TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Improved Kernel-Based Port-Knocking in Linux

Julian Kirsch

Version 1.1 (November 7, 2014)

TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

Master's Thesis in Informatics

Verbessertes Kernelbasiertes Port-Knocking unter Linux

Improved Kernel-Based Port-Knocking in Linux

Author:	Julian Kirsch
Supervisor:	Christian Grothoff, PhD (UCLA)
Date:	15. August 2014

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nur die angegebenen Quellen verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ort, Datum	Julian Kirsch

Version History

Version Number	Date	Annotations
1.0	August 15th, 2014	First release
1.1	November 7th, 2014	Added the `headers_install` step to the user guide

Contents

Acknowledgements

Knock started as a student project during the *Peer-to-peer Systems and Security* lecture as given by Christian Grothoff in Summer 2013. Since then, *Knock* has undergone drastic change and improvement which would not have taken place without the help of many people.

I thank my fellow student Maurice Leclaire, who can well be considered the co-father of the original *Knock* implementation. Without your deep understanding of the Linux kernel and your passion to create software that satisfies even the most ambitious objectives, *Knock* could not exist in its current format. Keep on rocking!

I thank my supervisor, Christian Grothoff, for giving me the opportunity to write this thesis, and also for your guidance to overcome the conceptual hurdles that appeared along the way. Thank you for hours of discussions and drawings on packets travelling through networks, cryptography and possible attack scenarios. Thank you for giving me the opportunity to make *Knock* known to the public during the *30c3*-conference and for extensive editing. Without you, TCP Stealth would not be as effective as it is today. I wish you all the best for your future!

I thank my co-authors of the Heise article [16], Christian Grothoff, Monika Ermert, Jacob Appelbaum, Laura Poitras and Henrik Moltke, for providing context and motivation for my thesis. Certain parts of the background chapter in this thesis are based on this joint effort. I also thank our source, and hope that this work helps in the spirit that motivated the choice to provide the information.

During the feasibility analysis for the project, Bart Polot was generous in providing a smart phone which helped to incorporate data of mobile devices. A big *thank you* to all contributors, especially to the people of the GNUnet community who kindly executed the measuring program on their devices. You were a great help.

Thanks also fly out to several persons who indirectly contributed to *Knock* by pointing out key issues during discussions, be it at the lunch table or during another presentation of *Knock*'s core concepts – Thomas Kittel and Florian Sesser, thanks for your time and helpful words.

Last but not least I want to thank my loving and supportive family and my wonderful girlfriend Barbara. I would not be where I am today without you.

Abstract

Port scanning is used to discover vulnerable services and launch attacks against network infrastructure. Port knocking is a well-known technique to hide TCP servers from port scanners. This thesis presents the design of TCP Stealth, a socket option to realize new port knocking variant with improved security and usability compared to previous designs.
TCP Stealth replaces the traditional random TCP SQN number with a token that authenticates the client and (optionally) the first bytes of the TCP payload. Clients and servers can enable TCP Stealth by explicitly setting a socket option or linking against a library that wraps existing network system calls.
This thesis also describes Knock, a free software implementation of TCP Stealth for the Linux kernel and `libknockify`, a shared library that wraps network system calls to activate Knock on GNU/Linux systems, allowing administrators to deploy Knock without recompilation. Finally, we present experimental results demonstrating that TCP Stealth is compatible with most existing middleboxes on the Internet.

Abstrakt

Portscans werden zum Auffinden und Ausnutzen verwundbarer Netzwerkdienste durchgeführt. Port-Knocking ist eine allgemein bekannte Technik, die es ermöglicht, TCP Server vor Portscannern zu verstecken. Diese Arbeit stellt TCP Stealth vor, welches eine neue Variante des Port-Knocking darstellt, die gegenüber bestehenden Konzepten eine höhere Sicherheit sowie eine bessere Benutzerfreundlichkeit verspricht.
TCP Stealth ersetzt die übliche Zufallszahl im TCP-SYN-Segment durch eine kryptographische Prüfsumme, welche den Client authentisiert und optional die ersten Bytes des TCP Datenstroms gegen Modifizierungen schützt. Clients und Server aktivieren TCP Stealth durch das Setzen einer Socket-Option oder verwenden eine dynamische Bibliothek, welche die relevanten Systemaufrufe passend zu den bereits Bestehenden ausführt.
In dieser Arbeit wird weiterhin *Knock* vorgestellt, eine Freie-Software-Implementierung von TCP Stealth im Linux-Kernel sowie `libknockify`, eine dynamische Bibliothek, die es Administratoren ermöglicht, Knock ohne Neukompilieren bestehender Anwendungen zu verwenden. Außerdem stellen wir die Ergebnisse unserer Experimente vor, welche zeigen, dass TCP Stealth kompatibel mit den meisten existierenden NAT-Geräten im Internet ist.

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 [6]. 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 [21]. 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 [9] — 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 [5] — may want to hide their existence from scans by censors.

Port knocking [17] is a method for making TCP servers less visible on the Internet. The basic idea is to make a TCP server not respond (positively) to a TCP SYN request unless a particular "knock" packet has been received first. This can be helpful for security, as an attacker who cannot establish a TCP connection also cannot really attack the TCP server.

Traditional port knocking techniques generally do not consider a modern nation state adversary. 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, it may still be safe to assume that adversary does not flag a standard TCP handshake with the TCP server as suspicious, as this is way too common.

This thesis describes TCP Stealth, a design for a stealthy port knocking method 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.

TCP Stealth is a variant of port knocking [17], 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 [27], and used for port knocking in SilentKnock [30].

The thesis also describes *Knock*, an implementation of TCP Stealth for the Linux kernel. Knock consists of a kernel patch and a shared library `libknockify` that can be used to add TCP Stealth support to existing applications without modifying the binaries. Knock makes it easy to implement and deploy TCP Stealth, providing a more usable and secure port knocking scheme compared to existing designs.

Key contributions of this thesis include:

- TCP Stealth offers integrity protection of the payload, defeating a man-in-the-middle attack trying to take control of a session after the TCP SYN packet. However, this only works for well-designed application protocols where the client provides key material for the session with the first bytes of the payload.

- TCP Stealth works with most middleboxes on the Internet, while SilentKnock fails in the presence of any NAT device.

- TCP Stealth is supposed to be fully integrated with the kernel and the design does not require additional proxies or daemons to run on the system. Thus, it is trivial to use TCP Stealth on operating systems that support it. We believe this is crucial to convince applications and system administrators to use it by default.

- On GNU/Linux, `libknockify` enables the use of TCP Stealth without the need to recompile existing network applications. Using the `LD_PRELOAD` mechanism, support for TCP Stealth can be easily activated — assuming the local kernel includes the Knock patch.

- TCP timestamps are used in order to randomize the authenticator and the integrity protector. For the reason that TCP timestamps describe relative time values the clocks of the communicating end-hosts do *not* need to be synchronized.

2 Background

2.1 The TCP Three Way Handshake

Whenever a TCP client wants to communicate with a TCP server, the two parties perform a TCP three way handshake (see [29], Section 6.5.5 and [24], Section 3.4). The flawed design of this handshake is the foundation for port mapping tools, as in the handshake the server leaks information about the availability of a service without checking the client's authorisation.

Figure 1 shows the sequence of TCP segments which are sent to establish a connection. In brevity, the establishment of the connection works as follows: The first TCP SYN segment is sent out by the host which wants to initiate a connection which is acknowledged by a SYN/ACK packet in case of the destination host accepting the connection request. After receiving a positive reply, host 1 sends out an ACK packet which finalizes the TCP three-way handshake.

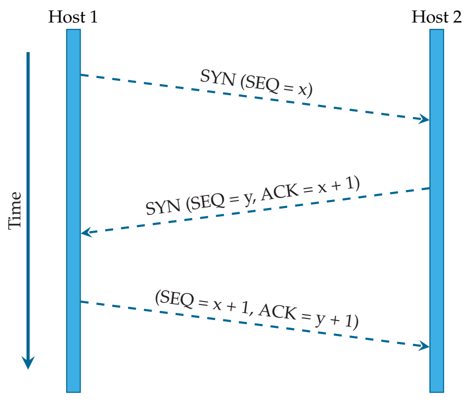


Figure 1: *Packet flow during the TCP three-way handshake*

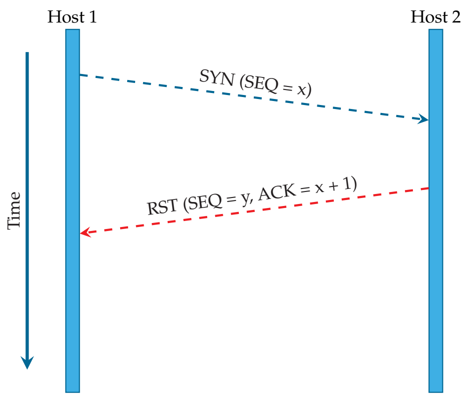
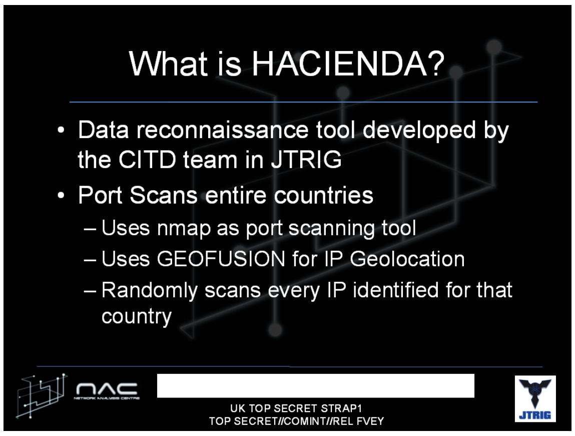


Figure 2: *Packet flow for a connection attempt to a closed TCP port*

The TCP three way handshake allows an adversary to easily determine if some TCP service is offered at a given port by a host on the Internet: if the TCP port is closed, the server reacts differently to the TCP SYN packet (Figure 2). Thus, an adversary can easily map Internet services by considering the differences in the server's replies in the packet flows depicted in Figure 1 and Figure 2 respectively.

2.2 A Modern Adversary Model

Since the early days of TCP, port scanning has been used by saboteurs to locate vulnerable systems. In a new set of top secret documents, it is revealed that in 2009, the British spy agency GCHQ made port scans a "standard tool" to be applied against entire nations (Figure 3). Twenty-seven countries are listed as targets of the HACIENDA program in the presentation (Figure 4), which comes with a promotional offer: readers desiring to do reconnaissance against another country need simply send an e-mail (Figure 5). The documents do not spell out details for a review process or the need to justify such an action. It should also be noted that the ability to port-scan an entire country is hardly wild fantasy; in 2013, a port scanner called Zmap was implemented that can scan the entire IPv4 address space in less than one hour using a single PC. [6] The massive use of this technology can thus make any server anywhere, large or small, a target for criminal state computer saboteurs.



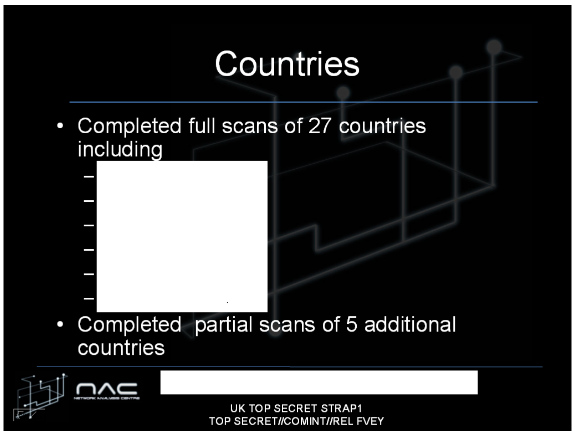
What is HACIENDA?

