Port Knocking: Beyond the Basics

Dawn Isabel

Port Knocking: Beyond the Basics

Dawn Isabel
GIAC Security Essentials Certification (GSEC)
Practical Assignment Version 1.4c
Option 1 – Research on Topics in Information Security
Submitted March 9, 2005

Abstract

Port knocking has recently become a popular and controversial topic in security. A basic overview of port knocking is given, and it is assumed that when carefully implemented, port knocking can be a useful tool in some situations. Two problems with static port knocking - detection and replay - are described, and three solutions are proposed: covert knocks, dynamic knocks, and one-time knocks. Four current implementations of port knocking are analyzed to demonstrate these solutions. An implementation using Net::Pcap generates static covert knocks over DNS. Cerberus encapsulates knocks in ICMP ping packets and uses one-time passwords for authentication. The SIG^2 Port Knocking Project implements dynamic knocks that are generated randomly and as needed. CÖK implements the One-Time Password specification to send one-time knocks over UDP. The improvements of these implementations over static port knocking mitigate some threats, but several concerns still exist. Implementations aimed at the enterprise environment will need to address additional needs. In conclusion, port knocking deserves future consideration and can be a valuable layer in defense-in-depth.

Overview of Port Knocking

There are a variety of definitions available for port knocking. Perhaps the broadest - and least implementation-specific - is expressed as "a method for delivery of information via closed ports on a networked computer" (Maddock, p.1). The concept of "closed ports" is key - in port knocking, information is passed to the recipient without establishing a direct connection to the recipient over an open port. This information - the "knock" - is used to trigger an event on the server. The goal of communicating over closed ports is simple - if the ports are closed, it is significantly more difficult to detect that any services are being offered.

Consider the following example: a firewall server has the option of remote administration using SSH on TCP port 22. The firewall's administrator only sporadically needs to use the service, and does not want to draw the scrutiny of attackers by keeping port 22 open all the time. The administrator sets up a script to monitor incoming TCP SYN packets for a secret knock - when the script sees connection attempts to the port sequence [333, 555, 222, 777] the script

will open port 22 on the firewall. Another sequence is defined to close the port. As a result, the administrator can keep the port closed until needed, and can open it from an arbitrary remote location.

Although port knocking has been discussed a great deal as of late[1], it is not a new concept. A well-known proof-of-concept, cd00r, was written in 2000 (FX). However, the mainstream security community has been slower to take port knocking seriously. Even now, debates rage - is port knocking a novel new layer in defense-in-depth? Is it simply another means of security through obscurity? This paper will not seek to put this debate to rest. Instead, it will start with a few assumptions:

1. **Port knocking will add value to some security postures.** While port knocking is undoubtedly not a panacea, there are situations that will benefit from the extra layer of security. For instance, the need for sporadic access to a service from an arbitrary location may preclude static IP filtering as a defense, but port knocking might fit the bill.
2. **Port knocking is one of many layers.** A basic assumption is that the event that is triggered by port knocking is appropriately secured. In the case of a service that is opened, the service should be properly patched, hardened, and logged.
3. **Port knocking can be more than a layer of obscurity.** The obscurity argument is often advanced against port knocking for two reasons. First, the stealth aspect of communicating over closed ports is often the main focus of port knocking discussions, overshadowing implementation details. This encourages the notion that the singular goal of port knocking is stealth, which in fact is not always true. Secondly, this argument is often applied to port knocking implementations where the knock is a static sequence that must be shared by the pool of legitimate users. This form of authentication is very weak, equivalent to a shared password or perhaps an obscure command. However, when strong authentication is incorporated into the port knocking implementation, the common example of offering a service in response to a valid knock is closer to the concept of least privilege than obscurity. In such a case, the service isn't hiding in plain sight. Instead, it only exists to the people who need it (authenticated users) for the amount of time necessary.

