Single Packet Authorization

Michael Rash

Abstract

Single Packet Authorization fills the gaps in port knocking.

Countless pieces of software, protocols and complex interdependencies together form a system for which it is difficult to guarantee any particular property—particularly security. Even software specifically designed to enhance security can, at the behest of clever individuals armed with detailed knowledge, work to its detriment. Vulnerabilities have been discovered in all sorts of security software from firewalls to implementations of the Secure Shell (SSH) Protocol. For example, OpenSSH is developed by some of the most security-conscious developers in the world, and yet it occasionally contains a remotely exploitable vulnerability. This is an important fact to note because it seems to indicate that security is hard to achieve and, therefore, bolsters the case for a defense-in-depth approach. This article explores the concept of Single Packet Authorization (SPA) as a next-generation passive authentication technology beyond port knocking.

When an attacker is on the prowl in an attempt to exploit a vulnerability in server software (as opposed to client software), the first step is reconnaissance; the attacker needs to locate a target. This process has been brilliantly automated by Nmap, so it is easy to construct a list of target systems that may be ripe for compromise. If the attacker has found a zero-day vulnerability in server software that you happen to be running, you don't want to appear in this list of targets! Both port knocking and Single Packet Authorization use a packet filter configured in a default-drop stance and simultaneously provide service only to those IP addresses that can prove their identity via a passive mechanism. No TCP/IP stack access is required to authenticate remote IP addresses via this passive means. Nmap cannot even tell that a server is running when protected in this way, and it does not matter even if the attacker has a zero-day exploit.

This article is the first of a two-part series on Single Packet Authorization, and it lays the theoretical foundation for Single Packet Authorization and why it is a next-generation passive authorization technology beyond port knocking. The next article will provide a hands-on look at using fwknop to provide Single Packet Authorization protection for your SSH daemon.

Introduction to Port Knocking

Port knocking is a first-generation technology that uses the port fields within TCP and UDP packet headers to communicate information. Normally, these protocols are used to encapsulate application layer data, but port knocking encodes information in sequences of packets to various ports by using the port numbers themselves as fields to transmit data. These packets are typically either monitored out of a firewall log or via a packet capture mechanism, such as libpcap. Typically, there is a port knocking client and a port knocking server. The terms client and server, in this case (and throughout the remainder of this article unless otherwise noted), refer to the software components that send and monitor packets,
respectively. The client is responsible for generating the port sequences, and the server is responsible for passively collecting the sequences and reconfiguring the packet filter to allow connections to protected services upon receipt of a valid sequence.

The typical port knocking scenario is for a port knocking server to configure a packet filter to block all access to a service, such as SSH, until a specific port knock sequence is sent by a port knocking client. For example, the server could require the client to send TCP SYN packets to the following ports in order:

- 23400
- 1001
- 2003
- 65501

If the server monitors this knock sequence, the packet filter reconfigures to allow an SSH connection from the IP address that sent it. By making use of a connection tracking mechanism provided by the packet filter (such as the conntrack system in Netfilter), an SSH session can remain established after the initial rule created by the knock server is removed after a timeout. Port knock sequences can be encrypted, and there are many implementations listed at [http://www.portknocking.org](http://www.portknocking.org). For a graphical representation of port knocking in action, see Figure 1.

![Figure 1. Port Knocking in Action](image)

**Port Knocking Limitations**

Port knocking offers some real benefits for limiting access to services, but where do the limitations lurk? First, it is clear that encrypting knock sequences is important, and this in turn implies that several bytes of information must be transmitted. For symmetric crypto systems, the encrypted data will be at least as large as the block size (128 bits for the Rijndael symmetric block cipher chosen for the Advanced Encryption Standard). For asymmetric crypto systems, the encrypted data will be substantially larger.

For instance, the raw ElGamal algorithm used by GnuPG doubles the plain-text size when encrypting data. Even though GnuPG also utilizes compression (which can sometimes reduce the size of the cipher text to below the original size of the plain text), the typically large key size of GnuPG keys implies that the cipher text for even the smallest messages will be in the hundreds of bytes.