- Data reconnaissance tool developed by the CITD team in JTRIG
- Port Scans entire countries
  - Uses nmap as port scanning tool
  - Uses GEOFUSION for IP Geolocation
  - Randomly scans every IP identified for that country

[redacted]

UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

Figure 3



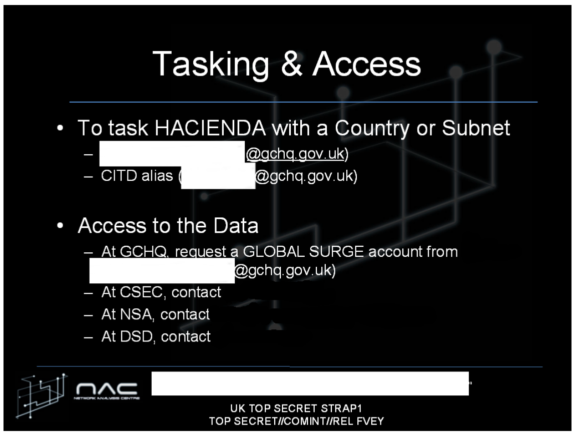
Countries

- Completed full scans of 27 countries including
  - [redacted]
- Completed partial scans of 5 additional countries

[redacted]

UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

Figure 4



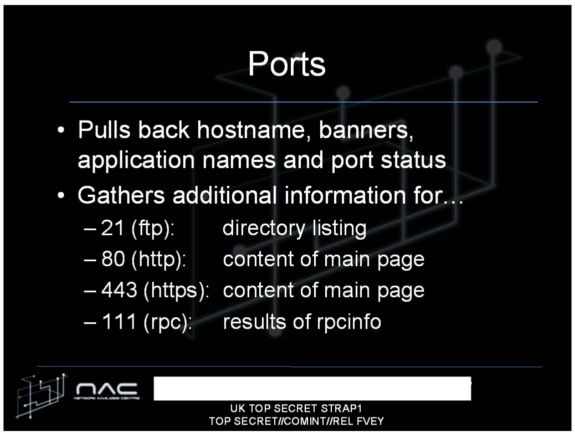
Tasking & Access

- To task HACIENDA with a Country or Subnet
  - [redacted]@gchq.gov.uk)
  - CITD alias ([redacted]@gchq.gov.uk)
- Access to the Data
  - At GCHQ, request a GLOBAL SURGE account from [redacted]@gchq.gov.uk)
  - At CSEC, contact
  - At NSA, contact
  - At DSD, contact

[redacted]

UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

Figure 5



Ports

- Pulls back hostname, banners, application names and port status
- Gathers additional information for…
  - 21 (ftp): directory listing
  - 80 (http): content of main page
  - 443 (https): content of main page
  - 111 (rpc): results of rpcinfo

[redacted]

UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

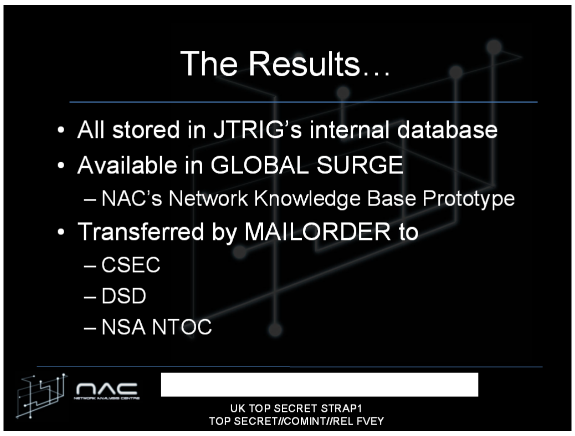
Figure 6

The list of targeted services includes ubiquitous public services such as HTTP and FTP, as well as common administrative protocols such as SSH and SNMP (Figure 6). Given that in the meantime, port scanning tools like Zmap have been developed which allow anyone to do comprehensive scans, it is not the technology used that is shocking, but rather the gargantuan scale and pervasiveness of the operation.

2.2.1 The Enemy Online

In addition to simple port scans, GCHQ also downloads so-called banners and other readily available information (Figure 6). A banner is data, usually text, sent by some applications when connecting to an associated port; this often indicates system and application information, including version and other information useful when looking for vulnerable services. Doing reconnaissance at the massive scale revealed in the documents demonstrates that the goal is to perform active collection and map vulnerable services ubiquitiously, not to go after specific targets. Rather than the cliche of merely listening to everything, the documents presented show active interaction with networks and systems.

By preparing for attacks against services offered via SSH and SNMP, the spy agency targets critical infrastructure such as systems used for network operations. As shown in the past with the penetration of Belgacom [1] and Stellar [23], when an employee's computer system or network credentials may be useful, those systems and people are targeted and attacked.

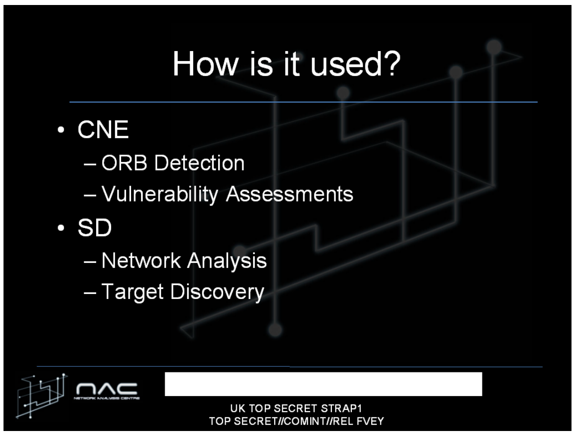


The Results…

- All stored in JTRIG's internal database
- Available in GLOBAL SURGE
  - NAC's Network Knowledge Base Prototype
- Transferred by MAILORDER to
  - CSEC
  - DSD
  - NSA NTOC

UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

Figure 7



How is it used?

- CNE
  - ORB Detection
  - Vulnerability Assessments
- SD
  - Network Analysis
  - Target Discovery

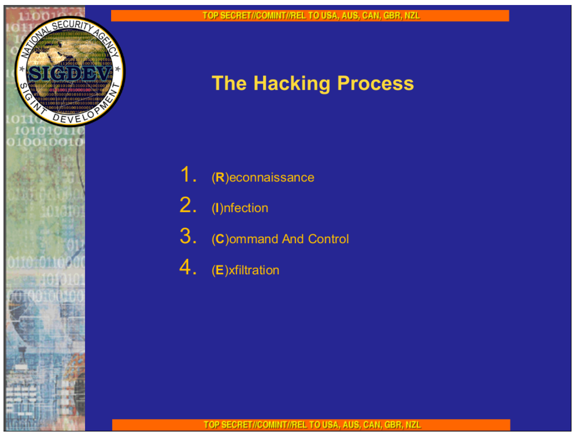
UK TOP SECRET STRAP1
TOP SECRET//COMINT//REL FVEY

Figure 8: *CNE stands for Computer Network Exploitation.*

The database resulting from the scans is then shared with other spy agencies of the Five Eyes spying club (Figure 7), which includes the United States, Canada, United Kingdom, Australia and New Zealand. MAILORDER is described in the documents as a secure transport protocol used between the Five Eyes spy agencies to exchange collected data.

2.2.2 Every Device a Target

The process of scanning entire countries and looking for vulnerable network infrastructure to exploit is consistent with the meta-goal of "Mastering the Internet", which is also the name of a GCHQ cable-tapping program. These spy agencies try to attack every possible system they can, presumably as it might provide access to further systems. Systems may be attacked simply because they might eventually create a path towards a valuable espionage target, even without actionable information indicating this will ever be the case. Using this logic, every device is a target for colonization, as each successfully exploited target is theoretically useful as a means to infiltrating, for monitoring or as an operational location that is useful for another possible target.



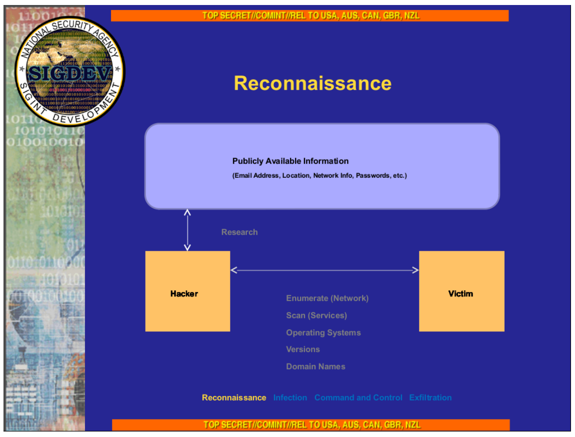
TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

The Hacking Process

1. (**R**)econnaissance
2. (**I**)nfection
3. (**C**)ommand And Control
4. (**E**)xfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 9



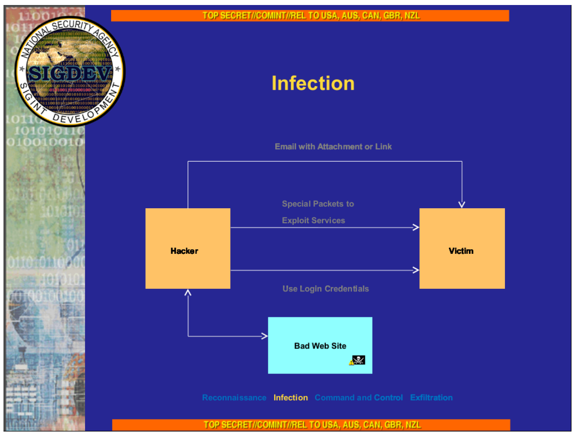
TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Reconnaissance

Publicly Available Information
(Email Address, Location, Network Info, Passwords, etc.)
Research
Hacker
Victim
Enumerate (Network)
Scan (Services)
Operating Systems
Versions
Domain Names
Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 10



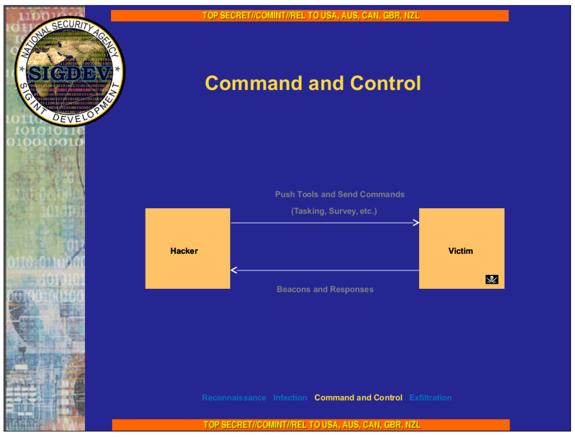
TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Infection

Email with Attachment or Link
Special Packets to
Exploit Services
Hacker
Victim
Use Login Credentials
Bad Web Site
Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 11



TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Command and Control

Push Tools and Send Commands
(Tasking, Survey, etc.)
Hacker
Victim
Beacons and Responses
Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 12

Port scanning and downloading banners to identify which software is operating on the target system is merely the first step of the attack (Figure 8). Top secret documents from the CSEC, NSA and GCHQ seen by Heise demonstrate that the involved spy agencies follow the common methodology of online organized crime (Figure 9): reconnaissance (Figure 10) is followed by infection (Figure 11), command and control (Figure 12), and exfiltration (Figure 13).