The previous example of an SSH server on a firewall is typical of simple implementations of port knocking in that it relies on a static knock using TCP SYN packets - a system analogous to authenticating with a simple shared password. In the example, we are essentially sending the shared password in clear text - not an ideal situation! However, even if we encrypt the knock, we are not necessarily safe from attack. There are two main problems with static knocks: they are easy to detect, and once detected can be replayed by an

[1] See Jeff et al. for an example of this discussion.

attacker.

The Trouble with Static Knocks

Martin Krzywinski brought port knocking into the limelight in 2003 with several articles on his own implementation, which uses static TCP SYN knocks. In a critique of Krzywinski's implementation, Arvind Narayanan notes that traffic can be sniffed to obtain a valid knock sequence ("A critique of port knocking"). While Krzywinski recognizes this in his rebuttal, he also asserts that communication over closed ports is useful when the packet payload is being sniffed - so long as the attacker does not realize that the knock is transmitted in the port numbers ("A Critique of Port Knocking – Author's Response"). It is probably safe to assume that by analyzing the traffic patterns involved, a dedicated attacker would eventually recognize that port knocking is being used. Krzywinski has an excellent example of such traffic - if an attacker notices SYN packets with no response, followed by traffic to an SSH port that usually appears to be closed, making the deduction is probably trivial ("A Critique of Port Knocking – Author's Response"). But there are other motivations for avoiding detection: circumventing intrusion detection systems is one; minimizing log fingerprints is another. In order to accomplish this, the knock must blend in with normal traffic patterns as much as possible.

Once an attacker has sniffed enough traffic to detect the valid knock sequence, the knock could be replayed to the server. Encryption will not necessarily guard against replay in and of itself. A knock that is encrypted using a one-way hash algorithm such as MD5 will create the same ciphertext every time, and can be replayed directly to the server without knowing the plaintext knock. In order for encryption to be useful, it needs to generate a different ciphertext each time. In challenge-response protocols, this is typically dealt with by sending a unique nonce to the client in the challenge. The nonce is then transformed in some manner and sent back to the server, which performs the same transform and compares the results to authenticate the client. The nonce is different for every challenge, preventing an attacker from replaying the response. However, port knocking usually requires one-way communication to the server. The server cannot issue a nonce in a challenge; therefore the only way to incorporate some random data into the knock is for both the client and server to know the data ahead of time, and be able to calculate the knock independently[2]. Alternatively, the knock must be randomly chosen by one party in the communication, and communicated to the other securely.

Guarding Against Detection and Replay

There are a variety of ways to address the issues of detection and replay.

[2] One way to accomplish this with time-synchronized systems is to include a timestamp when calculating the knock.

Covert Knocks

While the concept of port knocking is often described as covert, the implementations often are not. As previously noted, the "closed ports" that are used to communicate the knock are typically TCP ports. Using a knock sequence of TCP SYN packets to known closed ports will look anomalous to anyone monitoring the wire - especially if it is used repetitively. A better option is to encapsulate the knock in otherwise normal traffic. Using connectionless protocols like UDP and ICMP reduces the noise of the knock without sacrificing the stealth of traditional port knocking. In addition, traditional TCP SYN knocks rely on the packets arriving in the correct order - something that cannot be guaranteed without establishing a connection. If the knock is sent using a single packet, ordering is no longer an issue. The use of DNS requests and ICMP packets to send knocks will be explored in our overviews of Net::Pcap, Cerberus, and CÖK.

Dynamic Knocks

We have seen that static knocks are susceptible to replay. One way to combat this is by varying the knock itself for each session. If the knock is different each time, replaying it will only serve to notify the target that it is being sniffed. A dynamic knock is chosen on the fly, on an as-needed basis; ideally, it is random. The chosen knock is then communicated to the other party in the exchange, preferably using an out-of-band communication medium. As we will see in our overview of the SIG^2 Port Knocking Project, there are two difficult problems with dynamic knocks: how to choose the knock, and how to implement the negotiation of the knock securely between the client and the server.

