP address, port, time and checksum.

Instead of using the IP address stored in the packet's header, which can forged, the client inserted its IP address into the knock. The port and time fields stored the port to which the client wanted access and the amount of time to keep the port open. The 1-byte checksum was present to permit the client to validate the integrity of the knock. This knock was then encrypted and encoded onto a range of closed server ports. The server, upon receiving the knock SYN packets, would extract the destination port from each packet, decode and decrypt this port sequence and then act accordingly to the instructions encoded in the knock. This traditional port knocking process is illustrated in Figure 3.

Encrypting the port sequence is important to prevent spoofing and man-in-the-middle attacks, although does not address the issue of replay attacks which we will describe later. In this embodiment, port knocking is extremely easy to implement. The client can craft packets using a dedicated packet generator like sendip, obviating the need for a dedicated knocker. Incoming SYN packets can be logged to a iptables log file and thus can be monitored by even the most primitive file handling tools like grep. Alternatively, incoming packets can be monitored directly using tcpdump.

4.2 Port Knocking with sendip and tcpdump

What makes port knocking an attractive system is its conceptual simplicity and relative ease of implementation. Although robust port knocking systems designed for high-traffic environments require significant skill to create, a DYI solution can be rolled out by making use of commonly used tools. Let's look at one such very primitive implementation.

The purpose here will be to design as system which will allow us to establish an SSH connection to a system which initially presents no open ports. We'll use SendIP to craft and send trigger packets to a server, which will be monitoring for these packets using tcpdump. When an appropriately formatted trigger packet is received, the server will open the requested port to the incoming IP for 10 seconds to allow a connection to be initiated. After this timeout, the firewall rules will be reset and no new connections will be permitted. For this and all examples, we'll be using Red Hat Fedora 4 with the stock 2.6.11-1.1369_FC4 kernel, iptables 1.3.0, tcpdump 3.8, libpcap 0.8.3, and SendIP 2.5-1. Our port knocking server will be at IP 10.1.17.90 and our client will be at IP 10.1.17.1.

Listing 1. An initial firewall rule set

```
# Firewall configuration written by system-config-securitylevel
# Manual customization of this file is not recommended.
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -s 0/0 -d 0/0 -p udp -j DROP
-A INPUT -s 0/0 -d 0/0 -p tcp --syn -j DROP
COMMIT
```

We'll start with a very conservative firewall rule set on the server (see Listing 1 and 2). This rule set does not permit new TCP connections or UDP packets to any port, but permits established connections to continue.

We will trigger the opening of a port of our choice with a single TCP SYN packet. This packet will be sent to a port in the range 1000–10999 and have the sequence number in the header set to 17. These two conditions are both simplistic and arbitrary. The point is to make the packets reasonably unusual so that they can be distinguished from regular traffic. Using SendIP, let's send a packet to port 1022 on the server, for reasons that will become apparent very soon. The SYN flag is set using -tfs 1 and the sequence number is set using -tn 17

```
$ sendip -p ipv4 -p tcp -is 10.1.17.1 -ts 1000 -td 1005 -tfs 1 \
  -tn 17 10.1.17.90
```

Listing 2. An initial firewall rule set – table list

```
# iptables -L
Chain FORWARD (policy DROP)
target     prot opt source      destination

Chain INPUT (policy DROP)
target     prot opt source      destination
ACCEPT     all  --  anywhere    anywhere
ACCEPT     all  --  anywhere    anywhere       state
RELATED,ESTABLISHED
DROP       udp  --  anywhere    anywhere
DROP       tcp  --  anywhere    anywhere       tcp
flags:SYN,RST,ACK/SYN

Chain OUTPUT (policy ACCEPT)
target     prot opt source      destination
```

On the server, we'll have an instance of tcpdump listening for these trigger packets. We will want to limit the packets to SYN TCP packets addressed to the port range 1000–10999 with a sequence number 17. The tcpdump command that accomplishes this is:

```
$ tcpdump -vv -O "tcp[2:2] >= 1000 and tcp[2:2] <= 10999 \
  and tcp[4:4] = 17 and tcp[tcpflags] & tcp-syn != 0"
```

Unfortunately, tcpdump does not support port ranges applied to the “port” primitive and we must use tcp[2:2] to indicate the value of the destination port. This syntax indicates a 2 byte field in the TCP header starting at byte 2 (*tcp[start:length]*). It's also important to

turn off matching engine optimization (-O) since there are known bugs in some versions of tcpdump with using relative operators like <= and >=.