The NSA presentation makes it clear that the agency embraces the mindset of criminals. In the slides, they discuss techniques and then show screenshots of their own tools to support this criminal process (Figure 14, 15 and 16).

2.2.3 Internet Colonization

The NSA is known to be interested in 0-day attacks, which are attacks exploiting largely unknown vulnerabilities for which no patch is available. Once an adversary armed with 0-day attacks has discovered that a vulnerable service is running on a system, defense becomes virtually impossible.

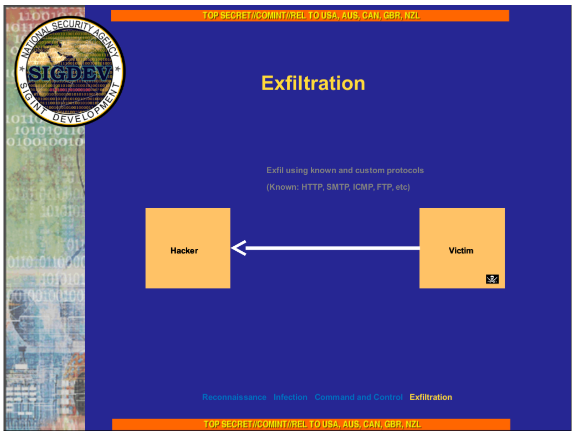
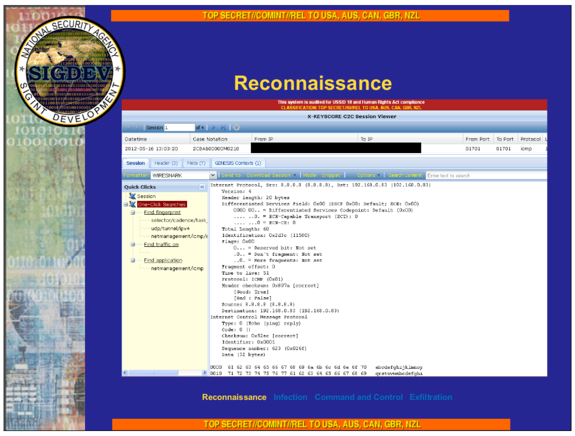

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL
NATIONAL SECURITY AGENCY SIGDEV SIGINT DEVELOPMENT
Exfiltration
Exfil using known and custom protocols
(Known: HTTP, SMTP, ICMP, FTP, etc)
Hacker
Victim
Reconnaissance Infection Command and Control Exfiltration
TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 13


TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Reconnaissance

This system is audited for USSID 18 and Human Rights Act compliance
CLASSIFICATION: TOP SECRET//SI//REL TO USA, AUS, CAN, GBR, NZL

X-KEYSCORE C2C Session Viewer

Session 1 of 4

Datetime	Case Notation	From IP	To IP	From Port	To Port	Protocol
2012-05-16 13:03:20	2CBAB00000M0210	[redacted]	[redacted]	01701	01701	icmp

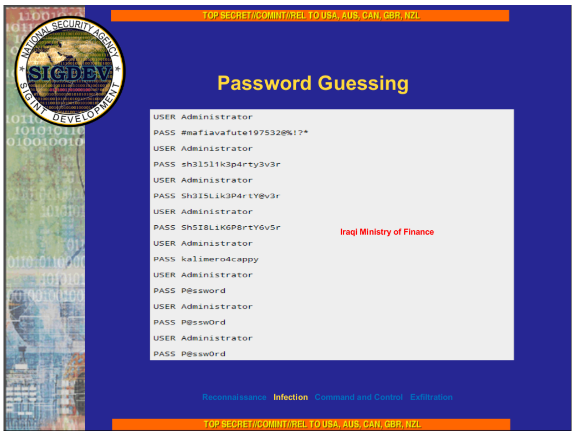
```
Internet Protocol, Src: 8.8.8.8 (8.8.8.8), Dst: 192.168.0.83 (192.168.0.83)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 60
    Identification: 0x2d3c (11580)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 51
    Protocol: ICMP (0x01)
    Header checksum: 0x897a [correct]
        [Good: True]
        [Bad : False]
    Source: 8.8.8.8 (8.8.8.8)
    Destination: 192.168.0.83 (192.168.0.83)
Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0 ()
    Checksum: 0x52ec [correct]
    Identifier: 0x0001
    Sequence number: 623 (0x026f)
    Data (32 bytes)

0000  61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70   abcdefghijklmnop
0010  71 72 73 74 75 76 77 61 62 63 64 65 66 67 68 69   qrstuvwabcdefghi
```

Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 14


TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Password Guessing

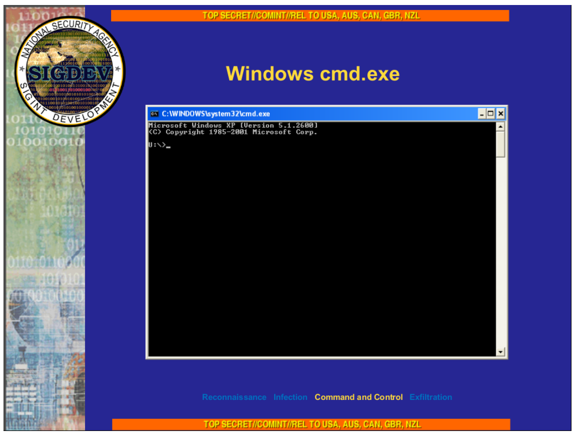
```
USER Administrator
PASS #mafiavafute197532@%!?*
USER Administrator
PASS sh3l5l1k3p4rty3v3r
USER Administrator
PASS Sh3I5Lik3P4rtY@v3r
USER Administrator
PASS Sh5I8LiK6P8rtY6v5r
USER Administrator
PASS kalimero4cappy
USER Administrator
PASS P@ssword
USER Administrator
PASS P@sswOrd
USER Administrator
PASS P@sswOrd
```

Iraqi Ministry of Finance

Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 15


TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Windows cmd.exe

C:\WINDOWS\system32\cmd.exe

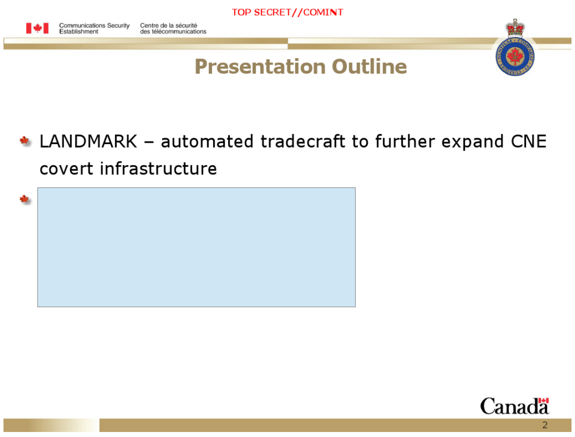
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