One-Time Knocks

A better way to implement varying knocks lies in the concept of the One-Time Password (OTP). The One-Time Password system specification, RFC 2289, was developed specifically to guard against replay attacks. To do so, it uses iterative hashing to generate a list of passwords from a user's passphrase. Each password is used only once. The system works as follows: initially, the user's passphrase is concatenated with a server seed and run through a one-way hash function n times. The resulting password, P_n, is stored on the server. When the user wishes to authenticate, the server will send a challenge containing the password number, in this case $n-1$, as well as the public server seed. The user will use an OTP generator program to calculate the password from the seed, the password number, and the user's passphrase - the generator will concatenate the seed and password, and apply the one-way hash function $n-1$ times to get the new one-time password. When the server receives this one-time password, P_{n-1}, it will apply the hash function to it once and compare it with the stored password P_n. If the two match, the user is authenticated. The server will then store P_{n-1} for the next login. It will also store the next password

number, `n-2`. The security of this system depends on the fact that the one-way hash cannot be reversed, so the hash algorithm must be selected carefully (Haller).

One-time knocks use the same concepts as OTP, but eliminate the challenge-response protocol. Port knocking implementations that use pre-determined one-time knocks supply the next knock in a series of agreed-upon one-time knocks to the server. Because the series is well-defined in advance, there is no need for the client and server to negotiate the knock or further encrypt the knock sequence. However, such a system also requires additional preparation prior to execution to generate and distribute the list of knocks. The overviews of Cerberus and CÖK will discuss the implementation of one-time knocks, as well as some of their limitations.

Crafting a Covert DNS Knocker with Net::Pcap

In a series of articles for the *Linux Security: Tips, Tricks, and Hackery* newsletter, Brian Hatch outlined a system that uses the Net::Pcap Perl module to sniff for special DNS requests that contain a static knock. In his implementation, a simple Perl program called `watch_dns` watches for UDP packets destined for port 53 on a specific host. The program outputs the source IP, destination IP, and the domain name to look up ("Sniffing with Net::Pcap to stealthily managing iptables rules remotely, Part 1"). A second Perl program tries to map the domain name to a command. If the domain name corresponds to a command, the command will be executed. In the example, a request to look up the domain "openssh" will result in the addition of a new rule to the firewall that allows inbound SSH access for the source IP address ("Running programs in response to sniffed DNS packets - stealthily managing iptables rules remotely, Part 2").

One of the biggest advantages of this implementation of port knocking is ease of use. Because DNS requests can be easily generated with standard tools like `nslookup` and `host`, a special client tool would be unnecessary for generating knocks. Hatch also suggests configuring an authoritative domain server to set up what amounts to a "knock subdomain", called "magic.example.net" in his example. The name server for this new subdomain is the server running the DNS sniffer. As a result, the sniffer will see a DNS query for anything under "magic.example.net". This greatly simplifies sending the knock - a request in a web browser for "http://openssh.magic.example.net" will generate a DNS request to the server running the sniffer. The sniffer can then look for a command that is mapped to "openssh.magic.example.net" ("Running custom DNS queries - stealthily managing iptables rules remotely, Part 3").

One of the biggest disadvantages of Hatch's implementation is actually a side effect of its ease of use. The user is not required to supply a source IP in the knock because it is taken from the UDP packet itself. While this cuts down on typing, it also may unwittingly expand the pool of authorized users. If the tool is

used from behind a NAT device or through a proxy server, the source IP that is granted access may represent an entire company's pool of users! We will see this problem crop up again in other implementations.

Another disadvantage is that while using DNS requests as knocks may not arouse suspicion from an intrusion detection system, an alert attacker sniffing the traffic would no doubt figure out the trick from studying the traffic patterns. In addition, there is no attempt to authenticate the source of the request. As a result, an attacker can easily re-create our DNS requests with a web browser, taking advantage of the ease of use of the implementation. However, Hatch's specific situation - allowing limited SSH access on demand, without enabling constant inbound access - doesn't require a very high level of additional security. If an attacker can supply a valid knock in this situation, all she gets is a login prompt, not total access to the system. What if we wish to use port knocking to trigger an event, not to enable access to a service? The next implementation, Cerberus, adds authentication to the concept of covert knocks to create a more appropriate solution for event triggers.

