Knock: Practical and Secure Stealthy Servers

Julian Kirsch
TU Munich
kirschju@in.tum.de

Maurice Leclaire
TU Munich
leclaire@in.tum.de

Christian Grothoff
TU Munich
grothoff@in.tum.de

Abstract

We present a small modification to the Linux kernel that provides a TCP socket option for advanced kernel-level port knocking. The focus of our work is on usability: with our approach, users and administrators should only have to set a configuration option to make their servers stealthy, while application developers should be able to support this configuration option with about five lines of new code. In this paper, we present the theoretical ideas behind our port knocking approach and discuss the security implications. The appendix includes sample code for integration of our method with applications. Our implementation is freely available from our website.

1 Introduction

Today it is possible to perform a port scan on all machines on the Internet in less than an hour using a single PC [2]. At the same time, major governments are actively developing, collecting and using undisclosed exploits to perform industrial espionage and gain an edge in international politics [6]. Thus it is increasingly important to minimize one's visible footprint and thus attack surface on the Internet. Due to insider threats — further fueled by court orders [3] — this even holds on intranets. Citizens may also simply prefer to not leak information about the services offered by their systems. Finally, applications that try to enable users to cicrumvent censorship — such as Tor bridges [1] — may want to hide their existence from scans by censors.

We also have to consider a modern adversary. Given a censored user, we have to assume that a nation state attacker is able to observe all traffic from the TCP client and perform man-in-the-middle attacks on traffic originating from the client. In fact, existing commercial solutions can initiate a man-in-the-middle attack after the initial TCP handshake has been completed. Furthermore, on the server side, an adversary looking for exploitable systems should be expected to have the ability to perform extensive port scans for TCP servers. Finally, an advanced attacker might be in control of parts of the core of the network, and may thus try to detect unusual patterns in network traffic. However, we assume that adversary does not flag a standard TCP handshake with the TCP server as suspicious, as this is way too common.

This paper describes *Knock*, a kernel modification that enables authorized clients to perform a standard TCP handshake with the server without additional bandwidth and without significant computational overhead by embedding an authorization token in the sequence number (ISN) of the TCP SYN packet. The token demonstrates to the server that the client is authorized and may furthermore authenticate the beginning of the TCP payload to prevent man-in-the-middle attacks. The TCP server is hidden from port scanners and the TCP traffic has no anomalies compared to a normal TCP handshake. Knock is both simpler to implement and deploy and more secure than existing designs.

2 Related Work

Our design is a variant of port knocking [4], where the TCP port only really opens after the client has transmitted some kind of authenticator. The idea of stealthy transmission of an authenticator in a TCP SYN packet is a basic form of network steganography. The specific idea of hiding information in TCP headers including using the ISN header was already described in [7].

Both of these ideas were previously combined in SilentKnock [9], which is a TCP port knocking method that also uses the ISN header to transmit an authenticator. Like the method used in this paper, SilentKnock uses a cryptographic MAC of 32 bits for the authentication and hides this MAC in a TCP SYN packet. In 2007, when the SilentKnock paper was published, the TCP ISN generated by the Linux kernel had only 24 bits of entropy. This forced the SilentKnock developers to use 24 bits of the ISN and

8 bits of the TCP timestamp, which is an optional header field. However, with modern Linux kernels the ISN has 32 bits of entropy and we thus do not need the additional complexity and possible information leak that results from involving a timestamp header. Unlike our design, SilentKnock does not work with clients behind NAT and uses a complex user-space implementation instead of integrating with the Linux kernel directly. SilentKnock also integrates replay protection using per-user counters, which creates challenges due to the possibility of counter desynchronization between client and server. BridgeSPA [8] is another port knocking mechanism which uses nearly the same technique as SilentKnock. In contrast to SilentKnock, BridgeSPA embeds a timestamp to avoid replay attacks. This reduces its use to time synchronized machines.

KnockKnock [5] also hides the authenticator in the TCP ISN number; however, it requires the client to send *two* SYN packets, the first to authenticate and the second to connect to the now opened socket. Our scheme is called *Knock* as it only uses a *single* SYN packet. Furthermore, Knock can be used to authenticate the beginning of the payload, while KnockKnock, BridgeSPA and SilentKnock are all vulnerable to man-in-the-middle attacks.

3 Design for Knock

Knock is a patch to the Linux kernel. Applications set TCP socket options to make a TCP server stealthy, or to connect to a stealthy TCP server. If the respective socket options are set for the TCP server, the kernel handles incoming SYN packets that do not include a proper authenticator in the same way as if the port is closed. For a TCP client, the socket options will cause the kernel to generate the appropriate ISN for a successful handshake.

Our implementation supports two basic variants, with and without payload authentication. Without payload authentication, the ISN is simply generated by truncating the result of a single round of the MD5 hash of a shared secret, which must be supplied by the application logic as an argument to the new `TCP_STEALTH` socket option. If payload authentication is enabled, the highest 16 bits of the authenticator are replaced with a HMAC of the first n bytes of the payload of the TCP stream. The TCP client needs to supply those first n bytes as an argument to the new `TCP_STEALTH_INTEGRITY` option (as the TCP handshake is performed way before the application can provide the payload to the kernel via `write()`). On the server side, the number of bytes n that should be authenticated must be supplied as an argument to the new `TCP_STEALTH_INTEGRITY_LEN` option. If the first n bytes received by the server do not match the HMAC or if the first packet is smaller than n bytes, the server responds to the request by closing the TCP stream. Applications are expected to use the first n bytes to establish a secure channel with the server, for example by transmitting an ECDHE public key.