U:\>_
```

Reconnaissance Infection Command and Control Exfiltration

TOP SECRET//COMINT//REL TO USA, AUS, CAN, GBR, NZL

Figure 16


TOP SECRET//COMINT

Communications Security Establishment Centre de la sécurité des télécommunications

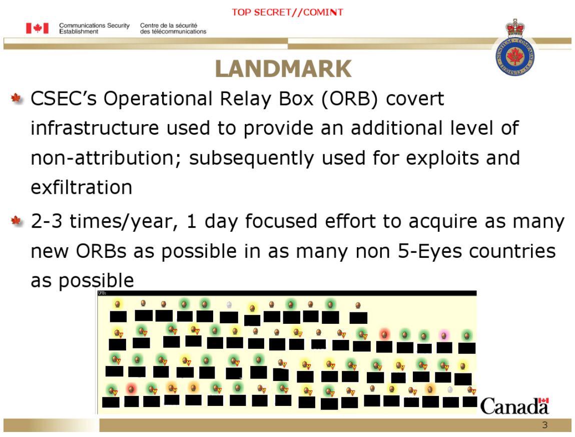
Presentation Outline

- LANDMARK – automated tradecraft to further expand CNE covert infrastructure
-

Canada

2

Figure 17


TOP SECRET//COMINT

Communications Security Establishment Centre de la sécurité des télécommunications

LANDMARK

- CSEC's Operational Relay Box (ORB) covert infrastructure used to provide an additional level of non-attribution; subsequently used for exploits and exfiltration
- 2-3 times/year, 1 day focused effort to acquire as many new ORBs as possible in as many non 5-Eyes countries as possible

[redacted]

Canada

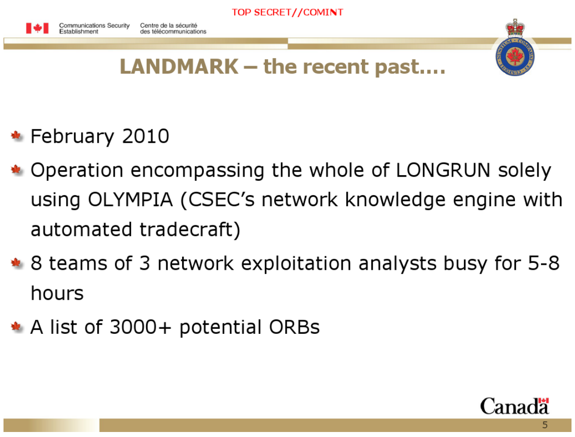
3

Figure 18

Firewalls are unlikely to offer sufficient protection, whether because administrators need remote access or because spy agencies have already infiltrated the local network [11].

Furthermore, adding additional equipment, such as firewalls administered via SNMP, into an internal network may also open up new vulnerabilities.

Figure 8 points to a particular role that HACIENDA plays in the spy club's infrastructure, namely the expansion of their covert infrastructure. The top secret documents seen by Heise describe the LANDMARK program, a program by the Canadian spy agency CSEC which is used to expand covert infrastructure (Figure 17).


TOP SECRET//COMINT

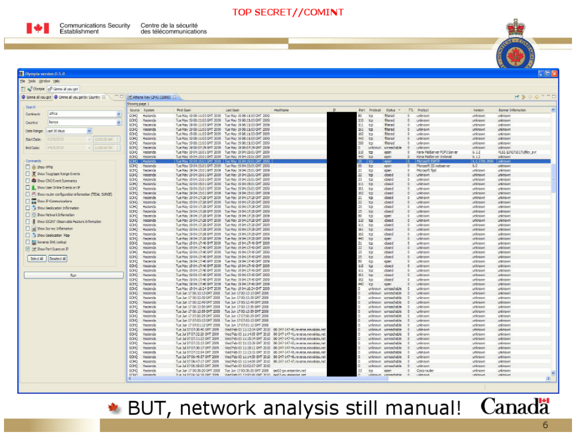
Communications Security Establishment Centre de la sécurité des télécommunications

LANDMARK – the recent past….

- February 2010
- Operation encompassing the whole of LONGRUN solely using OLYMPIA (CSEC's network knowledge engine with automated tradecraft)
- 8 teams of 3 network exploitation analysts busy for 5-8 hours
- A list of 3000+ potential ORBs

Canada 5

Figure 19


TOP SECRET//COMINT

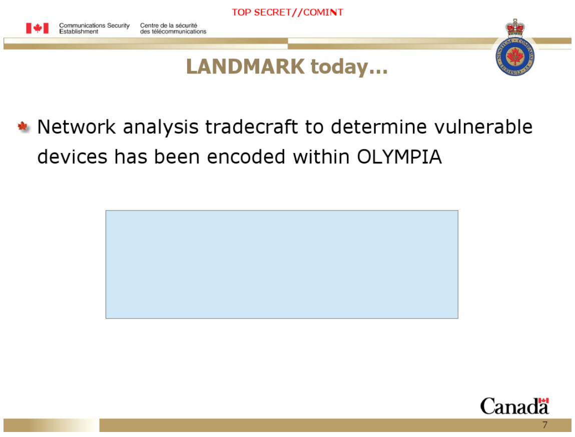
Communications Security Establishment Centre de la sécurité des télécommunications

[redacted]

- BUT, network analysis still manual!

Canada 6

Figure 20


TOP SECRET//COMINT

Communications Security Establishment Centre de la sécurité des télécommunications

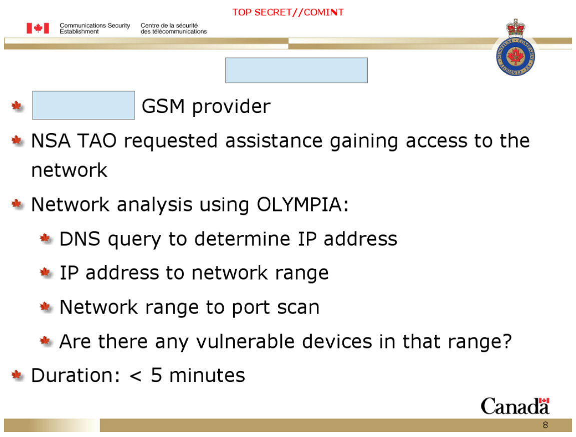
LANDMARK today…

- Network analysis tradecraft to determine vulnerable devices has been encoded within OLYMPIA

[redacted]

Canada 7

Figure 21


TOP SECRET//COMINT

Communications Security Establishment Centre de la sécurité des télécommunications

[redacted]

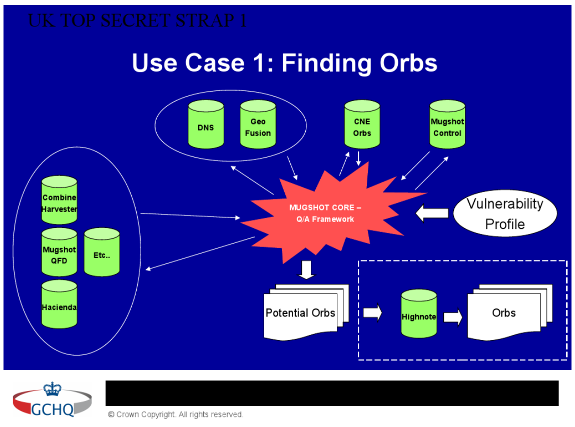
- [redacted] GSM provider
- NSA TAO requested assistance gaining access to the network
- Network analysis using OLYMPIA:
  - DNS query to determine IP address
  - IP address to network range
  - Network range to port scan
  - Are there any vulnerable devices in that range?
- Duration: < 5 minutes

Canada 8

Figure 22

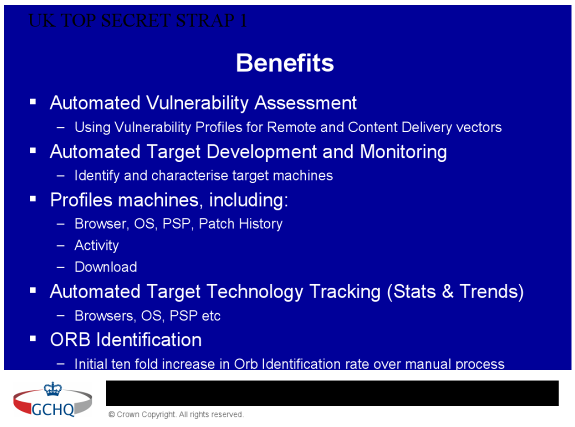
The covert infrastructure includes so-called Operational Relay Boxes (ORBs), which are used to hide the location of the attacker when the Five Eyes launch exploits against targets or steal data (Figure 18). Several times a year, the spy club tries to take control of *as many machines as possible*, as long as they are abroad. For example, in February 2010 twenty-four spies located over 3000 potential ORBs in a single work day (Figure 19). However, going over the port scan results provided by HACIENDA was considered too laborous (Figure 20), so they programmed their OLYMPIA system to automate the process (Figure 21). As a result, the spies brag that they can now locate vulnerable devices in a subnet in less than five minutes (Figure 22).

The Canadians are not the only ones using HACIENDA to locate machines to compromise and turn into ORBs. At GCHQ, the hunt for ORBs is organized as part of the MUGSHOT program (Figure 23). The GCHQ has also automated the process and claims significant improvements in accuracy due to the automation (Figure 24). Again the information obtained from HACIENDA plays a prominent role (Figure 25). A key point is that with MUGSHOT the GCHQ integrates results from active scans (HACIENDA) as well as passive monitoring (Figure 26), to "understand everything important about all machines on the Internet".



[redacted]

Figure 23


UK TOP SECRET STRAP 1

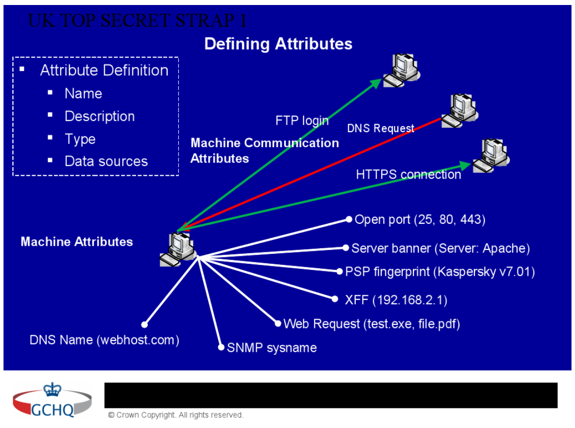
Benefits

- Automated Vulnerability Assessment
  - Using Vulnerability Profiles for Remote and Content Delivery vectors
- Automated Target Development and Monitoring
  - Identify and characterise target machines
- Profiles machines, including:
  - Browser, OS, PSP, Patch History
  - Activity
  - Download
- Automated Target Technology Tracking (Stats & Trends)
  - Browsers, OS, PSP etc
- ORB Identification
  - Initial ten fold increase in Orb Identification rate over manual process

GCHQ [redacted]

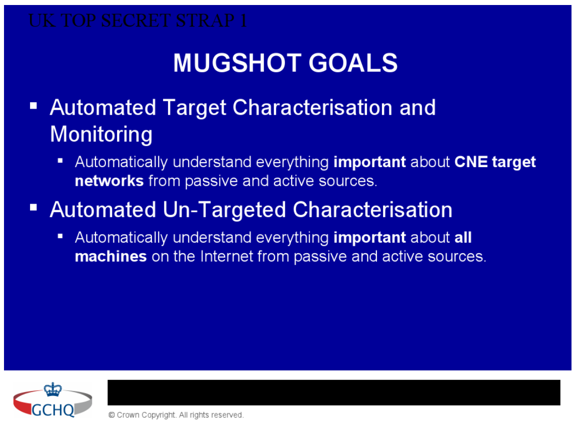
© Crown Copyright. All rights reserved.

Figure 24



[redacted]

Figure 25


UK TOP SECRET STRAP 1

MUGSHOT GOALS

- Automated Target Characterisation and Monitoring
  - Automatically understand everything **important** about **CNE target networks** from passive and active sources.
- Automated Un-Targeted Characterisation
  - Automatically understand everything **important** about **all machines** on the Internet from passive and active sources.

GCHQ [redacted]

© Crown Copyright. All rights reserved.

Figure 26

Thus, system and network administrators now face the threat of industrial espionage, sabotage and human rights violations created by nation-state adversaries indiscriminately attacking network infrastructure and breaking into services. Such an adversary needs little reason for an attack beyond gaining access and is supported by a multi-billion dollar budget, immunity from prosecution, and compelled collaboration by companies from Five Eyes countries. As a result, every system or network administrator needs to worry about protecting his system against this unprecedented threat level. In particular, citizens of countries outside of the Five Eyes have, as a result of these programs, greatly reduced security, privacy, integrity and resilience capabilities.

3 The Design of TCP Stealth

This section outlines and explains the decisions that were taken during the design process of TCP Stealth. The requirements and design goals are evaluated and justified and the general operation of TCP Stealth is explained. At a later point, focus is being placed on how TCP Stealth achieves authenticity of the communication participants as well as integrity protection of the first data that is sent by the side initiating the connection.

3.1 Overview

TCP Stealth assumes the existence of a secret that is known to both of the communicating parties, which serves as a *pre-shared-key* (PSK). The PSK needs to be distributed to the communication partners using a secure channel prior to any communication. Specifically, the PSK must be given to the TCP clients and the TCP server before the TCP handshake can be performed.

The basic goal of TCP Stealth is to limit connections to the TCP server to clients that know the correct PSK. Other clients should ideally not be able to deduce the existence of the TCP server; in particular, for unauthorized clients the TCP handshake should fail as if the server was not running. Furthermore, an adversary performing a man-in-the-middle attack after the TCP handshake should not be able to replace the payload of the first TCP segment sent from the client to the server after the handshake.

TCP Stealth is designed for a single PSK per TCP server. Thus, multiple authorised clients are expected to use the same PSK if they communicate with the same TCP server. Naturally, the application can change the PSK at any time, for example if the system administrator decides to change the passphrase.

3.2 Stealthiness

A key question for any port knocking design is how to communicate the “knock”, a message that informs the TCP server that the client is authorised to communicate.

For TCP Stealth, we decided that it should operate in a way that can not be distinguished from an ordinary TCP connection by a passive attacker.[1] The reason is that we do

[1]An active attacker is able to determine that *her* attempt to connect to the TCP server somehow fails. If the adversary then also observes authorised clients connecting successfully, she can deduce that the TCP server is somehow protected.

not want to needlessly alert network security middleware to the presence of TCP Stealth. Also, by not sending any unusual traffic, TCP Stealth is more likely to work in the presence of firewalls and other systems that may filter traffic they consider abnormal.

The requirement for a stealthy operation immediately rules out the possibility of sending specially crafted *additional* data, such as a UDP packet or any other special "knock" packet. In fact, even an *additional* TCP SYN might raise suspicion and thus should not be permitted.

For a truly stealthy knock, the security token has to be embedded in the TCP SYN part of the TCP three way handshake. Providing the security token later would enable an adversary to perform the handshake and thus detect that the port is open, and providing the security token outside of the TCP SYN would create additional traffic that could raise suspicion.

As TCP Stealth is to be stealthy, it is obvious that the TCP SYN used by TCP Stealth may not structually deviate from ordinary TCP SYN packets. Comparing the TCP header fields (see Figure 27) with their predefined values and meanings according to RFC793 [24] it follows that only the sequence number can be used in order to covertly transmit information: Source and destination port as well as the window size, the flags and the checksum cannot be altered without destroying the semantics of the SYN packet, and the acknowledgement number, data offset and the urgent pointer are defined to be zero when the SYN flag is set. This leaves the sequence number the only good place to hide information.

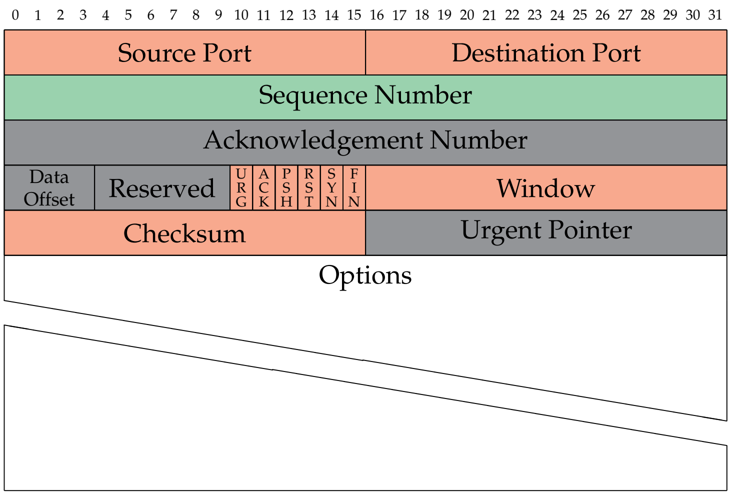


Figure 27: *TCP header fields as defined per RFC793 [24], section 3.1.), colored according to their predefined values in a SYN segment – red fields cannot be altered without changing the semantics of the connection request, gray fields are defined to be zero, green fields can be chosen arbitrarily, white fields are optional*

The sequence number conveyed by the first SYN packet is referred to as the *initial sequence number* (ISN). Thus, TCP Stealth interprets the value of the ISN as a security token; its value must match the result of a specific calculation involving the PSK, thereby demonstrating that the TCP client is in possession of the PSK and thus authorised to connect. In case a connecting client fails to set the correct ISN and is thus assumed to not know about the PSK, the TCP server reacts as if the port were closed (see Figure 2).

The main limitation of using the ISN for the security token is that this limits the security token to 32 bits, giving the adversary a chance of $1 : 2^{32}$ to access the service without knowing the PSK. Using a non-standard port for the TCP service may further lower the chance of accidental discovery to $1 : 2^{48}$.

This design choice of embedding the security token into the ISN also simplifies the implementation of the TCP server: the server's state machine can remain virtually unchanged, with the exception of a possible transition to an error state if the ISN is incorrect.

3.3 The Authentication Security Token

Authentication itself is desired in order to provide a way for the server to tell if the client is authorised to know about the existence of the TCP server. Note that in TCP Stealth authentication is done only in one direction: the server can be (almost, see section 8.11) sure that the client is in possession of the PSK, but the client can not tell if she is communicating with the intended server. Thus, clients using TCP Stealth must still authenticate the server using application-level protocols.

Authentication of the communication partners is achieved by the client proving the ownership of the PSK. The PSK is a symmetric (shared) secret with the following properties:

- The PSK is 64 bytes long (which offers sufficient security and works well with the MD5 hash function)
- The PSK should satisfy the statistical requirements of a random number
- The PSK needs to be known only to authorised clients and the server

In the following, we will refer to a binary string which fulfills all of the above requirements as the secret S. To enable TCP Stealth, both client and server must set the TCP Stealth option and provide the secret S as the argument.

Conceptually, the authentication token A is then calculated as the hash (h) of the destination address (consisting of the destination IP address IP_d and TCP port P_d) and a TCP timestamp T:

$$\mathrm{A} := h((IP_d, P_d, T), S) \tag{1}$$

More precisely, TCP Stealth uses a single round of the MD5 hash function for $h(IV, D)$ (i.e. the function MD5Transform defined by RFC 1321 [26], Appendix 3). Here, D is the domain of the hash function and IV is the initialisation vector. A single round of MD5 was chosen as the Linux kernel *already* uses MD5 to calculate ISNs. Furthermore, given that the final output has to be reduced to a 32 bit value, the quality of MD5 is sufficient for the level of security attainable with only 32 bits. Finally, a single round of MD5 is very fast — indeed, MD5 is also used for the calculation of TCP SYN COOKIES to defeat TCP SYN flooding attacks [2,7]. Thus, MD5 is also a good choice with respect to defeating timing attacks and ensuring availability.

Algorithm 1 describes the steps of the ISN generation for authentication in pseudo code. The goal of this presentation is to enable binary-compatible implementations of the method, thus the specific layout of the data in memory is included. The $\oplus$ denotes bitwise xor operation. Arrays accesses are written with byte indexes starting from zero in brackets. The timestamp T is obtained from the TCP timestamp option, represented in T in the following algorithms. As the timestamp option is optional, a value of zero is used if the option is not present.

Algorithm 1 ISN generation without integrity protection