Cerberus: Covert One-Time Knocks over ICMP

The ICMP port knocker Cerberus is quite possibly one of the oldest publicly known variations on port knocking, having been written by Dana Epp for private use around 1999. Cerberus is a daemon that watches for ICMP ping packets that contain knocks. Like the Net::Pcap implementation, Cerberus does not require a separate knock generator to be installed - a ping tool, like those that come standard on Linux, Unix, and Windows, is all that is required. Using ICMP ping packets to send the knock has the advantage of looking like common traffic, which makes it more likely to pass through intrusion detection systems and firewalls without tripping alarms (Epp).

While the Net::Pcap implementation uses static knocks with no real authentication, Cerberus uses a variation of the one-time knock concept to authenticate the user. To generate a knock, an MD5 hash is constructed from the current date and time to the minute, a server seed, the user's password, and the IP address that will be allowed access (Epp). The last 16 characters of the hash are used as a one-time password, and sent in the ping packet along with the user ID and IP address in plaintext[3]. In order for an attacker to craft a valid knock, she would need to know both the server seed and the user's password.

While using a one-time password mitigates the risk of replay, and use of ICMP may aid in stealth, there are known disadvantages to Cerberus. The most worrisome is probably the risk of denial of service - a ping flood could easily overwhelm the daemon. To mitigate this risk, an intrusion detection system

[3] It is unclear from Epp's slides if this is implemented using iterative hashing in the manner outlined in RFC 2289. The presence of a timestamp implies that iterative hashing is not used, and that the server validates the one-time password by independently calculating it. It is also appropriate to note that the server seed is probably not secret.

could be layered over the port knocking system to filter these sorts of attacks. Another consideration is time synchronization. Since calculation of the one-time password relies on both the client and server choosing the same timestamp, measures need to be taken on the server to ensure that the time chosen upon sniffing the packet will result in the correct hash.

The last and most important consideration is the user herself. Because Cerberus allows the user to specify two pieces of information - the IP address and the password - a great deal of the security in the system relies on the user. If the user is sending a knock to activate access to a service, she must be aware of the number of users that share the IP address she sends, and weigh the risk accordingly. She also must select a password that cannot be trivially guessed. To ensure that the password is strong, complexity requirements should be enforced at the time of selection.

The SIG^2 Port Knocking Project: From Dynamic to One-Time

The port knocking implementation created by SIG^2 specifically seeks to combat replay. The first iteration uses dynamic knock sequences that are determined by the client and declared to the server prior to the actual knock. The implementation also seeks to bolster the stealth inherent in port knocking by randomizing the port that is opened to the client. The port actually forwards the connection to a service that runs on `localhost`.

In the SIG^2 implementation, the client generates a random knock that consists of three port numbers and three Initial Sequence Numbers (ISNs), encrypts it with the user's password hash, and sends it to the server in a UDP packet along with an MD5 HMAC to ensure integrity. The server will use a local copy of the user's password hash to decrypt the knock sequence, and then will wait for the client to send the actual knock. The client sends the knock as three TCP SYN packets that correspond to the ports and ISNs sent in the UDP packet. In addition, it will send a second encrypted UDP packet to the server to check the status of the knock. Once the server receives both the knock and the status packet, it will send the client an encrypted UDP packet that contains the port number that will be opened. To prevent an attacker from replaying all of the packets in the exchange, a timestamp is included in all the encrypted packets (Tan).

An immediate disadvantage to SIG^2's dynamic knock implementation is how noisy it is - not only is the knock transmitted, but also three additional UDP packets. This stands in stark contrast to the previous systems discussed, each of which only required a single packet to send the knock.

A potential problem in SIG^2's implementation is its use of encryption. While the details of the encryption used for the UDP packets are vague, we know that it is a two-way encryption algorithm that involves a shared secret - the user's password hash. Given the amount of information that an attacker could sniff

from the exchange of packets, it is conceivable that the user's password could be cracked. An attacker could sniff and collect the first and last encrypted UDP packets and their corresponding "plaintext" values - matching the first packet with the sniffed TCP SYN knock packets, and the last packet with the port that the client eventually connects to. With this information, the attacker could launch a known-plaintext attack to recover the user's password hash. A better approach would involve using asymmetric-key encryption to encrypt and digitally sign the UDP packets, ensuring both the integrity of the data and the identity of the client.