Let's try sending a couple of packets, the first to port 1005 (-td 1005) and the second to port 1022 (-td 1022). tcpdump will detect these packets and display their header information (see Listing 3).

Listing 3. tcpdump detecting packets to ports designated for port knocking.

```
21:24:48.931043 IP
  (tos 0x0, ttl 255, id 23105, offset 0, flags [none], proto 6, length: 40)
  10.1.17.1.1000 > 10.1.17.90.1005: S [tcp sum ok] 17:17(0) win 65535
21:24:54.033659 IP
  (tos 0x0, ttl 255, id 2483, offset 0, flags [none], proto 6, length: 40)
  10.1.17.1.1000 > 10.1.17.90.1022: S [tcp sum ok] 17:17(0) win 65535
```

If we are sending information across a closed port, where is the information? Well, we will be using the source IP address of the packet along with the destination port to change the firewall settings on the server. The firewall will be opened for 10 seconds to the source IP address for connection to the destination port with the leading *10** stripped. Thus, the two trigger packets would permit connections from 10.1.17.1 to ports 5 (1005 without the leading 10*=100) and 22 (1022 without the leading 10) respectively. We will accomplish triggering the firewall rule set with the bash script *guard.sh* presented on Listing 4.

Listing 4. guard.sh – a bash script for triggering firewall rules for opening connections

```
#!/bin/bash
# guard.sh
# allow incoming packets
/sbin/iptables -I INPUT -j ACCEPT -i eth0 -p tcp -s $1 --dport $2
sleep 10
# deny incoming packets
/sbin/iptables -D INPUT -j ACCEPT -i eth0 -p tcp -s $1 --dport $2
```

To open port 22 for 10 seconds for source IP 10.1.17.1, the script would be called as follows:

```
$ guard 10.1.17.1 22
```

Now that the pieces are in place, the only thing left to do is to get tcpdump to trigger the execution of the script as it is scanning for packets. This will be accomplished by piping tcpdump to xargs and constructing a bash command based on the contents of the output of tcpdump:

```
$ tcpdump -l -O "tcp[2:2] >= 1000 and tcp[2:2] <= 10999 \
    and tcp[4:4] = 17 and tcp[tcpflags] & tcp-syn != 0" \
  | sed -u 's/.*IP \([0-9]*\.[0-9]*\.[0-9]*\.[0-9]*\)\
    .*\.10*\([0-9]*\): S.*/\1 \2/'
  | xargs -t -l -i bash \
    -c "guard.sh {} &"
```

There's quite a bit going on here, so let's look at it a pipe at a time. We've already seen the output of tcpdump. In this case, we're not using the verbose flag (-vv) since we don't need all the extra header information and are using the -l flag to avoid buffering the output of tcpdump. The output is handled by sed to remove everything from the line except the IP

address and port number. Thus the line:

```
22:28:31.750989 IP 10.1.17.1.1000 > 10.1.17.90.1022: S 17:17(0) win 65535
```

Will be transformed to

```
10.1.17.1 22
```

Finally, xargs is used to construct a background command to call the *guard.sh* bash script.

The pieces are all in place. When the trigger packet is sent from the client to the server, tcpdump will print the packet header, sed will transform the output and xargs will call *guard.sh* with the resulting text as parameters. Immediately afterwards, a port will be opened to permit the client to connect (see Listing 5). This door will disappear in 10 seconds, which should be plenty of time for the client to initiate a connection.

Listing 5. A port in the firewall is opened

```
# iptables -L
(...)
Chain INPUT (policy DROP)
target     prot opt source          destination
ACCEPT     tcp  --  10.1.17.1      anywhere        tcp dpt:ssh
ACCEPT     all  --  anywhere       anywhere
ACCEPT     all  --  anywhere       anywhere        state RELATED,ESTABLISHED
DROP       udp  --  anywhere       anywhere
DROP       tcp  --  anywhere       anywhere        tcp flags:SYN,RST,ACK/SYN
(...)
```

Why were we able to delete the initial rule that allowed the client to connect without rudely terminating its SSH session? Because of another existing rule that tracks connections and accepts packets associated with established connections, which are connections for which packets from both directions have been seen (e.g. after a three-way handshake, for example, a TCP connection is established).

