MICHAEL RASH

combining port knocking and passive OS fingerprinting with fwknop

Michael Rash holds a master's degree in applied mathematics and works as a security research engineer for Enterasys Networks, Inc. He is the lead developer of the cipherdyne.org suite of open source security tools, including PSAD and FWSnort, and is co-author of the book Snort-2.1 Intrusion Detection published by Syngress.

mbr@cipherdyne.org

IT WAS AROUND 2:45 A.M. ONE SUMMER night in 2002 and I had finally finished. My shiny new Linux system was fully installed and connected to my broadband cable modem connection in my apartment. All unnecessary services had been turned off, a restrictive iptables policy had been deployed, and a tripwire filesystem snapshot had been taken, all before connecting the system to the network.

Reasoning that the only servers I needed to have accessible from arbitrary IP addresses around the Net were Apache and OpenSSH, iptables could be configured to log and drop almost all connection attempts. After connecting the system to the network and scanning it from a shell account on a different external network, I saw that only TCP ports 22, 80, and 443 were accessible, so I was satisfied that my system was fit to remain connected.

It was late, though, and I forgot one important detail. I neglected to check the version of OpenSSH that came bundled with the Linux distribution. Back then RedHat 7.3 was my Linux distribution of choice even though more recent versions of RedHat (and other Linux distributions) were available. After getting some well-deserved sleep, I woke the next morning only to find that, sure enough, my system had been compromised. Luckily, I had no important data on the box yet, but it could have been a lot worse.

It became clear that in addition to upgrading to the latest version of sshd, it would also be desirable to protect sshd as much as possible with iptables. Yet at the same time, the ability to log in remotely and administer the system from anywhere was highly desirable. Unfortunately, these two goals are fundamentally at odds. Sure, sshd does not allow just anyone to log in or execute commands; users must have the proper authentication credentials for at least one type of authentication method supported by sshd (username/password, RSA, Kerberos, etc.), but all of this may not help if there is a buffer overflow vulnerability (as in my case) buried within a section of the sshd code that is accessible over the network.

An attacker may only need the capability of connecting to sshd in order to be in a position to exploit such a vulnerability. Being able to connect means the attacker can send packets up through the server's IP stack, establish a TCP session with the transport layer, and, finally, talk directly to sshd. Alternatively, if iptables does not allow the attacker's IP to connect to sshd, then any packets sent from the attacker are blocked by iptables before they even make it into the IP stack in the Linux kernel, let alone to the SSH daemon itself.

Clearly, the most desirable way to protect an arbitrary service is with iptables. However, since not all IP addresses that should be allowed to connect to sshd can be enumerated a priori, I would need to add an additional authentication layer to iptables. Port knocking provides a simple but effective solution to this problem.

Port Knocking

Port knocking[1] provides a method of encoding information within sequences of connection attempts to closed (or open) ports. The most common application of such information (which can include IP addresses, port numbers, protocols, usernames, etc.) is the modification of firewall policies or router ACLs in response to monitored valid port knock sequences. In essence, port knocking provides a means of network authentication that only requires the ability to send packets that contain transport layer headers.

Knock clients do not need to actually talk to a server-side application or even have a TCP session established; the knock server can behave completely passively as far as network traffic is concerned. On the server side, knock sequences can either be monitored via firewall log messages or with a packet capture library such as libpcap, in the same way an IDS watches traffic on a network. Although using a packet capture library would provide the ability to encode information such as a password at the application layer, a full-featured port knock implementation is completely feasible without using any packet data above the transport layer—hence firewall logs are ideally suited for this application. Implementing a port knocking scheme around firewall logs has the added bonus of helping to ensure the firewall is configured correctly, or at least that it is logging packets.