```
Require: $P_d$, $\mathrm{IP_d}$ in network byte order $\wedge$ $\mathrm{secret}[0:63] \neq 0$
Ensure: ISN in network byte order
  if $\nexists\, T$ then
    $T \Leftarrow 0$
  end if
  if network layer is IPv4 then
    $\mathrm{IV}[0:3] \Leftarrow \mathrm{IP_d}[0:3]$
    $\mathrm{IV}[4:15] \Leftarrow 0$
  else
    if network layer is IPv6 then
      $\mathrm{IV}[0:15] \Leftarrow \mathrm{IP_d}[0:15]$
    end if
  end if
  $\mathrm{IV}[8:11] \Leftarrow \mathrm{IV}[8:11] \oplus T$
  $\mathrm{IV}[12:13] \Leftarrow \mathrm{IV}[12:13] \oplus P_d$
  $H[0:15] \Leftarrow \mathrm{MD5Transform}(\mathrm{IV}[0:15], \mathrm{secret}[0:63])$
  return $H[0:3] \oplus H[4:7] \oplus H[8:11] \oplus H[12:15]$
```

Algorithm 1 calculates the 32 bit ISN authentication token as follows: First, the 16 bytes of an initialization vector IV are set to the destination address, in the case of IPv4 padded with zeros. Then, the timestamp value (in network byte order) is XORed at offsets 8 to 11, and finally the destination port (in network byte order) is XORed at offsets 12 to 13 (those are expected to be low-entropy offsets in IPv6 addresses). Then, A is calculated using a single round of MD5 (MD5Transform) using IV for the initialisation vector and the 64-byte PSK as the argument for the hash function. Finally, the resulting four 32-bit words of the MD5 hash are combined using XOR to calculate the 32-bit ISN in network byte order.

The server can perform the same calculation and compare the received ISN with its own result. If the ISN fails to match, the client is not authorised and the connection is refused.

3.4 Authentication and Integrity Protection

The ISN calculation from the previous section can be used if only client authentication is desired. However, TCP Stealth can also be used to offer integrity protection for the first segment of TCP data transmitted from the TCP client to the TCP server. This is important given the possibility of an advanced adversary performing a man-in-the-middle attack after the TCP handshake.

The design of TCP Stealth limits integrity protection to the first TCP segment, as we wanted to ensure that integrity checks would be applied before the TCP server application would receive any data from the TCP client. By limiting integrity protection to the first TCP segment, the kernel can check the payload integrity before ever passing data to the application. As the TCP client's kernel may combine data from multiple `write()` operations into a single segment (see `TCP_CORK`[2]), and as IP routers may fragment packets,

[2]man 7 tcp

both client and server must also apriori agree on the number of bytes that are to be integrity protected.

To enable integrity protection, the TCP client must provide the payload (and length) that is to be protected to the TCP Stealth implementation *before* the TCP handshake is initiated. This is unproblematic, as long as the application-protocol is designed such that the client begins the conversation. On the server side, integrity protection is enabled by setting an option that provides the number of bytes that are to be integrity protected. Naturally, both client and server also still need to provide the PSK.

When TCP Stealth is used to provide authentication and integrity protection, the security token is conceptually split into two halves (A, I), one providing authentication (A) and the other integrity protection (I). I is simply calculated as the hash of the integrity protected payload and the PSK. A is now calculated as the hash (h) of the destination address (consisting of the destination IP address IP_d and TCP port P_d), the TCP timestamp T and I:

$$\mathrm{A} := h((IP_d, P_d, T, I), S) \tag{2}$$

Algorithm 2 describes the steps of the ISN generation for authentication and integrity protection in pseudo code. $\oplus$ denotes the bitwise xor operator while the $\circ$ means concatenation. Arrays accesses are written with byte indexes starting from zero in brackets.

Algorithm 2 ISN generation with integrity protection

```
Require: $P_d$, $\mathrm{IP_d}$ in network byte order $\wedge$
  $\mathrm{len} \neq 0 \wedge \mathrm{payload}[0:\mathrm{len}] \neq 0 \wedge \mathrm{secret}[0:63] \neq 0$
Ensure: ISN in network byte order
  if $\nexists T$ then
    $T \Leftarrow 0$
  end if
  $I[0:15] \Leftarrow \mathrm{MD5}(\mathrm{secret}[0:64] \circ \mathrm{payload}[0:\mathrm{len}])$
  $\mathrm{IH}[0:1] \Leftarrow I[0:1] \oplus I[2:3] \oplus I[4:5] \oplus I[6:7] \oplus I[8:9] \oplus I[10:11] \oplus I[12:13] \oplus I[14:15]$