This simple example shows that no knowledge of network programming or the TCP/IP stack is necessary to implement port knocking in its simplest form. That being said, don't use this implementation for your daily use – the trigger packet is trivial to replay and offers no protection against forgery. The example serves to illustrate that you don't need to do a lot of work to significantly bolster the security of your system. If you are administering remote machines via SSH, and are the only one logging in, this example shows how you can maintain connectivity from anywhere without opening the SSH port to everyone.

4.3 Port knocking with a twist

There are a number of implementations of port knocking (see Figure 4) and Bruce Ward's Doorman takes a different and more elegant approach to the canonical port knocking specification. We'll use Doorman to show you how to set up port knocking on your system, because it uses a single UDP trigger packet (see Inset *UDP or TCP for port knocking*), supports a password file, includes anti-replay measures, and provides out-of-the-box compatibility with a variety of firewalls, including ipchains, iptables and pf. Our goal is to set up Doorman to accept connections from two fictitious users who will be

able to connect to network services on a closed system. We'll also analyse the way Doorman works to see what techniques it uses for security.

Instead of sending authentication information by connection attempts to closed ports, Doorman uses a UDP packet containing 4 strings for authentication and access control. The payload consists of a MD5 hash of a shared secret, created using as a key a combination of the requested port, user ID and a random number. When the Doorman daemon receives this kind of packet, it verifies the validity of authenticating information by looking up the secret phrase for the corresponding user/group ID found in the packet and creating its own version of the MD5 hash. It then compares this hash with the one found in the packet and permits access to the specified port if they match.

5. Configuring Doorman

Using Doorman (for download address, see Inset *On the Net)* to protect your system is straightforward. The compilation and installation instructions can be found in the INSTALL and README files. The service binary is *doormand* and the knocker script is *knock*.

Listing 6. Contents of the guestlist file

```
martink  hushhush  22        10.1.17.0/24
jacekz   pizzapie  22 23 25  192.168.0.0/16
```

The *doormand* daemon listens on a particular port for UDP packets with a specific payload. The UDP port is defined in configuration file */usr/local/etc/doorman/doorman.cf*. The *EXAMPLE* file found in the same folder after installation should be used as the basis for our configuration. By default Doorman uses UDP port 1001, which we don't need to chance unless using this port for another application. Of course, by using a different port we can lower the chance of Doorman knocks being discovered by a potential intruder.

Doorman requires a password file */usr/local/etc/doorman/guestlist* to enumerate who can trigger the firewall to open for connections. We want two users to have access to SSH: martink and jacekz. The other fields in the *guestlist* file contain the password (stored in

Listing 7. Doorman output from the first knock attempt

```
Aug 23 23:26:29 asphyxia doormand[26673]:
  notice: Doorman V0.8 starting; listening on eth0 10.1.17.90
Aug 23 23:26:34 asphyxia doormand[26673]:
  info: knock from 10.1.17.1 :
  22 martink 1519966632 4270a3248a23c209b37d35bb060e1cd6
Aug 23 23:26:34 asphyxia doormand[26673]:
  debug: knock from 10.1.17.1 was valid.
Aug 23 23:26:34 asphyxia doormand[26675]:
  debug: open a secondary pcap:
  'tcp and dst port 22 and src 10.1.17.1 and dst 10.1.17.90'
Aug 23 23:26:34 asphyxia doormand[26675]:
  debug: run script:
  '/usr/local/etc/doormand/iptables_add eth0 10.1.17.1 0 10.1.17.90 22'
Aug 23 23:26:34 asphyxia doormand[26675]:
  debug: output from script: '0'
```

plain text), ports which the user can request to be opened and network address from which the user can connect (see Listing 6).

5.1 Gaining Access with Doorman

Having made the changes to *guestlist*, it's time to start Doorman. We'll keep it in the foreground using -D to see debugging messages.

```
$ /usr/local/sbin/doorman -D
```

Doorman is now ready to receive authentication packets on our server (10.1.17.90). We'll send the trigger packet from our client (10.1.17.1) using Doorman's knock client. For this example, we'll identify ourselves as *martink* and request that Doorman open port 22.

```
$ /usr/local/bin/knock -g martink -p 1001 -s hushhush 10.1.17.90 22
```

After sending the trigger packet, we'll see that *doormand* has produced output that indicates that the packet has been intercepted and and a door in the firewall has been opened to permit the client to connect to port 22. The debug output (see Listing 7) will show, that Doorman has used the helper script *iptables_add* to achieve this.