This has important implications for port knocking. Each packet within a port knock sequence can send only two bytes of information due to the 16-bit-wide port fields in the TCP and UDP headers. (This assumes that other fields within packet headers are not also used to transmit data. However, even if other
fields are used, this still cannot result in nearly as much data transmission as using packet payloads.) Hence, for a block cipher, an encrypted sequence must contain at least \( \frac{B}{2*8} \) packets, where \( B \) is the block size in bits. This by itself would not be so bad when considering the general speed and reliability of today's networks, but the real issue is out-of-order delivery.

Decrypting garbled data results in garbled data, and because there is no notion of a “connection” (in the TCP sense) between the port knock client and server, the server has no ability to re-order out-of-order packets.

Packets may take different routing paths, some of which may be slow. Hence, the client must resort to an artificial mechanism to try to reduce the potential for out-of-order delivery: time. By introducing a time delay between each packet in a knock sequence, say on the order of a half second, packet order usually can be maintained by the time the packets reach the server. Now, for a block size of 128 bits, the corresponding port knock sequence is \( \frac{128}{2*8} = 8 \) packets. By factoring in the half-second delay, it takes four seconds just to transmit the sequence. For a much larger message, such as those that would be generated by an asymmetric cipher, this data transmission rate is simply not practical.

Having a limited ability to transmit data introduces another limitation in port knocking schemes. It is difficult to guard against a replay attack effectively. Anyone who can monitor a knock sequence as it is sent from the client to the server is free to replay the sequence against the server in an effort to gain the same access. This is an especially important issue if the sequence is sent through a NAT device, and the source IP that is allowed through the packet filter at the server side is the external NAT address. For example, if the port knock client is on an RFC 1918 subnet, say 10.10.1.0/24, and the port knock server is on a remote network that is accessible only over the open Internet, the server must allow access to the NAT IP address. Anyone on the same subnet who can replay the sequence will be granted the same level of access. Also, anyone on the same subnet has the same level of access once a rule is instantiated to accept connections from the NAT address as long as the rule exists (no sequence replay is required in this case, and this remains true for SPA as well).

There have been variations made on traditional port knocking to try to provide a solution for the replay problem, such as making time a significant factor, using S/Key-style hash function iteration and even simply changing the encryption key after each use. However, each of these methods requires some state to be maintained by both the port knock client and server and does not scale very well when multiple users become involved.

An additional port knocking limitation is that it is extremely easy for a malicious third party to bust a knock sequence just by spoofing an additional packet into the port sequence as it is sent over the wire by the client. The attacker would simply set the source address on the packet to be the same as that of the real client and choose the same port number as the last packet sent by the client. This extra packet would break the knock sequence, so the server would not allow the legitimate client any additional access. Although the chances that people would actually do this are relatively small (they still need to be able to monitor packets emanating from the client), the main issue is that such an attack is so trivially easy to perform. A single packet is all that is required, and the attacker doesn't even need to be inline to the original packet data path.

Finally, knock sequences are easily detectable as port scans by any intrusion detection system (IDS) that is able to monitor traffic between the client and server. This is particularly true for encrypted knock sequences, which tend to be longer than simple shared sequences. To an IDS, port knocking looks just like a series of probes to various ports from a single IP address within a relatively short period of time,
and this fits the definition of a port scan quite nicely.

**Single Packet Authorization**

The end result of the above discussion is that port knocking provides some real benefits that enhance security, but some serious limitations also need to be addressed. Single Packet Authorization is a relatively new protocol that retains all of the benefits of port knocking, but fixes the limitations discussed above. The first publicly available SPA implementation was released in May 2005 as a piece of software called fwknop ([http://www.cipherdyne.org/fwknop](http://www.cipherdyne.org/fwknop)). fwknop was originally created in 2004 as the first port knocking implementation to combine passive OS fingerprinting and port knocking (this made it possible to do things like “accept knock sequences only from Linux-2.4 systems”), but the SPA method is now the most popular (and default) authentication method offered by fwknop. Note that fwknop provides both authentication and authorization services, but a full discussion of the difference between the two is beyond the scope of this article.

Single Packet Authorization mandates a similar architecture to port knocking. Both have client and server components, the server maintains control of a default-drop packet filter, and the server monitors packets passively. However, this is where the architectural similarities between port knocking and SPA diverge.