Shortly after describing their dynamic knock implementation, SIG^2 posted a second paper acknowledging several shortcomings in their implementation, and proposing fixes. The first issue is similar to the denial-of-service concerns in other implementations, but with a twist. The dynamic knock implementation required the server to decrypt the UDP packets sent by the client. As a result, a flood of properly crafted UDP packets could cause resource starvation on the server by forcing a large amount of unnecessary decryption. Another familiar issue involves the randomness of the knock sequence generated on the client. Since the sequence is out of the control of the server, a weak random number generator on the client could make the knock sequences predictable. These two issues were resolved by having the server select the knock sequence, and send it to the client. The decryption problem is then pushed off to the client (Cappella).

But the most important issue is the timestamp used to prevent replay of the UDP packets. Because time synchronization is difficult when the client computer is not pre-determined, the timestamps were removed from the packets in favor of using a variation of a one-time password system. In this new implementation, a user is given an ID and password offline, and sent an encrypted initial knock sequence via email. The user stores the encrypted knock on the client computer. To generate a knock to the server, the knock is decrypted using the user's password and sent to the server. The server sends back an encrypted UDP packet that contains the port number that will be opened and the next knock that the client should use. The next knock is retained on disk until needed (Cappella).

The one-time knock implementation improves upon the noisiness of the dynamic knock version, eliminating two of the three UDP packets. As a trade-off, the user will now need to carry a copy of the next knock sequence if she intends to connect to the server from a different computer. If the client computer indeed varies, it will become more difficult for an attacker to correlate the plaintext knocks to their encrypted counterparts, as they will not be transmitted in the same session. However, the port number and corresponding encrypted packet could still be collected and analyzed. It also appears that the new implementation may have eliminated the MD5 HMAC that was originally sent in the encrypted packets. If so, this reduces the complexity of cracking the

encrypted packets.

If an attacker is able to obtain the user's password, she can sniff the UDP packet sent by the server that contains the next knock and use the knock herself. Once the attacker does this, she will have successfully subverted the series of knocks away from the legitimate user, since she will receive the next knock in the series from the server directly. In addition, the legitimate user will not realize that there is a problem until she tries to use the next knock, which could be weeks or months down the road. Worse, if the system believes she is trying to replay an old knock (since the knock she is using has already been sent by the attacker), she may end up blacklisted.

It is also appropriate to point out that despite the name, this one-time knock system is not an implementation of the One-Time Password system described in RFC 2289. In that specification, the strength of the hash function is what makes or breaks the system. In this one-time knock system, it is the randomness of the numbers selected by the server that the system depends on. The next section looks at an implementation that is directly based on the One-Time Password specification.

CÖK: Cryptographic One-Time Knocking

A port knocking system that adheres more closely to the One-Time Password specification was presented at the 2004 Black Hat conference by David Worth. His implementation, called CÖK, makes a few changes to the specification for simplicity. There is no challenge/response system - instead, the alphanumeric seed in the OTP specification is replaced with a publicly known command that specifies the action to be taken. In addition, the onus is on the user to remember which knock from the list of one-time knocks should be used next.

In Worth's implementation, a daemon watches for UDP packets that contain the next knock. A client tool in the form of a Java applet constructs the knocks by prompting for the knock number, the user's passphrase, the command to be issued to the server, and the target port for the knock. The command and passphrase are concatenated by the client tool, and the one-time knock is calculated using the knock number provided. The knock is encapsulated in a UDP packet and sent to the port specified. The client tool also supports sending the knock in a DNS request (Worth).

The biggest disadvantage of this one-time knock system is the finite nature of the one-time knock list. Eventually, the user will run out of knocks to use and will need to re-initialize this list with a new password or new commands. This may result in the user getting locked out of the system if she forgets that she is out of knocks.