For an in-depth analysis of the trigger packet with tcpdump refer to the Inset *Doorman's trigger*. When Doorman intercepted this packet, it opened a door in our firewall. To see this, we can list the current rules with -L (see Listing 8).

Indeed, we can now connect from 10.1.17.1 to port 22. We are free to connect to the SSH

Listing 8. A door opened with Doorman

```
$ iptables -L
(...)
Chain INPUT (policy DROP)
target     prot opt source       destination
ACCEPT     tcp  --  10.1.17.1    10.1.17.90    tcp dpt:ssh
ACCEPT     all  --  anywhere     anywhere
ACCEPT     all  --  anywhere     anywhere      state RELATED,ESTABLISHED
DROP       udp  --  anywhere     anywhere
DROP       tcp  --  anywhere     anywhere      tcp flags:SYN,RST,ACK/SYN
(...)
```

Listing 9. Doorman detecting SYN packets and limiting port access

```
Aug 24 00:17:47 asphyxia doormand[27227]:
  debug: Initial SYN packet detected, martink@10.1.17.1:58360 -> 22
Aug 24 00:17:47 asphyxia doormand[27227]:
  debug: run script:
  '/usr/local/etc/doormand/iptables_add eth0 10.1.17.1 58360 10.1.17.90 22'
Aug 24 00:17:48 asphyxia doormand[27227]:
  debug: output from script: '0'
Aug 24 00:17:48 asphyxia doormand[27227]:
  debug: run script:
  '/usr/local/etc/doormand/iptables_delete eth0 10.1.17.1 0 10.1.17.90 22'
Aug 24 00:17:48 asphyxia doormand[27227]:
  debug: output from script: '0'
Aug 24 00:17:48 asphyxia doormand[27227]:
  info: connection established,  martink@10.1.17.1:58360 -> sshd(pid 27231)
```

Listing 10. Doorman-induced rule for the firewall, limiting SSH access to a selected source port

```
$ iptables -L
(...)
Chain INPUT (policy DROP)
target     prot opt source       destination
ACCEPT     tcp  --  10.1.17.1    10.1.17.90    tcp spt:58359 dpt:ssh
(...)
```

service on our server. Doorman will wait for 10 seconds (waitfor parameter in *doorman.cf*) before closing the firewall and returning it to its previous state. When we begin our SSH connection, Doorman will detect the SYN packet and refine the firewall rule to limit the client's source ports from which packets will be permitted. Doorman will accept further packets only from the same port from which the SYN packet was sent (see Listing 9). At this stage, the firewall contains the new refined rule (Listing 10). A couple of minutes later, when we terminate our connection by logging off the server, doorman spots this and deletes the refined firewall rule. The server now no longer accepts packets from our port 58360 (see Listing 11).

Listing 11. Doorman removes the door for SSH

```
Aug 24 00:19:52 asphyxia doormand[27227]:
  info: sshd(pid 27231) (martink@10.1.17.1:58360) has stopped running.
Aug 24 00:19:52 asphyxia doormand[27227]:
  debug: run script:
  '/usr/local/etc/doormand/iptables_delete eth0 10.1.17.1 58360 10.1.17.90
22'
```

5.2 Doorman's Trigger

During Doorman's authentication phase, we started a session of tcpdump on the server to independently monitor the trigger packets.

```
# tcpdump -XX ipv4 udp port 1001
```

Here is one such trigger packet intercepted by tcpdump.

```
23:26:34.564881 IP 10.1.17.1.57014 > 10.1.17.90.1001: UDP, length 54
        0x0000:  000c 29be cde8 0050 0468 1429 0800 4500  ..)....P.h.)..E.
        0x0010:  0052 ef07 4000 4011 1537 0a01 1101 0a01  .R..@.@..7......
        0x0020:  115a deb6 03e9 003e 003c 3232 206d 6172  .Z.....>.<22.mar
        0x0030:  7469 6e6b 2031 3531 3939 3636 3633 3220  tink.1519966632.
        0x0040:  3432 3730 6133 3234 3861 3233 6332 3039  4270a3248a23c209
        0x0050:  6233 3764 3335 6262 3036 3065 3163 6436  b37d35bb060e1cd6
```