Single Packet Authorization moves the data transmission to where it belongs—in the application layer. This implies that instead of being able to send only two bytes of data per packet, as in the case of port knocking, SPA is able to send up to the minimum MTU worth of data (1,500 bytes on Ethernet networks) between the client and the server in each packet. This far outstrips the data transmission rate possible with port knocking, and having easy access to this amount of packet data opens up a huge range of possibilities. The remainder of this article discusses Single Packet Authorization as implemented by fwknop.

fwknop defines the following packet format at the application layer:

- 16 bytes of random data
- Client username
- Client timestamp
- fwknop version
- Mode (access or command)
- Access (or command string)
- MD5 sum

Many of the fields in the SPA packet format have a variable length, but are separated by a : character (fields are base64-encoded, so embedded colons cannot break this syntax). Once the fwknop client builds the packet format above, the entire packet is encrypted using one of two encryption algorithms: the Rijndael symmetric block cipher with a 128-bit shared key or the asymmetric ElGamal algorithm with up to a 2,048-bit public/private key pair generated by GnuPG. By default, the fwknop client sends...
SPA packets over UDP port 62201, but this easily can be changed from the command line; see the --Server-port argument. (fwknop offers many configuration options—see Resources for a link to the documentation and man pages.) For a graphical representation of SPA in action, see Figure 2.

![Figure 2. SPA in Action](image)

So, what are all the fields for? First, the 16 bytes of random data allows one of the highest priority limitations in port knocking to be solved—the replay problem. Every SPA packet is prepended with 16 bytes of random data before being encrypted, and then upon a successful decrypt by the fwknop server, the MD5 sum of the entire packet is cached. The random data allows every SPA packet to be different (even when the same access directive is sent), so the MD5 sum of every packet also has a high probability of being different. If the MD5 sum of any new packet matches the sum of a previous packet, the fwknop server takes no action and writes a warning message to syslog. Hence, any SPA packet that is intercepted by a third party cannot be replayed on the network in an effort to get access through the default-drop packet filter.

The client username and timestamp are placed within the packet by fwknop and the username is used to maintain different authorization levels for remote users by the fwknop server. fwknop can be installed on a multiuser system, and each user can be authorized to connect to different services by a remote fwknop server. The fwknop version field is used to maintain backward compatibility. Fields can be added or deleted in new releases of fwknop, but by using the version number, the fwknop server can remain compatible with the manner in which older clients build SPA packets. The mode field tells the fwknop server whether the client wants to access a service or execute a command (with the specific access control directive or command in the next field). For example, to gain access to TCP port 22, the Access field would contain the string `<IP>,tcp/22` where `<IP>` is whatever IP address the client chose to put in the packet. Finally, the MD5 sum field contains the MD5 sum of the unencrypted packet before the client transmits it. This is used by the server to verify message integrity after decryption.

We already have seen how the increased amount of data that can be transmitted via an SPA packet has solved the replay problem and the extremely low data transmission rate in port knocking schemes. We have two remaining limitations in port knocking that need to be addressed. First, the single packet nature of the SPA protocol means that a malicious third party cannot break the authentication scheme just by spoofing a packet to the same port over which a monitored SPA packet is sent. Finally, because the SPA protocol requires only a single packet, it does not appear to any intermediate IDS like a port scan. All that any IDS can see is an unintelligible blob of data seemingly spuriously sent to some IP address.

**Conclusion**

Single Packet Authorization provides similar security benefits to port knocking in terms of protecting services with a packet filter that is configured in a default-drop stance. Anyone scanning for a target service that is protected in this way will be unable to detect such a service is listening, and this makes
even the exploitation of zero-day vulnerabilities much more difficult. SPA offers elegant solutions to many limitations in port knocking implementations. These allow SPA to solve the replay problem, achieve a data transmission rate that makes the use of asymmetric encryption possible, thwart simple spoofing attacks and remain under the radar of intrusion detection systems that are monitoring networks for port scans.

See next month’s *LJ* for Part II to this article, which will show exactly how to use SPA.

**Resources**


There is only one other SPA implementation that I am aware of at the time of this writing, available at [http://www.unspecific.com/spa](http://www.unspecific.com/spa).

Another implementation called Tumbler ([http://tumbler.sourceforge.net](http://tumbler.sourceforge.net)) employs a single packet, but it uses a hashed payload instead of an encrypted payload, and this results in a significantly different architecture.

fwknop documentation and man pages: [http://www.cipherdyne.org/fwknop/docs](http://www.cipherdyne.org/fwknop/docs)