Protecting Against Replay Attacks

It should be noted that many port knocking techniques are susceptible to replay attacks if an attacker is in a position privileged enough to be able to sniff traffic between the port knock client and server. An attacker need only replicate a knock sequence for the server to grant the same level of access that would be granted to a legitimate client. Hence some would argue that port knocking suffers from the standard arguments against "security through obscurity." However, port knocking is not designed to act as the only security mechanism for secure communications; encryption implemented by sshd serves as the main line of defense. Port knocking provides an additional layer of security on top of the secure communications already implemented by sshd. The argument against "security through obscurity" is only valid if security is *completely* dependent on obscurity.[2] In addition, there are several techniques for raising the bar for the attacker even if the entire sequence has been observed on the wire. Let us examine four such techniques:

1. Relative timings between sequence packets can be made significant. For example, the knock server may require that the minimum time delay between successive knock sequence packets is at least three seconds, but not longer than six seconds.
2. Multiple protocols (TCP, UDP, and ICMP3) can be used within the knock sequence. If an attacker has restricted the view of a sniffer to just, say, the TCP protocol, then some portion of such a sequence will be missed and hence cannot be replayed on the network.

3. Encryption can be used. Due to the fact that an IP address, a protocol number, and a port number together only require seven bytes of information to represent, it is easy to use a symmetric block cipher (such as the Rijndael algorithm) to encrypt port knock sequences before they are sent across a network. Encrypting an IP address within a knock sequence allows a knock client to instruct a knock server to allow access for a third-party IP address that cannot be guessed by anyone observing the knock sequence. As usual, use of a symmetric encryption algorithm requires a shared key that is known to both the knock client and the knock server. There are also fancier methods of using encryption, such as one-time passwords,4 that genuinely make replay attacks infeasible.
4. Requirements can be made on the type of operating system that generates a knock sequence. Additional fields in the IP and TCP headers—TTL values, fragment bits, overall packet size, TCP options, TCP window size, etc.—can be made significant. If a knock sequence is monitored between a client and server, then any duplicated sequence will not be honored by the server unless the OS of the duplicate sequence exactly matches that of the original client. For example, if a knock sequence between two Linux machines is sniffed off the wire and an attacker replays the sequence from a MacOS X machine, the duplicate sequence will be ignored. Of course, OS characteristics can be spoofed by the attacker, but this may not be worth the trouble (again, although this is not unbreakable, port knocking adds an additional layer of security).

Fwknop

This article discusses a tool called fwknop (Firewall Knock Operator), which supports both shared and encrypted port knock sequences along with all four of the obfuscation techniques mentioned above. fwknop exclusively uses iptables log messages to monitor both shared and encrypted knock sequences instead of appealing to a packet capture library. In addition, due to the completeness of the iptables logging format, fwknop is able to passively fingerprint operating systems from which connection attempts originate. fwknop uses this capability to add an additional layer of security on top of the standard knock sequences by requiring that the TCP stack that generates a knock sequence conform to a specific OS. This makes it possible to allow, say, only Linux systems to issue a valid knock sequence against the fwknop knock server. I develop and release fwknop as free and open source software under the GNU Public License (GPL); fwknop can be downloaded from http://www.cipherdyne.org/fwknop/.

IMPLEMENTATION

Firewall logs, especially those created by iptables, can provide a wealth of information about port scans, connection attempts to back door, DDoS programs, and attempts by automated worms to establish connections to vulnerable software. One of the most important characteristics of firewall logs is that packets can be logged completely passively; the firewall is under no obligation to allow the target TCP/IP stack to generate any return traffic in response to a TCP connection attempt. Yet, at the same time, all sorts of juicy bits of information can be logged from a connection attempt, such as TTL and IP ID values, source and destination port numbers, TCP flags, TCP options, and more. (Note that UDP and ICMP packets will generate iptables log messages that contain information appropriate to those protocols.)

fwknop parses iptables log messages that are sent to syslog as iptables intercepts packets that traverse the firewall interfaces. By default, iptables logs packets via the syslog `kern` facility at a priority of `info`. Such messages are usually sent to the file /var/log/messages, but fwknop reconfigures syslog to also send kern.info messages to a named pipe, where they are read by fwknop. Let us examine an iptables log message generated by the following iptables rule:

```
iptables -A INPUT -p tcp -i eth0 -j LOG —log-tcp-options
```