4 Discussion

Compared to SilentKnock [9] our design does not include sequence numbers to prevent replay attacks. We believe that this is a good idea in practice, as many potential application scenarios will include a single TCP server being used by many clients, and thus strong replay protections in the kernel would mostly create a usability problem. However, if strong replay protections are required, applications can still achieve this by opening a fresh TCP server socket with a fresh secret after each successful connection.

SilentKnock includes the source IP and source destination port in the authentication MAC. However, clients behind NAT typically cannot easily predict the source address that will be seen by the server. As a result, including the source address would prevent about 90% of clients on the Internet today from connecting to the hidden server. Furthermore, an adversary that is placed to observe the TCP SYN packet can typically easily fake the source address. Thus, Knock does not include the source address in the MAC, as this fails to improve security against realistic adversaries while seriously limiting usability.

In theory, another possible problem with NAT is that NAT implementations may replace the ISNs in SYN packets. However, we do not know of any good technical reason for NAT implementations to do this. We are not aware of any research that would definitively say how common ISN replacement is. The only NAT implementation that we know that replaces ISNs is QEMU's NAT for virtual machines. This likely happens because QEMU guests leave some of the TCP logic (such as calculating checksums) to the host operating system.

The reader may be surprised by our choice of MD5 for the hash function. However, MD5 is also used by the Linux kernel for ordinary ISN generation and for TCP SYN cookies and thus the respective computation time and bit patterns should be indistinguishable from ordinary TCP connections. Furthermore, the use of a weaker hash function has no real downside, as the 32 bit values of the ISN hardly require a high-qualtiy hash function.

5 Conclusion

Knock provides a simple and secure method for obsuring the existence of TCP servers on the Internet. Deploying Knock only requires minor modifications to existing client and server software while providing effective protections against modern adversaries. Knock is free software and available from `https://gnunet.org/knock`.

Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1. We thank Jacob Appelbaum constructive discussions on an earlier version of our design.

References

[1] Roger Dingledine and Nick Mathewson. Design of a blocking-resistant anonymity system. Technical report, The Tor Project, Nov 2006.

[2] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, pages 605–619. USENIX Association, 2013.

[3] Ed Felten. A court order is an insider attack. https://freedom-to-tinker.com/blog/felten/a-court-order-is-an-insider-attack/, 2013.

[4] M. Krzywinski. Port knocking: Network authentication across closed ports. *SysAdmin Magazine*, 12:12–17, 2003.

[5] M. Marlinspike. Knockknock. http://www.thoughtcrime.org/software/knockknock/, 2009.

[6] Barack Obama. Presidential policy directive/ppd 20. http://www.theguardian.com/world/interactive/2013/jun/07/obama-cyber-directive-full-text, 2012.

[7] Craig H. Rowland. Covert channels in the tcp/ip protocol suite. *FirstMonday*, 2(5), May 1997.

[8] Rob Smits, Divam Jain, Sarah Pidcock, Ian Goldberg, and Urs Hengartner. Bridgespa: improving tor bridges with single packet authorization. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, WPES '11, pages 93–102. ACM, 2011.

[9] Eugene Y. Vasserman, Nicholas Hopper, and James Tyra. Silentknock: practical, provably undetectable authentication. *Int. J. Inf. Secur.*, 8(2):121–135, February 2009.

A ISN Calculation Details

For the scheme without data integrity protection, the ISN is simply calculated as $ISN := H_K(IP, Port, 0)$ where IP and $Port$ represent the destination address and K is the shared secret. For the scheme with data integrity protection, the ISN is calculated as $ISN := (A, I)$ where A is a 16-bit authenticator and I is a 16-bit integrity protection value. Here, $I := MD5(K \circ Data)$ where $Data$ represents the first n bytes of the payload of the TCP stream, and $A := H_K(IP, Port, I)$. By including I in the calculation of A, we ensure that all bits of the ISN depend on $Data$.

For exact details on how H_K is calculated, please refer to the implementation (i.e. the source files).

B Integration with Applications

Nearly every network application can be easily adapted to use hidden TCP sockets. Only one additional `setsockopt()` call is needed to activate hidden sockets; a second `setsockopt()` call can then optionally be used to authenticate the beginning of the TCP payload. Note that it is up to the application to decide how many bytes of the payload will be authenticated; however, both client and server must agree on the number of authenticated bytes. Listing 1 gives an example for this integration for a client, with authenication for the first 42 bytes of the data sent by the client to the server. Here, after the socket is generated and before `connect()` is called, the application sets the secret which is used for authentication and then notifies the kernel about the first bytes of the payload that it will transmit.

Listing 1: Client code for stealthy TCP

```
int sock;
struct sockaddr_in addr;
char secret[64] = "This is my secret";
char payload[42];
sock = socket(AF_INET, SOCK_STREAM,
              IPPROTO_TCP);
setsockopt(sock, SOL_TCP, TCP_STEALTH,
           secret, sizeof(secret));
setsockopt(sock, SOL_TCP,
           TCP_STEALTH_INTEGRITY,
           payload, sizeof(payload));
connect(sock, &addr, sizeof(addr));
write(sock, payload, sizeof(payload));
```

The modifications on the server side would be similar (Listing 2); the main difference is that the server would specify how many bytes of authenticated payload are expected.

Listing 2: Server code for stealthy TCP

```
int sock;
struct sockaddr_in addr;
char secret[] = "This is my secret";
uint32_t len = 42;
sock = socket(AF_INET, SOCK_STREAM,
              IPPROTO_TCP);
setsockopt(sock, SOL_TCP, TCP_STEALTH,
           secret, sizeof(secret));
setsockopt(sock, SOL_TCP,
           TCP_STEALTH_INTEGRITY_LEN,
           &len, sizeof(len));
listen(sock, 5);
```

While our examples are for IPv4, our patch works for both IPv4 and IPv6.