Another disadvantage of this specific implementation is that the server seed has been replaced with a command. It is not clear from the documentation if any

time-sensitive data is added to the knock when it is calculated (like a timestamp). If not, there could be a problem if an administrator sets up knocks with the same password and same commands on two or more servers. The resulting lists of knocks would be identical, which would make them susceptible to replay. In addition, a user that recycles her password and command when re-initializing the list of knocks would end up generating the same list over and over again.

This implementation is probably the most robust of the group, but is also very complex. In addition to the daemon and client, there is a knock manager that allows for administration of knocks and commands. While it improves upon other implementations of one-time knocks, and allows for more flexibility and extensibility, this comes at the price of complexity. This added complexity is one of several continuing issues in port knocking systems in general.

Continuing Issues

There are still some known problems with port knocking that future implementers should be cognizant of:

- Even with the use of one-time knocks, port knocking is still vulnerable to man-in-the-middle attacks. However, this is a well-known problem for many programs, including SSL and SSH.
- Spoofed traffic will probably cause problems for port knocking systems, but the scale depends on the implementation. An attacker would probably use spoofed traffic to frustrate a valid knock (when the knock contains multiple packets), or to send extraneous traffic after a valid knock has been received.
- Denial of service is a problem for port knocking across the board, but may be a bigger concern in systems where the server's port knocking daemon includes proactive techniques to ward off intruders. For instance, Krzywinski suggests that intrusion detection systems can block attempts from attackers to brute-force the port knocking sequence by blacklisting the IP addresses of attackers ("Is Port Knocking an Obscurity Hack?"). Unfortunately, this technique would make it easy to deny service to a legitimate user by spoofing an attack from the user's IP address.
- Finally, part of the attraction to port knocking has been the simplicity of the implementation. As implementations become more complex to ward off the inherent weaknesses of static port knocking, it becomes more likely that vulnerabilities will be introduced into the system. This is a big problem if the system is authorized to manipulate critical pieces of the infrastructure, as is the case when a port knocking system is modifying firewall rules.

None of these issues are necessarily deal-breakers for port knocking; all are

common problems for many networked services. But even without these issues, there are other considerations that might deter administrators from using port knocking in an enterprise environment – at least for now.

Future Work: Enterprise Port Knocking

Most current implementations of port knocking are proof-of-concepts, and as such warn their audiences away from using the code in a production environment. Another oft-repeated assertion is that port knocking should only be used as a layer over an existing secure service, implementing an additional stealthy layer of authentication. But when implemented with authentication that is resistant to replay, is there a real reason that port knocking shouldn't be used as a standalone authentication service? The complexity of implementing such a system may make it overkill for the administrator who just wants to selectively open ports, but in the context of triggering events on the server it would be a good fit. Consider an administrator who needs to reboot a web server remotely. Using a one-time port knock to trigger the reboot would be ideal, especially if using a covert protocol. Such a knock could not be trivially replayed, and would not be easily detected using traffic analysis – no ports are opened in response to the knock, and no response is sent. In this case, the port knocking system acts as an authentication service for potentially many triggers behind the scenes that could be used for debugging, maintenance, and more – all without granting shell access to the user. This has very powerful possibilities for collaborative development and test environments, where several non-privileged users may need to securely invoke very specific events on a server.

Even with robust authentication, port knocking will still need some work before it is ready for the enterprise market. While most of the focus of port knocking implementations is on keeping attackers out, it is only a matter of time before one gets in. An enterprise-level port knocking implementation must address this by working on privilege separation and ensuring that the system fails gracefully. Some questions that implementers should be able to answer include:

- Will a vulnerability in the port knocking daemon result in root access?
- What happens if the port knocking system is compromised or fails when it is in the middle of triggering an event?
- If part of the system is disabled, will the other parts still function (or fail) gracefully?
- If the port knocking system restarts unexpectedly, what effect will the re-initialization have on the predictability of the knocks?
- Is enough logging supported to find the root cause in the event of a failure?