A TCP syn packet to port 60000 on the eth0 interface will result in the following log message logged via syslog to /var/log/messages:

```
Aug  7 17:22:57 orthanc kernel: IN=eth0 OUT=
MAC=00:0c:41:24:68:ef:00:0c:41:24:56:37:08:00 SRC=192.168.10.2 DST=10.3.2.1 LEN=60
TOS=0x10 PREC=0x00 TTL=64 ID=56686 DF PROTO=TCP SPT=32811 DPT=60000 WINDOW=5840 RES=0x00
SYN URGP=0 OPT (020405B40402080A06551B7A0000000001030300)
```

Iptables does a good job of decoding packet information before sending it to syslog. Clearly displayed (among other things) are source and destination IP addresses, packet length and TTL values, source and destination ports, TCP window size, and TCP flags. The TCP options portion of the TCP header is also visible, but because decoding it would place an undue burden on the kernel, only the raw options data is logged. In an effort to passively fingerprint the operating system that generated the above log message, fwknop uses a strategy similar to p0f,[5] which is one of the best passive OS fingerprinters available. Since matching a p0f signature against the packet above requires the examination of specific TCP option values, fwknop must decode the options string. A quick examination of RFC 793 informs us that there are two formats for TCP options: three 8-bit-wide fields denoting the option type, length, and value, or a single 8-bit-wide field denoting the option type. Interpreting these two formats along with the appropriate TCP option definitions with an eye toward what is required by p0f, fwknop decodes the options string in the packet above,

`020405B40402080A06551B7A0000000001030300`,

as the following:

```
- Maximum segment size = 5840  - Selective acknowledgment is permitted  - The timestamp
is set  - NOP  - Window scale = 0
```

Hence, the packet log message above is matched by the following p0f signature:

```
S4:64:1:60:M*,S,T,N,W0     Linux:2.4::Linux 2.4/2.6
```

Now let's turn to some concrete port knocking examples. The following two knock sequence examples will involve the execution of fwknop from the command line in client mode from the source IP 192.168.10.2 to the destination machine 10.3.2.1, where fwknop is running in server mode. (RFC 1918 addresses were chosen for illustration purposes so as not to step on the toes of any real networks out there.) In both sequence examples iptables is configured to block access to sshd on the knock server, but after receiving a valid port knock sequence, fwknop will reconfigure iptables to allow access to sshd. In order to indicate clearly how access is modified, connection attempts to sshd on the knock server will be made from the knock client system before and after sending the knock sequences. As fwknop receives and parses knock sequences and modifies access controls, it writes information to syslog, and these messages will also be displayed below. All command-line invocations of fwknop below take place on the client system.

SHARED SEQUENCE

First let's examine a shared sequence that involves multiple protocols. Fwknop supports the use of TCP, UDP, and ICMP echo requests within shared knock sequences. Shared sequences must be defined in two places: the file ~/.fwknoprc on the client system, and the file /etc/fwknop/access.conf on the server system. Hence, our first knock sequence is defined as follows on the server:

```
[server]# cat /etc/fwknop/access.conf
SOURCE: ANY;
SHARED_SEQUENCE: tcp/50053, udp/6020, icmp, icmp,
tcp/24034, udp/9680;
OPEN_PORTS: tcp/22;
FW_ACCESS_TIMEOUT: 30;
REQUIRE_OS_REGEX: linux;
```

The `SOURCE` keyword defines from which IP address or network a knock sequence will be accepted (with the special value `ANY` accepting knock sequences from any source IP). The `SHARED_SEQUENCE` keyword defines the specific port numbers and protocols that constitute a valid sequence. The `OPEN_PORTS` keyword defines the set of ports and corresponding protocols to which the source address should be allowed to connect. Fwknop will reconfigure iptables on the underlying Linux system only upon receiving a valid knock sequence. The `FW_ACCESS_TIMEOUT` specifies the length of time (in seconds) the underlying iptables policy will be configured to accept connections from an IP address that has issued a valid knock sequence. The `REQUIRE_OS_REGEX` variable instructs fwknop to accept a knock sequence if and only if the p0f signature derived from the originating operating system contains the specified string (the match is performed case-insensitively).