The beginning of the packet is gibberish in ASCII – these are the Ethernet, IP and UDP headers (see Figures 1 and 2). Let's look at each of these headers in turn. The Ethernet header is the first, with the preamble and frame start delimiter removed. What we see are the two MAC addresses – of the server at 10.1.17.90 (destination MAC) and of the client at 10.1.17.1 (source MAC). The last 2 byte field (0x0800) indicates the Ethernet IP protocol:

```
000c 29be cde8 0050 0468 1429 0800
```

The next 20 bytes are the IP header:

```
                                        4500
0052 ef07 4000 4011 1537 0a01 1101 0a01
115a
```

The first byte is 4, which indicates the IP protocol. The next byte, 5, indicates the length of the packet in 32 bit words. The IP protocol we are using is UDP so byte 10 is 0x11 = 17 which is the value for UDP (TCP is 0x06). The last 8 bytes are the two IP addresses – first the client's IP address (0x0a01 0x1101 = 10.1.17.1) and the server's IP address (0x0a01 0x115a = 10.1.17.90).

After the IP header, the next 8 bytes are the UDP header:

```
deb6 03e9 003e 003c
```

with the first two bytes being the source port (0xdeb6 = 57,014) and the next two bytes the destination port (0x03e9 = 1001). The remaining parts of the UDP header are the length of the header and data (0x003e = 62 bytes) and checksum. The remaining part of tcpdump's output is the data payload of the packet:

```
22 martink 1519966632 4270a3248a23c209b37d35bb060e1cd6
```

Listing 12. An example .knockcf

```
# cat ~/.knockcf
grp martink
port 1001
secret hushhush
run "ssh %H%"
```

These are doorman's request and authentication tokens. First is the port we requested to open (22), followed by the user name, followed by a random value chosen by the knocker. The last value is the HMAC MD5 hash of our secret (*hushhush*) with the string *22 martink 1519966632* as the key. Doorman will not accept more than one trigger packet with the same random value, limiting the ability to replay a trigger.

Listing 13. Perl code to calculate a Doorman hash

```
#!/usr/bin/perl
# doormandigest [secret] [port] [user]
$,=" ";
my ($secret,$port,$user) = @ARGV;
use Digest::HMAC_MD5 qw(hmac_md5 hmac_md5_hex);
my $r = int rand() * (2**31-1);
my $key = "$port $user $r";
print $key,hmac_md5_hex($secret,$key),"\n";
```

5.3 Advanced Doorman

Instead of entering the user ID and Doorman's port at the command line when using *knock*, you can create *.knockcf* in your home directory that will store these values. You can include a command to be executed by *knock* 0.1 seconds after the trigger packet is sent. This is very useful when you're always connecting to the same service, like SSH,

and don't want to bother starting the SSH client yourself. You can also choose to store your secret, although this is in plain text, in this file. For protection, knock will not run unless the permission of *.knockcf* is 0600, so that others cannot see your secret. Listing 12 shows an example of *.knockcf*.

You can send a trigger packet manually, using SendIP, instead of *knock*. You will have to compute your own HMAC MD5 digest for a given combination of user, port, random number and secret. Listing 13 contains Perl code to do this, using the Digest::HMAC_MD5 module available from CPAN. The script, when invoked, prints a payload, which can then be used with SendIP (see Listing 14). The -d flag specifies the data payload and the -ud the destination port (Doorman's port).

With SendIP you have the ability to generate whatever packet you like, and adjust the header fields to your desire. You can make the packet look like it came from any IP, not just yours. Here we used -is 10.1.17.1, which is the actual IP address of the client, but any value is acceptable. Therefore, you can use SendIP from a third computer to ask Doorman to open a port for a different client.

Listing 14. Calculating a hash and using SendIP instead of Doorman's knock script

```
$ ./doormandigest hushhush 22 martink
22 martink 1035771909 bcc15f315ace1c34a7ed25a80b548594
$ sendip -p ipv4 -p tcp -is 10.1.17.1 -us 1002 -ud 1001 \
  -d "22 martink 1035771909 bcc15f315ace1c34a7ed25a80b548594" 10.1.17.90
```

6. Implementations and extensions

Port knocking describes a method of sending authentication information across closed ports for the purpose of remotely triggering events. Given that this covers a broad spectrum, it is not surprising that a large number of port knocking implementations exist.