  if network layer is IPv4 then
    $\mathrm{IV}[0:3] \Leftarrow \mathrm{IP_d}[0:3]$
    $\mathrm{IV}[4:15] \Leftarrow 0$
  else
    if network layer is IPv6 then
      $\mathrm{IV}[0:15] \Leftarrow \mathrm{IP_d}[0:15]$
    end if
  end if
  $\mathrm{IV}[4:5] \Leftarrow \mathrm{IV}[4:5] \oplus \mathrm{IH}[0:1]$
  $\mathrm{IV}[8:11] \Leftarrow \mathrm{IV}[8:11] \oplus T$
  $\mathrm{IV}[12:13] \Leftarrow \mathrm{IV}[12:13] \oplus P_d$
  $\mathrm{AV}[0:15] \Leftarrow \mathrm{MD5Transform}(\mathrm{IV}[0:15], \mathrm{secret}[0:63])$
  $\mathrm{AV}[0:3] \Leftarrow \mathrm{AV}[0:3] \oplus \mathrm{AV}[4:7] \oplus \mathrm{AV}[8:11] \oplus \mathrm{AV}[12:15]$
  return $\mathrm{AV}[0:1] \circ \mathrm{IH}[0:1]$
```

For the operation mode with integrity protection, the protected portion of the payload is limited to at most the first segment. Additionally, both client and server must know the exact number of bytes to be protected beforehand. The secret is prepended to the protected payload and hashed using full MD5 and the resulting eight 16-bit words of the MD5 hash are combined using XOR to a 16-bit integrity hash (IH).

Then, the 32 bit ISN is calculated using a single round of MD5 using the 64-byte shared secret as the argument for the hash function and a 16 byte initialization vector IV computed as follows:

First, the 16 bytes are set to the destination address, in the case of IPv4 padded with zeroes. The IH is XORed at offset 4 to 5. Then, the timestamp value (in network byte order) is XORed at offsets 8 to 11, and finally the destination port (in network byte order) is XORed at offsets 12 to 13.

The resulting four 32-bit words output by the MD5Transform function are combined using XOR to calculate at 32-bit authenticator value (AV). The upper 16 bit of the AV are concatenated with the 16 bit of the IH to create the final 32-bit ISN in network byte order.

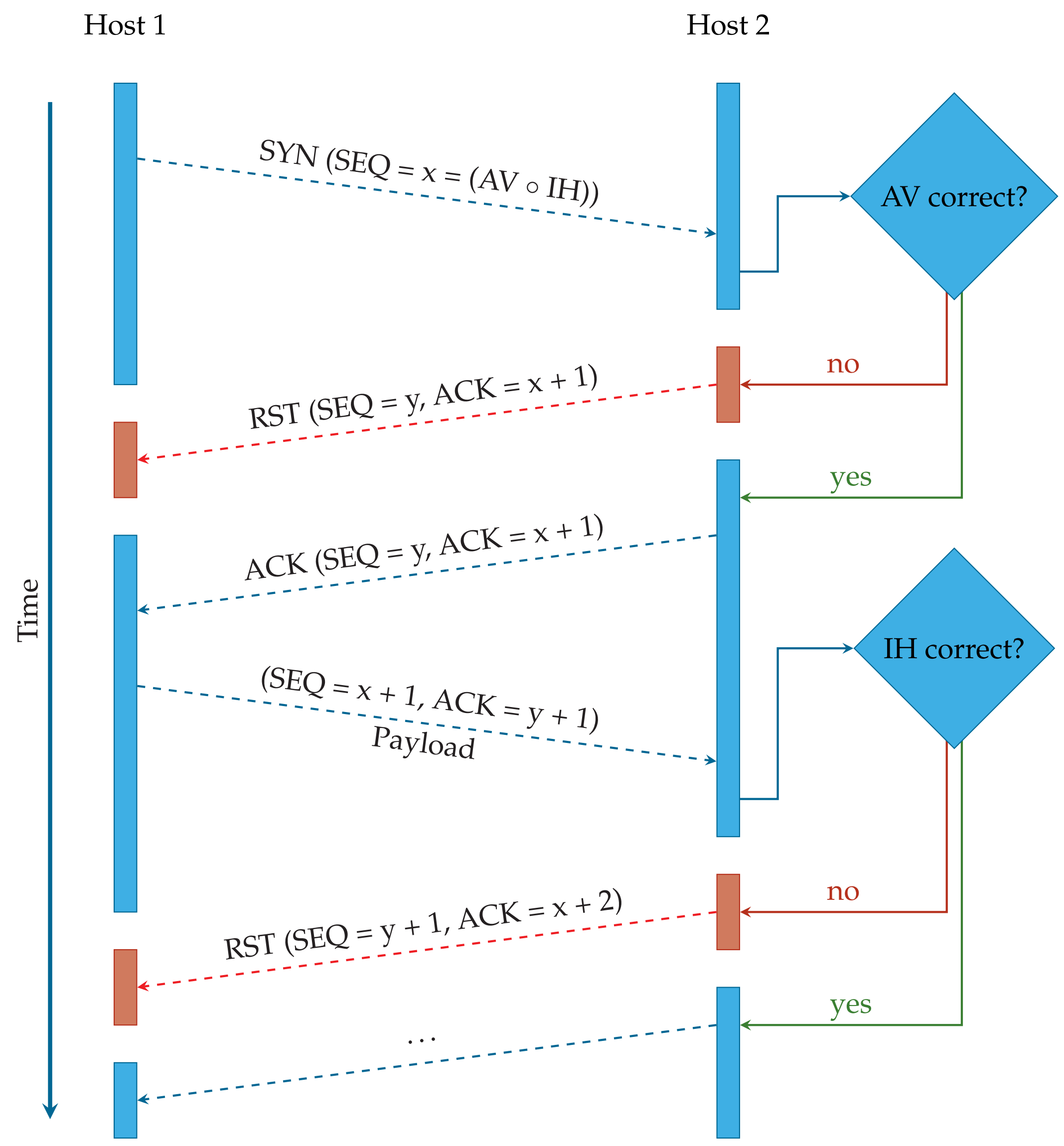

Host 1
Host 2
Time
SYN (SEQ = x = (AV ∘ IH))
AV correct?
no
RST (SEQ = y, ACK = x + 1)
yes
ACK (SEQ = y, ACK = x + 1)
IH correct?
(SEQ = x + 1, ACK = y + 1)
Payload
no
RST (SEQ = y + 1, ACK = x + 2)
yes
. . .

Figure 28: *Possible packet and message flows during a TCP Stealth enhanced TCP handshake*

The TCP server uses the 16 bit IH from the ISN to calculate the AV when receiving the TCP SYN packet. If the AV matches, the handshake is allowed to proceed. Then, when the TCP server receives the first segment, it checks that the included payload is of sufficient length and in combination with the shared secret hashes to the IH. If these checks fail, the connection is closed (by sending a TCP RST). The three-way-handshake with authentication and integrity protection is shown in Figure 28.

Given a handshake using TCP Stealth with integrity protection, a TCP client protocol would then ideally transmit critical data of some application-layer key exchange during the first TCP segment. For example, a TCP client might send his half of a Diffie-Hellman (DH) key exchange in the first segment. By providing integrity protection for the first segment, TCP Stealth then makes it difficult for an adversary to perform a man-in-the-middle attack after the TCP handshake, as substituting the first TCP segment with the attacker's key exchange data would likely ($1 - 1/2^{16}$ in the best case, $1 - 1/2^{15}$ on average) be detected by the TCP server.

By including I in the calculation of A, we ensure that if I changes the authenticator A also changes; this prevents an adversary from separately breaking authentication and integrity protection. While the adversary can now (in the best case) detect the existence of the TCP server with probability $1/2^{16}$ ($1/2^{15}$ on average), sending arbitrary application payload to the server still only succeeds with probability $1/2^{32}$ in the best case ($1/2^{31}$ on average).

3.5 SYN Retransmissions

In case the first SYN segment does not reach the server (due to packet loss or by being intentionally withheld by an attacker) a all following SYN requests by this connection may not update the TCP timestamp value, if present. This is needed to ensure that the server side can still correctly verify authenticity and integrity.

Note that if the client intends to establish a second connection, the source port will change and thus even a repated ISN will not confuse the server — even if the TCP timestamp did not advance and the content integrity protections are not used.

4 Implementation

TCP Stealth was implemented in two components. The first component is a patch to the Linux kernel called "Knock", which adds support for a `setsockopt()` call to activate TCP Stealth. The second component is a user space shared library called `libknockify` which can be linked against existing binary to enable TCP Stealth for legacy code without modifications to the source.

4.1 Kernel Space

This section explains which functions are added to the Linux kernel and where all relevant calls to the newly added functions are placed. We also give a detailed overview of the edits done at a file level.

4.1.1 Summary of the Patched Files

In summary, the proposed kernel patch modifies eleven source files in the kernel. This section briefly explains the purpose of each modification. All paths are relative to the root directory of the Linux kernel source.

include/linux/tcp.h At the beginning, an inclusion of `linux/cryptohash.h` is added in order to make the `MD5_MESSAGE_BYTES` constant available to the code. The `struct tcp_sock` is extended to contain the `struct stealth` which holds all needed information about the current state of TCP Stealth.

include/net/secure_seq.h The signatures of the functions `tcp_stealth_do_auth` and `tcp_stealth_sequence_number` are added to make them available to the TCP/IP stack.

include/net/tcp.h The signatures of the functions `tcp_parse_tsval_option` and `tcp_stealth_integrity` are added to make them available to the TCP/IP stack as well as the two macros `be32_isn_to_be16_av` and `be32_isn_to_be16_ih`.

include/uapi/linux/tcp.h The modifications to this file introduce the option numbers for the new *setsockopt* calls `TCP_STEALTH`, `TCP_STEALTH_INTEGRITY` and `TCP_STEALTH_INTEGRITIY_LEN` and makes them available to the user space applications.

net/core/secure_seq.c Several include directives add signatures of functions which are needed by the added code. The functions `tcp_stealth_sequence_number` and `tcp_stealth_do_auth` are newly defined.

net/ipv4/Kconfig We add a configuration parameter to the network branch of the kernel called `TCP_STEALTH` which is used to toggle TCP Stealth support at compile time. At the time of writing, the default value for this option is set to not include TCP Stealth support in the kernel.

net/ipv4/tcp.c The patch adds the definition of the new function `tcp_stealth_integrity` as well as the definition of the *setsockopt* handlers for the newly introduced *setsockopt* calls `TCP_STEALTH`, `TCP_STEALTH_INTEGRITY` and `TCP_STEALTH_INTEGRITIY_LEN`.

net/ipv4/tcp_input.c In this file the definition of the functions `tcp_parse_tsval_option` and `__tcp_stealth_integrity_check` is added. Two patches of existent functions are performed:

- Patch the function `tcp_rcv_established` to call `tcp_stealth_integrity_check` in order to verify the integrity protector of any incoming segment that was able to pass the authenticator check (fast path).
- Patch the function `tcp_data_queue` to call `__tcp_stealth_integrity_check` in order to verify the integrity protector of any incoming segment that was able to pass the authenticator check (slow path).

net/ipv4/tcp_ipv4.c The patch adds an include directive in order to make the TCP Stealth related functions as defined in `net/secure_seq.h` available. The function `tcp_v4_connect` is patched to call `tcp_stealth_sequence_number` to generate TCP Stealth conforming ISN values and `tcp_v4_do_rcv` is patched to call `tcp_stealth_do_auth` in order to verify the authenticator of incoming connections.

net/ipv4/tcp_output.c Modifications to this file concern the handling of the TCP timestamp option. The function `tcp_connect` is patched to use the TSVal that was stored at the time the ISN was generated whereas the function `__tcp_retransmit_skb` is patched to revert the TSVal field in case of any SYN retransmits to the original value of the first SYN packet of the current TCP session.

net/ipv6/tcp_ipv6.c The patch adds an include directive in order to make the TCP Stealth related functions as defined in `net/secure_seq.h` available. The function `tcp_v4_connect` is patched to call `tcp_stealth_sequence_number` to generate TCP Stealth conforming ISN values and `tcp_v4_do_rcv` is patched to call `tcp_stealth_do_auth` in order to verify the authenticator of incoming connections.

4.1.2 New Functions

The *Knock* patch introduces five new functions which are described hereafter.

tcp_stealth_sequence_number This function is responsible for creating a ISN that can be used by TCP Stealth by calculating the AV and optionally (if integrity protection is enabled) adds the IH to the constructed AV. It does so by applying a unification of the steps specified by Algorithms 1 and 2 on the input parameters.

`tcp_stealth_sequence_number`		
Declaration	include/net/secure_seq.h	
Definition	net/core/secure_seq.c	
Xrefs	`tcp_stealth_do_auth`, `tcp_v4_connect`, `tcp_v6_connect`	
Callees	`memcpy`, `md5_transform`	
Parameters	`struct sock *sk`	a pointer to a TCP Stealth enabled `struct sock`
	`__be32 *daddr`	a pointer to a memory location containing the `daddr_size` many bytes of the destination IP address in network byte order
	`u32 daddr_size`	the size of the destination IP address in bytes (i.e. either 4 for IPv4 or 16 for IPv6) in host byte order
	`__be16 dport`	the destination port in network byte order
	`u32 tsval`	item the value of the TSVal field of the optional TCP timestamp extension in host byte order, or 0 if no timestamp is going to be used
Return Value	a 32-bit ISN in host byte order	

Table 1: *Summary of function* `tcp_stealth_sequence_number`

We go through the function as shown in Listing 1 step by step:

After the function signature a `struct tcp_sock *tp` is initialized with a pointer to the current TCP socket on which TCP Stealth should be used. Two local arrays are allocated on the stack: One of type `__u32` called `sec` which will hold an intermediate representation of the PSK (see below) and another one of type `__le32` called `iv` which will hold the initialization vector IV (compare Algorithms 1 and 2). The resulting ISN in big endian format will be stored in the local variable `isn` and `i` denotes the usual loop counter used in several `for` statements.

After the declaration of the local variables the destination address is copied into the IV using `memcpy`. Note that the zero-padding as described by the formal algorithms in section 3.4 is achieved by initializing the IV with zeroes at runtime. According to the formal algorithm, lines 15 to 17 modify the IV using the exclusive or operator (note the conversions to network byte order). Even though the integrity hash IH is only present in Algorithm 2 the described function implements a unified version of both formal Algorithms, the integrity hash is defined to be zero if only the authenticator should be calculated. In the latter case the exclusive or operator in line 15 simply becomes a NOP as basic math tells. ($X \oplus 0 = X$)

In order to provide portability across little and big endian platforms the IV and the secret are converted to the machine's native format in lines 19 to 22. This is needed because the `md5_transform` function that is built into the kernel implements one round of MD5 which, according to RFC 1321, works on blocks of size `MD5_MESSAGE_BYTES` by interpreting the data and the iv as 32-bit words in the CPUs native format to produce a hash of length `MD5_DIGEST_WORDS` many 32-bit words (i.e. `MD5_DIGEST_WORDS` times 4 many bytes).

For the same reason (portability), the result of the MD5 round (now residing in `iv`) is converted back to big endian and the accumulative exclusive or of the resulting 32-bit words as specified by the formal Algorithms is performed. The only point where the current TCP Stealth mode matters is when the ISN is output: Only ff integrity checking is desired the second 16-bit word (in network byte order) of the resulting ISN is replaced with the 16 bits of the integrity hash IH (in network byte order). Afterwards the result is converted to the machine's native byte order and is returned to the caller.