In the file ~/.fwknoprc on the client system, a similar block of text defines the same sequence for the fwknop client. Note the specific IP address of the fwknop server is listed immediately preceding the sequence definition:

```
[client]$ cat ~/.fwknoprc
10.3.2.1: tcp/50053, udp/6020, icmp, icmp, tcp/24034,
udp/9680
```

Now for the actual execution of fwknop. First, connectivity to sshd is tested from the client, then the port knock sequence is sent across the network to the server, and, finally, an additional connection attempt shows that access has indeed been granted:

```
[client]$ telnet 10.3.2.1 22
Trying 10.3.2.1...
[client]$ fwknop -k 10.3.2.1
[+] Sending port knocking sequence to knock server:
10.3.2.1
[+] tcp/50053 -> 10.3.2.1
[+] udp/6020  -> 10.3.2.1
[+] icmp echo request -> 10.3.2.1
[+] icmp echo request -> 10.3.2.1
[+] tcp/24034 -> 10.3.2.1
[+] udp/9680  -> 10.3.2.1
[+] Finished knock sequence.
[client]$ telnet 10.3.2.1 22
Trying 10.3.2.1...
Connected to 10.3.2.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.8.1p1
```

On the server the following messages are written to syslog by fwknop as it monitors the port knock sequence in the iptables log:

```
Aug 8 13:17:46 orthanc fwknop: port knock access sequence matched for 192.168.10.2
Aug  8 13:17:46 orthanc fwknop: OS guess: Linux:2.4::Linux 2.4/2.6 matched for
192.168.10.2
Aug  8 13:17:46 orthanc fwknop: adding INPUT ACCEPT rule for source: 192.168.10.2 to con-
nect to tcp/22
Aug  8 13:18:18 orthanc fwknop: removed iptables INPUT ACCEPT rule for 192.168.10.2 to
tcp/22, 30 second timeout exceeded
```

The log shows that the fwknop server added a rule in the iptables INPUT chain for a total of 30 seconds to accept connections from 192.168.10.2 over tcp/22. Although the 30-second timeout seems a bit short, if the iptables policy on the underlying system is written so that packets that are part of established sessions are accepted first before remaining packets are dropped, then any SSH session that was established within the 30-second window will not be killed when the ACCEPT rule is removed. Note that the port number for which the fwknop server permitted access never appears in the knock sequence itself; it is defined in /etc/fwknop/access.conf, so the client has to know to which port(s) it has access after sending the sequence. This characteristic holds true for all shared sequences.

ENCRYPTED SEQUENCE

Now let's take a look at an encrypted knock sequence. This time the sequence itself will change depending on the key used to encrypt the source IP address, protocol, port number, and local username that fwknop is being executed as. Thus, encrypted sequences are not defined within any configuration file on the server or client systems. Sequences are monitored on the server; if successfully decrypted then such a sequence is valid and access controls will be modified. The fwknop server must still be configured with the appropriate encryption key and port(s) to open, and as usual this information is contained in the /etc/fwknop/access.conf file on the fwknop server:

```
[server]# cat /etc/fwknop/access.conf
SOURCE: ANY;
ENCRYPT_SEQUENCE;
KEY: 3ncryptk3y;
OPEN_PORTS: tcp/22;
FW_ACCESS_TIMEOUT: 30;
REQUIRE_OS_REGEX: linux;
```