There are other good reasons that an administrator might not want to use port knocking for a large group of users. In cases where a user sends port knocks only sporadically, long periods will go by where the user's account is not being accessed or maintained. One problem with this was illustrated in the

discussion of SIG^2's implementation: if the account is quietly compromised, the user would not notice until she tried to log on again – potentially weeks or months down the road. To prevent this, administrators could use formal periodic access reviews to ensure that accounts have not been subverted without the user's knowledge. An enterprise port knocking system would need to ensure that enough data is logged to conduct these sorts of audits. The system might also be required to provide non-repudiation when critical events are involved.

The long periods between logins also create another problem in an enterprise – password changes. It is common for password policies to require a password change at 60 or 90 days. A port knocking system that enforces this must provide a way for users to change passwords periodically, or must be synchronized with a system that propagates password changes. While synchronizing with an LDAP directory prevents adding an additional interface to the port knocking system for password changes, it also increases the possible points of vulnerability in the PK system. An attacker who wants to gain access to the port knocking system could simply compromise a less secure application to obtain a valid single sign-on password. On the other hand, using one-time knocks for authentication would probably require the port knocking system to manage password changes, since a password change is required whenever the user runs out of valid knocks.

Finally, a port knocking system that allows access from anywhere, anytime may give administrators headaches. While such a system gives remote users significant flexibility, it also removes a lot of useful information that an administrator may depend on to determine if something strange is occurring. For instance, an administrator may normally depend on the location of an IP address, the type of operating system used, or the time of day accessed as clues to whether an unauthorized user may be using the system - for instance with a compromised account or an attempt at replay. Since one of the selling points of port knocking is the ability to authenticate from an arbitrary machine, much of this information becomes far less useful. Enterprise port knocking implementations could allow administrators to create rules that limit this arbitrary scope using allowable IP ranges, OS fingerprinting to ensure that a minimally secure configuration is being used, and enforcing "normal business hours" for certain types of commands.

Overall, there is still much work to be done before port knocking will be a useful tool in a production environment. In the end, the complexity required to meet the needs of such an environment may not be worth the added security. However, it seems logical to conclude that this is not due to any inherent weakness in the concept of port knocking itself. For those who have more flexibility, port knocking can be a novel solution to meet unique needs.

Summary

This paper has described the two major drawbacks of static port knocking - detection and replay. To mitigate these risks, three solutions are proposed: sending the knock over more covert channels such as ICMP or in DNS requests, varying the knock dynamically, and incorporating the concept of one-time passwords into knocks. Four very different implementations display some or all of these characteristics with varying degrees of success. Covert DNS knocks with Net::Pcap succeed at stealth but are still static and susceptible to replay. Cerberus also does a good job of hiding the knocks in ICMP packets, and seems to implement a strong one-time password scheme. SIG^2 implements an interesting dynamic knocker that uses randomly selected knock sequences, but their implementation is noisier than the others and may be vulnerable to cryptanalysis. Finally, CÖK sticks closely to the One-Time Password specification, but may have some problems with replay if no time-sensitive data is incorporated into the knock generation. All port knocking implementations still have to deal with denial of service risks and man-in-the-middle attacks, but when properly implemented can be robust remote authentication mechanisms. However, future implementers will need to work on ensuring that system failures will not lead to full compromise, and that attacks on inactive accounts will be noisy. As port knocking systems grow to meet these needs, implementers must take care to ensure that the complexity of their system does not create more risks than it mitigates.

Conclusions

Port knocking continues to prompt lively debate in the security world. While it may not be the best solution for all circumstances, it seems apparent that port knocking can be considered an extension of the concept of least privilege. Why should a service that is meant for use by a limited group be open to the public for probing? Does the public need to know it exists? However, port knocking has applications beyond just hiding services, like triggering server events on-demand from an arbitrary location. If the security community can keep an open mind and continue to scrutinize the concepts and implementations of port knocking, the result will be a robust new layer in defense-in-depth.

References

Cappella and Tan Chew Keong. "Remote Server Management With One-Time Port Knocking (OTPK)". SIG^2 Port Knocking Project. 08 June 2004. 08 Feb 2005 <http://www.security.org.sg/code/portknock2.html>.