In general, any port knocking client and server implementation will share certain characteristics. Fundamentally, the client and server must agree upon the form of the authentication. This could be the format of the port sequence, packet data payload, or both. Since validated authentication can trigger any event on the port knocking server, a map between such events and knocks is defined.

In some implementations all knocks are mapped onto either port opening or closing, and in others knocks can be associated with arbitrary system commands. Figure 4 shows a feature matrix for few popular implementations of port knocking (see *http://portknocking.org* for a complete list).

Two powerful means to combat replay attacks are challenge-response and one-time passwords. The challenge-response mechanism, in which the server asks the client to perform some computation on randomly selected input, is not appropriate for port knocking because the client authenticates passively. One-time passwords, however, can be implemented, and are supported by COK (cryptographic one-time knocks) by David Worth.

		authentication					access		
project	language	prot	header	payload	OTP	misc	firewall rules	arbitrary command	persistent state
cd00r	C	TCP	X			launches inetd			
SAd00r	C	UDP TCP ICMP1	X	X		final authentication packet stores encrypted system command		X	
COK	Java	UDP TCP	X	X	X2	DNS knocks	X	X	X
doorman	C	UDP		X	X		X		X
knockd	C	UDP TCP1	X				X	X	
pasmal	C	UDP TCP ICMP	X		X	intrusion detection; smoke screen; webadmin	X	X	
tumbler	Perl/Java	UDP		X			X	X	
author's implementation	Perl	TCP	X		X	variable length knock mapping to up to 32,768 closed ports; delayed responses	X	X	X
fwknop	C	TCP UDP ICMP	X	X		OS fingerprinting	X	X	

Figure 4. Feature matrix for popular port knocking implementations

Like Doorman's anti-replay strategy that incorporates a random number that can be used only once in the authentication string, COK runs a cryptographic hash function on the previous password to calculate the next password. Replay attacks are trivially detected – a spent OTP is detected. COK also supports knocking via DNS – the client makes a lookup request to the server's DNS daemon at OTP.domain.com where OTP is the one-time password. The DNS lookup fails, of course, but the request is logged and can be acted upon.

Pasmal by James Meehan has two interesting features to handle attempts at intrusion. One is an intrusion detection module which will prevent further connections from an IP when port scans or repeated failed knock attempts are detected. Another is a smoke screen feature, in which after the client has sent an initial authenticating knock, the remaining knock packets are interspersed among a large number of other packets set by the client. In this scheme, the client and server mutually decide which packets of the stream are part of the knock. Pasmal also features a web front-end for configuration.

Out of all the implementations, SAdoor takes a unique approach to how the client communicates to the server what command the server should execute. SAdoor encrypts the command in a command packet, which is the last packet of the authenticating packet sequence.

Many of the implementations use various aspects of packets for authentication, including header and data payload to store this information. fwknop implements an additional layer of authentication based on operating system fingerprinting. By examining a variety of TCP option values found in a TCP header, fwknop can match a combination of options to a specific operating system. Thus, the port knocking system can include the OS from which the packet was sent as a filter.

What makes a good knock?

Different implementations take a variety of approaches of how port knocking authentication should happen. Below is a list of desired characteristics of a knock.

Size – a knock should be as compact as possible. For knocks in the form of a packet sequence this is particularly important since fewer connection attempts mean fewer potential transmission errors and lower chance of your knock being intercepted and discovered. The knock could be mapped onto as large a port range as possible (e.g. 32,768) to maximize the information content for a given number of ports. For knocks composed of a single packet with a data payload, the practical limit is the average MTU (1500 bytes), which is much larger than the practical amount of information that can be encoded by a sequence of ports.

Encryption – the knock, or data within the knock, must be encrypted to limit tainting and replay attacks. In the very least, the secret used in the knock should be encrypted.

Checksum – for packet combination knocks, this can be an additional layer of validation. Once the port sequence is decoded and decrypted, if one of the values is reserved as a checksum, the contents of the knock can be verified. The checksum is an additional level of protection against forged knocks.