```
u32 tcp_stealth_sequence_number(struct sock *sk, __be32 *daddr, u32 daddr_size,
                                __be16 dport, u32 tsval)
{
        struct tcp_sock *tp = tcp_sk(sk);

        __u32  sec[MD5_MESSAGE_BYTES / sizeof(__u32)];
        __u32 i;

        __be32 iv[MD5_DIGEST_WORDS] = { 0 };
        __be32 isn;

        memcpy(iv, (const __u8 *)daddr,
               (daddr_size > sizeof(iv)) ? sizeof(iv) : daddr_size);

        ((__be16 *)iv)[2] ^= cpu_to_be16(tp->stealth.integrity_hash);
        iv[2] ^= cpu_to_be32(tsval);
        ((__be16 *)iv)[6] ^= dport;

        for (i = 0; i < MD5_DIGEST_WORDS; i++)
                iv[i] = le32_to_cpu(iv[i]);
        for (i = 0; i < MD5_MESSAGE_BYTES / sizeof(__le32); i++)
                sec[i] = le32_to_cpu(((__le32 *)tp->stealth.secret)[i]);

        md5_transform(iv, sec);

        isn = cpu_to_be32(iv[0]) ^ cpu_to_be32(iv[1]) ^
              cpu_to_be32(iv[2]) ^ cpu_to_be32(iv[3]);

        if (tp->stealth.mode & TCP_STEALTH_MODE_INTEGRITY)
                be32_isn_to_be16_ih(isn) =
                        cpu_to_be16(tp->stealth.integrity_hash);

        return be32_to_cpu(isn);

}
```

Listing 1: *Source code of the function* `tcp_stealth_sequence_number`

tcp_stealth_integrity This function is responsible for calculating the integrity hash IH. It does so by applying the steps specified by Algorithms 1 and 2 on the input parameters.

`tcp_stealth_integrity`		
Declaration	include/net/tcp.h	
Definition	net/ipv4/tcp.c	
Xrefs	`do_tcp_setsockopt`, `tcp_stealth_integrity_check`	
Callees	`crypto_alloc_hash`, `sg_init_table`, `sg_set_buf`, `crypto_hash_digest`, `crypto_free_hash`	
Parameters	`__be16 *hash`	a pointer to a memory location where the integrity hash IH will be stored
	`u8 *secret`	a pointer to the TCP Stealth secret
	`u8 *payload`	a pointer to a memory location holding at least `len` bytes of the payload that should be hashed
	`int len`	the number of bytes which should be hashed
Return Value	zero if construction of the IH succeeded, an error number otherwise	

Table 2: *Summary of function* `tcp_stealth_integrity`

The function as shown in Listing 2 starts with the declaration of several local variables on the stack: a `struct scatterlist sg` of size two is needed to supply pointers to the distributed data in memory for hashing. The `struct crypto_hash *tfm` and the `struct hash_desc desc` belong to the kernel's crypto API [20] and describe the state of the hash function. The array `h` is used to store the resulting hash – it is declared to be of type `__be16` which allows accessing the hash in a way which allows us to perform the 16-bit-wise accumulation using exclusive or operations as introduced by the algorithms in section 3.4. The local integers `i` and `err` are used as a loop counter and to hold error codes that might arise during the computation.

The construction of the hash is straightforward: lines 10 to 16 allocate an instance of the MD5 hash function and initialize the needed structures in the descriptor of the hash function. In lines 18 to 20 the scatterlist is initialized to refer to the secret and the payload respectively. Using a scatterlist is advantageous as it drops the need to copy the data into a contiguous block before hashing. By calling `crypto_hash_digest` in line 22, the concatenation of `MD5_MESSAGE_BYTES` many bytes of the `secret` and `len` many bytes of the `payload` are finally hashed using MD5.

Afterwards lines 27 to 29 combine each of the eight 16-bit words of the MD5 hash using the exclusive or operator after converting the output of the hash function to its native byte order first. (This is because the final destination of the integrity hash IH, the `struct stealth` is defined to hold the IH in host byte order.)

```
 1 int tcp_stealth_integrity(u16 *hash, u8 *secret, u8 *payload, int len)
 2 {
 3         struct scatterlist sg[2];
 4         struct crypto_hash *tfm;
 5         struct hash_desc desc;
 6         __be16 h[MD5_DIGEST_WORDS * 2];
 7         int i;
 8         int err = 0;
 9 
10         tfm = crypto_alloc_hash("md5", 0, CRYPTO_ALG_ASYNC);
11         if (IS_ERR(tfm)) {
12                 err = -PTR_ERR(tfm);
13                 goto out;
14         }
15         desc.tfm = tfm;
16         desc.flags = 0;
17 
18         sg_init_table(sg, 2);
19         sg_set_buf(&sg[0], secret, MD5_MESSAGE_BYTES);
20         sg_set_buf(&sg[1], payload, len);
21 
22         if (crypto_hash_digest(&desc, sg, MD5_MESSAGE_BYTES + len, (u8 *)h)) {
23                 err = -EFAULT;
24                 goto out;
25         }
26 
27         *hash = be16_to_cpu(h[0]);
28         for (i = 1; i < MD5_DIGEST_WORDS * 2; i++)
29                 *hash ^= be16_to_cpu(h[i]);
30 
31 out:
32         crypto_free_hash(tfm);
33         return err;
34 }
```

Listing 2: *Source code of the function* `tcp_stealth_integrity`

Finally the label `out` marks the cleanup procedure that remains to be done: The allocated hash function is released and (consequently) kernel memory is freed. The return value `err` is initialized to zero and thus is still zero if hashing was successful or an error code if hashing failed for some reason.

tcp_stealth_do_auth This function is called by the server side in order to authenticate the connection request of a client. This is done by checking if the AV meets the requirements imposed by sections 3.3 and 3.4.

`tcp_stealth_do_auth`		
Declaration	include/net/secure_seq.h	
Definition	net/core/secure_seq.c	
Xrefs	`tcp_parse_tsval_option`, `tcp_stealth_sequence_number`, `printk`	
Callees	`tcp_v4_do_rcv`, `tcp_v6_do_rcv`	
Parameters	`struct sock *sk`	a pointer to a TCP Stealth enabled listening socket `struct sock`
	`struct sk_buff *skb`	a pointer to a `struct sk_buff` which holds the contents of the SYN packet whose ISN should be checked
Return Value	zero if authentication was successful, nonzero otherwise	

Table 3: *Summary of function* `tcp_stealth_do_auth`

The function as shown in Listing 3 starts with the allocation of the local stack variables. A pointer to a `struct tcp_sock *tp` is derived from the general-purpose socket in order to have the TCP Stealth related state information available. Afterwards, a pointer to the TCP header within the received SYN segment is assigned to the `struct tcphdr *th`. Finally space for two big endian 32-bit words is allocated: The first `__be32 isn` holds a copy of the ISN received by the server in network byte order and the second `__be32 hash` will be used to store the authenticator derived from the TCP Stealth parameters.

As (according to sections 3.3 and 3.4) the timestamp is needed to calculate the AV, the TCP header options of the received SYN segment need to be parsed. This is done in line 9 by calling `tcp_parse_tsval_option` which places the result (the TSVal field of the timestamp option) in host byte order in the `tsval` field of the `struct stealth` associated with the TCP socket `tp`.

The one-setting of the `TCP_STEALTH_MODE_INTEGRITY_LEN` bit in the current TCP Stealth operation mode indicates that integrity protection should be performed – thus the second 16-bit word of the ISN is used as the integrity hash IH for payload protection once the first segment carrying data arrives. This is done in lines 11 to 12.

The core of the function is to verify the correctness of the AV. Lines 14 to 34 first check if the current socket uses IPv4 or IPv6 as network layer protocol and call `tcp_stealth_sequence_number` in order to compute the expected AV from the TCP Stealth parameters (destination IP and port of the incoming SYN segment, timestamp and PSK). The parameters for the called function are derived from the the IP and TCP headers of the incoming packet/segment using the `ip_hdr` and `ipv6_hdr` macros provided by the kernel. In case of a network layer protocol different from IPv4 or IPv6, a warning is printed into the local kernel log file (dmesg) and the authentication consequently fails by returning one.

The result of the computation (AV) stored in `hash` is then checked against the AV sent by the client. If integrity protection is enabled, only the first 16-bit words of the calculated and the received AV are compared. Otherwise, if only authentication should be provided, the full 32-bits of the AVs are computed. If the respective comparison yields true, zero is returned to signal a successful authentication to the calling function. Lines 36 to 44 implement the compare logic. If the function end is reached, one is returned in order to indicate that authentication has failed.