The SOURCE, OPEN_PORTS, FW_ACCESS_TIMEOUT, and REQUIRE_OS _REGEX keywords are used as before, but two additional keywords, ENCRYPT_SEQUENCE and KEY, are defined to instruct fwknop to accept a port knock sequence encrypted with the subsequent key. Now for our encrypted port knocking example:

```
[client]$ telnet 10.3.2.1 22
Trying 10.3.2.1 . . .
[client]$ fwknop -e -a 192.168.10.2 -P tcp -p 22 -r -k
10.3.2.1
[+] Enter an encryption key (must be as least 8 chars, but
less than 16
chars). This key must match the key in the file
/etc/fwknop/access.conf
on the remote system.
[+] Encryption Key:
[+] clear text sequence: 192 168 10 2 0 22 6 28 109 98 114
0 0 0 0
[+] cipher text sequence: 182 246 253 35 195 76 157 229 86
13 152 30 120 172 58 140
[+] Sending port knocking sequence to knock server:
10.3.2.1
[+] tcp/61182 -> 10.3.2.1
[+] udp/61246 -> 10.3.2.1
[+] tcp/61253 -> 10.3.2.1
[+] udp/61035 -> 10.3.2.1
[+] tcp/61195 -> 10.3.2.1
 . . .
```

```
[+] Finished knock sequence.
[client]$ telnet 10.3.2.1 22
Trying 10.3.2.1 . . .
Connected to 10.3.2.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.8.1p1
```

On the server the following messages are written to syslog by fwknop:

```
Aug  8 13:00:28 orthanc fwknop: decrypting knock sequence for 192.168.10.2
Aug  8 13:00:28 orthanc fwknop: OS guess: Linux:2.4::Linux 2.4/2.6 matched for
192.168.10.2
Aug  8 13:00:28 orthanc fwknop: username mbr match
Aug  8 13:00:28 orthanc fwknop: adding INPUT ACCEPT rule for source: 192.168.10.2 to con-
nect to tcp/22
Aug  8 13:01:00 orthanc fwknop: removed iptables INPUT ACCEPT rule for 192.168.10.2 to
tcp/22, 30 second timeout exceeded
```

Note that fwknop has allowed the real source through the firewall since this is the source address that was encrypted in the sequence with the `-a` option. Any other third-party IP address could have been specified here. Also notice that as with the previous shared sequence, fwknop passively fingerprinted the client operating system and the required Linux OS was found. Finally, note that the encrypted sequence is rotated through the TCP and UDP protocols. More information about fwknop, including a detailed description of all configuration directives, can be found at http://www.cipherdyne.org/fwknop/.

Conclusion

Port knocking adds an additional layer of security for arbitrary services that are accessible over a network. A client system must send a specific sequence of connection attempts to the knock server before access is granted to any protected service through a firewall or other access control device. Port knocking is useful for enhancing security because anyone who casually scans the target system will not be able to tell that there is any server listening on the ports protected by the knock server. Port knocking is not designed to provide bullet-proof security, and, indeed, replay attacks can easily be leveraged against a port knock server in an effort to masquerade as a legitimate client. However, there are several techniques for obfuscating port knock sequences through timing requirements, multiple protocols, passive fingerprinting of knock client operating systems, and encryption in order to make knock sequences more resistant to replay attacks. Fwknop is a complete port knocking implementation based around iptables, and supports multi-protocol knock sequences (shared or encrypted) along with passive OS fingerprints derived from p0f.

REFERENCES

1. M. Krzywinski, "Port Knocking: Network Authentication Across Closed Ports," *SysAdmin Magazine*, vol. 12, no. 6 (June 2003).

2. J. Beale, "'Security Through Obscurity' Ain't What They Think It Is" (2000): http://www.bastille-linux.org.

3. ICMP is implemented strictly as a network layer protocol and hence has no concept of a transport layer port number. However, the mere presence of ICMP echo requests can be made significant in terms of what a knock server expects to see, and thus adds an additional dimension to a port knock sequence.

4. See David Worth, "Cryptographic One-Time Knocking: Port Knocking Done Better": http://www.hexi-dump.org/bytes.html.

5. The original p0f was developed by Michal Zalewski and is available for download from http://lcamtuf.coredump.cx/p0f.shtml.