One-time knock (OTK) – for the truly paranoid, one-time knock can be easily implemented. If a knock is tagged with an index, the server need only keep track of which knock indexes it has received from a certain client. The client must increment the index for each new knock in order to be considered valid.

Means of transport – the knock can be either a sequence of packets, a single packet with authentication tokens in the payload, or both. Multi-packet knocks require very little effort to implement and subvert the distinction between headers and data payload (in this case the header is the data payload). Traditionally, if someone is sniffing traffic for passwords or other useful information, they are likely to ignore packets that lack a data component. On the other hand, when a knock is constructed in a single packet, transmission may be more reliable and the data encoding is more flexible.

7. Applications

There are a large number of ways in which port knocking can be used. In some cases, port knocking provides security features not found in other systems. One such application is the maintenance of near-line servers, which appear off-line (no open ports) but to which connections, mediated through a port knocking layer, is possible. Such servers could be placed on the Internet, or within a subnet on a local network, and could act as gateways to sensitive data.

Probably the most practical aspect of port knocking is to extend the time-to-patch without direct impact on security. By protecting a high-profile service like SSH, the administrator, who has likely more responsibilities than time, can check up on a server less frequently.

In addition to raising security, port knocking can be used in malware to provide undetectable backdoors (see Inset *On the Net*). A backdoor so protected would not be detectable by a scan of local ports and it is just a matter of time before examples of such applications appear in the wild.

8. Detection and attack

It is not possible to completely conceal a system protected by port knocking. Eventually, a packet must travel to or from the system, either as part of the authentication phase or a legitimate connection. Sniffers may therefore identify and detect your system by:

- detecting any outgoing traffic,

- detecting incoming knocks or

- detecting established connections.

Because of the variety of port knocking methods and the relative novelty of the method, anyone wishing to identify, intercept and possibly exploit a port knocking server will have to be extremely persistent in

intercepting and decoding traffic. Depending on the implementation, the window of opportunity to connect after a knock is validated is very narrow. Doorman, for example, immediately tightens the firewall rule after receiving the first SYN packet from the client. Therefore, for an attacker to target a system protected with port knocking it is more worthwhile to expend energies exploring standard attack vectors, such as session hijacking, then to attempt to punch a hole through a well-implemented port knocking system.

Because port knocking is an additional layer of passive security, bringing it down by overwhelming the port knocking daemon with a DoS attack does not leave the server exposed. If the port knocking layer is knocked out, the server will remain inaccessible to incoming traffic until the layer can be reset. While inconvenient, this is a desirable characteristic of a security layer. It should also be remembered that port knocking is meant to conceal and protect network services which should require their own authentication. It may be argued that a port knocking system protecting an SSH server does not require a challenge-response phase because the SSH server implements this. Thus, if the SSH service is kept reasonably up-to-date, the additional port knocking layer provides significantly more protection and eliminates thread from all but the most steadfast attackers.

9. Conclusions

Port knocking is still a relatively new method, with early adopters realizing an additional security bonus. Already, a large number of implementations exist and many are easy to install and configure.

Ultimately, incorporation of port knocking in hardware devices, such as initially SOHO routers which already support port forwarding and port triggering or small-footprint embedded devices dedicated to the task, would result in wider adoption. As we've shown, in its minimal form, port knocking is very easy to implement and you can increase the security of your system significantly with very little work. It's time to try port knocking and keep your ports open – for yourself.

10. About the author

Martin Krzywinski (http://mkweb.bcgsc.ca) is bioinformatics research scientist. He works with fingerprint maps of large genomes and loves to write Perl scripts of all sizes. He has a background in UNIX system administration and system automation. He is originally from Warsaw, but now lives in Vancouver, Canada where he kayaks and drinks a lot of espressos.

11. On the Net

http://www.portknocking.org – author's port knocking site,
http://doorman.sourceforge.net – the Doorman port knocking implementation,
http://www.phenoelit.de/stuff/cd00rdescr.html - cd00r
http://freshmeat.net/projects/pasmal/ - pasmal
http://www.symantec.com/press/2004/n040920b.html – the Emerging trends section of this article speaks of backdoors with port knocking,

http://www.usenix.org/publications/login/1998-8/tcpdump.html - Tcpdump : the spanner wrench of network monitoring.
http://www.networksorcery.com/enp/topic/ipsuite.htm - headers for IP protocols