Epp, Dana. "Port Knocking with covert packets to secretly open your firewall." SilverStr's Blog. June 2004. 08 Feb 2005 <http://silverstr.ufies.org/blog/Cerberus.ppt>.

FX. "cd00r.c - not listening remote UN*X shell." Phenoelit. June 2000. 08 Feb 2005 <http://www.phenoelit.de/stuff/cd00rdescr.html>.

Haller, N., et al. "RFC 2289: A One-Time Password System." IETF RFC Repository. Feb 1998. 08 Feb 2005 <http://www.ietf.org/rfc/rfc2289.txt>.

Hatch, Brian. "Running custom DNS queries - stealthily managing iptables rules remotely, Part 3." Hacking Linux Exposed. 25 Aug 2003. 08 Feb 2005 <http://www.hackinglinuxexposed.com/articles/20030825.html>.

Hatch, Brian. "Running programs in response to sniffed DNS packets - stealthily managing iptables rules remotely, Part 2." Hacking Linux Exposed. 14 Aug 2003. 08 Feb 2005 <http://www.hackinglinuxexposed.com/articles/20030814.html>.

Hatch, Brian. "Sniffing with Net::Pcap to stealthily managing iptables rules remotely, Part 1." Hacking Linux Exposed. 30 July 2003. 08 Feb 2005 <http://www.hackinglinuxexposed.com/articles/20030730.html>.

Jeff, et al. "Port Knocking For Added Security." Slashdot. 05 Feb 2004. 16 Feb 2005 <http://slashdot.org/article.pl?sid=04/02/05/1834228>.

Krzywinski, Martin. "A Critique of Port Knocking – Author's Response." Port Knocking. 14 Nov 2004. 08 Feb 2005 <http://www.portknocking.org/view/about/critique>.

Krzywinski, Martin. "Is Port Knocking an Obscurity Hack?" Port Knocking. 13 July 2004. 08 Feb 2005 <http://www.portknocking.org/view/about/obscurity>.

Krzywinski, Martin. "Port Knocking." Linux Journal. 15 June 2003. 08 Feb 2005 <http://www.linuxjournal.com/article/6811>.

Maddock, Ben. "Port Knocking: An Overview of Concepts, Issues and Implementations." SANS Institute. 2004. 08 Feb 2005 <http://www.giac.org/practical/GSEC/Ben_Maddock_GSEC.pdf>.

Narayanan, Arvind. "A critique of port knocking." NewsForge. 08 Oct 2004. 08 Feb 2005 <http://software.newsforge.com/software/04/08/02/1954253.shtml>.

Tan, Chew Keong and Cappella. "Remote Server Management using Dynamic Port Knocking and Forwarding." SIG^2 Port Knocking Project. 02 May 2004. 08 Feb 2005 <http://www.security.org.sg/code/sig2portknock.pdf>.

Worth, David. " CÖK - Cryptographic One-Time Knocking." hexi-dump.org. 2004. 08 Feb 2005 <http://www.hexi-dump.org/bytes/cok/blackhat04_printed_slides.pdf>.

Last Updated: June 10th, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Amsterdam August 2020 Part 1	**Amsterdam, NL**	**Aug 03, 2020 - Aug 08, 2020**	**Live Event**
SANS FOR508 Canberra August 2020	**Canberra, AU**	**Aug 17, 2020 - Aug 22, 2020**	**Live Event**
SANS Amsterdam August 2020 Part 2	**Amsterdam, NL**	**Aug 17, 2020 - Aug 22, 2020**	**Live Event**
SANS Virginia Beach 2020	**Virginia Beach, VAUS**	**Aug 31, 2020 - Sep 05, 2020**	**Live Event**
SANS Philippines 2020	**Manila, PH**	**Sep 07, 2020 - Sep 19, 2020**	**Live Event**
SANS London September 2020	**London, GB**	**Sep 07, 2020 - Sep 12, 2020**	**Live Event**
SANS Baltimore Fall 2020	**Baltimore, MDUS**	**Sep 08, 2020 - Sep 13, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**