```
 1 u32 tcp_stealth_do_auth(struct sock *sk, struct sk_buff *skb)
 2 {
 3
 4         struct tcp_sock *tp = tcp_sk(sk);
 5         struct tcphdr *th = tcp_hdr(skb);
 6         __be32 isn = th->seq;
 7         __be32 hash;
 8
 9         tcp_parse_tsval_option(&tp->stealth.tsval, th);
10
11         if (tp->stealth.mode & TCP_STEALTH_MODE_INTEGRITY_LEN)
12                 tp->stealth.integrity_hash = be16_to_cpu(be32_isn_to_be16_ih(isn));
13
14         switch (tp->inet_conn.icsk_inet.sk.sk_family) {
15 #if IS_ENABLED(CONFIG_IPV6)
16         case PF_INET6:
17                 hash = cpu_to_be32(tcp_stealth_sequence_number(sk,
18                                    ipv6_hdr(skb)->daddr.s6_addr32,
19                                    sizeof(ipv6_hdr(skb)->daddr.s6_addr32),
20                                    th->dest, tp->stealth.tsval));
21         break;
22 #endif
23         case PF_INET:
24                 hash = cpu_to_be32(tcp_stealth_sequence_number(sk,
25                                    &ip_hdr(skb)->daddr,
26                                    sizeof(ip_hdr(skb)->daddr),
27                                    th->dest, tp->stealth.tsval));
28         break;
29         default:
30                 /* We don't authenticate unknown network layer protocols */
31                 printk("TCP Stealth: WARNING - Unknown network layer protocol,\n");
32                 printk("TCP stealth will not work as expected!\n");
33                 return 1;
34         }
35
36         if (tp->stealth.mode & TCP_STEALTH_MODE_AUTH &&
37             tp->stealth.mode & TCP_STEALTH_MODE_INTEGRITY_LEN &&
38             (be32_isn_to_be16_av(isn) == be32_isn_to_be16_av(hash)))
39                 return 0;
40
41         if (tp->stealth.mode & TCP_STEALTH_MODE_AUTH &&
42             !(tp->stealth.mode & TCP_STEALTH_MODE_INTEGRITY_LEN) &&
43             (isn == hash))
44                 return 0;
45
46         return 1;
47 }
```

Listing 3: *Source code of the function* `tcp_stealth_do_auth`

__tcp_stealth_integrity_check This function is called by the TCP Stealth server in order to ensure the integrity of the first bytes sent by the client. This is done by checking if the IH meets the requirements imposed by section 3.4. Table 4 shows the summary of of the function.

In the local variables area a pointer to the TCP header within the received data segment is assigned to the `struct tcphdr *th`. Afterwards a pointer to a `struct tcp_sock *tp` is derived from the general-purpose socket in order to have the TCP Stealth related state information available. The variable `__be32 seq` filled with a copy of the next sequence

`tcp_stealth_integrity_check`		
Declaration	net/ipv4/tcp_input.c	
Definition	net/ipv4/tcp_input.c	
Xrefs	`tcp_data_queue`, `tcp_rcv_established`	
Callee	`tcp_stealth_integrity`	
Parameters	`struct sock *sk`	a pointer to a TCP Stealth enabled receiving socket `struct sock`
	`struct sk_buff *skb`	a pointer to a `struct sk_buff` which holds the contents of the first data segment sent by the client
Return Value	zero if integrity checking was successful, nonzero otherwise	

Table 4: *Summary of function* `__tcp_stealth_integrity_check`

number expected from the client and decremented by one. This is done because the value of the original ISN carries the integrity hash and the sequence number of the first data segment always corresponds to ISN+1 (compare Figure 1) the ISN can be reconstructed from the expected sequence number by subtracting one. The pointer `char *data` is initialized to point at the beginning of the TCP payload before the length of the received data segment `int len` is determined by subtracting the header size in bytes (thus the multiplication by 4) from the total size of the segment.

Lines 10 to 11 check if the number of payload bytes is sufficiently large to contain at least as many bytes as should be integrity protected by TCP Stealth followed by a call to `tcp_stealth_integrity` which calculates the IH of the received payload. If the payload is too small or hashing does not succeed, one is returned to the caller to indicate that integrity checking failed. The comparison of the calculated IH and the original IH from the first SYN segment is evaluated in line 17: In case the second 16-bit word (in network byte order) of the sequence number does not match the calculated hash the function indicates the integrity check failure by returning 1 to the caller.

Finally, if the function end is reached, the integrity protection has been successful and therefore is deactivated such that TCP Stealth falls back to normal TCP.

```
 1 static int __tcp_stealth_integrity_check(struct sock *sk, struct sk_buff *skb)
 2 {
 3         struct tcphdr *th = tcp_hdr(skb);
 4         struct tcp_sock *tp = tcp_sk(sk);
 5         u16 hash;
 6         __be32 seq = cpu_to_be32(TCP_SKB_CB(skb)->seq - 1);
 7         char *data = skb->data + th->doff * 4;
 8         int len = skb->len - th->doff * 4;
 9
10         if (len < tp->stealth.integrity_len)
11                 return 1;
12
13         if (tcp_stealth_integrity(&hash, tp->stealth.secret, data,
14                                   tp->stealth.integrity_len))
15                 return 1;
16
17         if (be32_isn_to_be16_ih(seq) != cpu_to_be16(hash))
18                 return 1;
19
20         tp->stealth.mode &= ~TCP_STEALTH_MODE_INTEGRITY_LEN;
21         return 0;
22 }
```

Listing 4: *Source code of the function* `__tcp_stealth_integrity_check`

tcp_parse_tsval_option This function is called by the TCP Stealth server in order to determine if the incoming SYN segment contains a timestamp optional header and if so, returns the value of the `TSVal` field.

`tcp_parse_tsval_option`		
Declaration	include/net/tcp.h	
Definition	net/ipv4/tcp_input.c	
Xrefs	`tcp_stealth_do_auth`	
Callees	none	
Parameters	`u32 tsval *`	a pointer to a memory location where the parsed timestamp will be stored
	`struct tcphdr *th`	a pointer to the TCP header of the segment that should be parsed
Return Value	false (zero) if no timestamp option could be found, true (nonzero) otherwise	

Table 5: *Summary of function* `tcp_parse_tsval_option`

The function in Listing 5 starts with determining the size of the TCP options `int lenght` in the segment to be parsed. This is done by subtracting the (fixed) TCP header size from the data offset measured in bytes (thus the multiplication by 4). The pointer `const u8 *ptr` is inizialized to reference the first byte after the TCP header (note how the cast is applied *after* the increment of one).

In the case of the area of TCP options being smaller than the size of a TCP timestamp option, the function immediately returns false.

Lines 10 to 33 form a loop which successively reads the current byte to be parsed from the TCP options area followed by an increment of the pointer.

Sensible TCP options are defined to start with a unique one byte identifier (referred to as `int opcode` followed by the size of the option including the identifier in bytes. This rule applies to all TCP options but the special option `TCPOPT_NOP` which has no special meaning and never is followed by a size parameter as well as the option `TCPOPT_EOL` which denotes the end of the list of TCP options. The code in lines 15 to 19 skips all `TCPOPT_NOP` options and returns immediately if the end of the list option is encountered.

In case the `opcode` denotes a TCP timestamp option (`TCPOPT_TIMESTAMP`) the size parameter is constrained to two properties: First, it needs to be exactly the size of the `TCPOPT_TIMESTAMP` and second it may not be larger than the amount of remaining bytes that still need to be parsed. If these two requirements are met, the code dereferences the location behind the `opsize` parameter to a 32-bit word in network byte order, converts it to host byte order (using the macro `get_unaligned_be32`) and returns execution to the caller indicating that the timestamp value has been parsed successfully.

If the parser finds a option which does not equal any of the three cases above, it simply reads the size parameter of the TCP option, validates that it is greater than two (the minimum size of a TCP option consisting only of `opcode` as well as `opsize`) and also smaller or equal to the amount of bytes that still wait to be parsed. If this is applicable then the pointer is increased by the size of the unknown option (in order to skip to the next option) and the number of remaining bytes is adjusted.

The while loop is exited as soon as there are no more bytes that remain to be parsed (`length >= 0`) and returns false as the control flow never reaches the function end if the option list contains a (benign) timestamp.

```
const bool tcp_parse_tsval_option(u32 *tsval, const struct tcphdr *th)
{
        int length = (th->doff << 2) - sizeof(*th);
        const u8 *ptr = (const u8 *)(th + 1);

        /* If the TCP option is too short, we can short cut */
        if (length < TCPOLEN_TIMESTAMP)
                return false;

        while (length > 0) {
                int opcode = *ptr++;
                int opsize;

                switch(opcode) {
                case TCPOPT_EOL:
                        return false;
                case TCPOPT_NOP:
                        length--;
                        continue;
                case TCPOPT_TIMESTAMP:
                        opsize = *ptr++;
                        if (opsize != TCPOLEN_TIMESTAMP || opsize > length)
                                return false;
                        *tsval = get_unaligned_be32(ptr);
                        return true;
                default:
                        opsize = *ptr++;
                        if (opsize < 2 || opsize > length)
                                return false;
                }
                ptr += opsize - 2;
                length -= opsize;
        }
        return false;
}
```

Listing 5: *Source code of the function* `__tcp_parse_tsval_option`

4.1.3 Modified Functions

In summary, nine functions are modified by the patch, four to establish the client's functionality, four at the server side and the handler of the *setsockopt* system call.

Modifications to the Client TCP Stack In order to generate ISNs for outgoing connections as required by TCP Stealth, function calls are added to the functions `tcp_v4_connect` and `tcp_v6_connect`. In case at least TCP Stealth authentication is enabled on the socket, the normal generation algorithm `secure_tcp_sequence_number` is replaced by a call to `tcp_stealth_sequence_number`. As one of the parameters of the ISN generation algorithm, the exact value of the TCP timestamp value that is going to be added to the SYN segment needs to be known. In Linux, the TCP timestamp is simply a snapshot of the current uptime in jiffies (see [18], Chapter 10) - consequently, the taking of the snapshot needs to be done before the ISN is generated. The current implementation in the Linux kernel takes the snapshot right when the SYN packet is put on the wire, which is too late. To complicate things, the kernel may decide that it sends out the SYN *without* a timestamp option even though it would be available. Precisely, the timestamp option is only appended to the TCP header if the related *sysctl*[3] is set to one and if the TCP MD5 signature option [12] is

[3] `/proc/sys/net/ipv4/tcp_timestamps`

disabled for the current socket. Therefore the patch introduces code which always stores the TCP timestamp snapshot in `struct stealth` and then performs the checks in order to decide if the TCP timestamp option is going to be used and consequently either passes the timestamp value or a zero constant to the `tcp_stealth_sequence_number` function.

The fact that the timestamp value has to be known as early as possible to TCP Stealth leads to the modification of the function `tcp_connect`: As the timestamp is taken *before* ISN generation for TCP Stealth sockets, the kernel should always use the timestamp value held by `struct stealth` at the original point where it would normally obtain the snapshot for the TCP timestamp.

Another modification related to TCP timestamps concerns retransmissions. If the SYN segment of a connection request does not reach the destination host the TCP stack triggers a SYN retransmit. Retransmissions are implemented by a timer callback function named `__tcp_retransmit_skb`. In case of a SYN retransmit the kernel will generate a new timestamp for the retransmitted SYN segment which still holds the ISN of the original connection attempt potentially breaking TCP Stealth. The patch edits the retransmission function to check if the socket on which the SYN loss occurred operates in TCP Stealth mode. If appropriate, the original timestamp from `struct stealth` is reused, if not, the new SYN segment is constructed to contain a fresh timestamp.

Modifications to the server TCP stack The additions to the server are twofold: The first piece enables authentication directly after receiving a connection request from a client, the other piece concerns integrity protection at the time the first data segment arrives.

The code which is responsible for accepting connections is located in `tcp_v4_do_rcv` and `tcp_v6_do_rcv` for IPv4 and IPv6 respectively. The patch therefore adds code to these two function which checks if the socket is in `LISTEN` state, if the SYN flag in the received payload is set and if TCP Stealth is activated on the socket. If all of these conditions are met the server checks the authenticator by calling `tcp_stealth_do_auth` and depending on the result either immediately triggers a reset (authentication failed) or lets the control flow proceed (authentication successful).

The remaining two modifications both concern integrity checking. The kernel waits for the first data from the client and checks via `__tcp_stealth_integrity_check` if the received data hashes to the expected integrity protector value. The reason why we need two modifications to achieve this is that the TCP implementation in the kernel is highly optimized: It features a fast path which only handles the most common events that may occur during a TCP session and falls back to a slow path only if data is sent which the fast path cannot handle. The integrity check for the fast path is placed in function `tcp_rcv_established` whereas the check for the slow path is placed in `tcp_data_queue`. The inserted lines of code are equivalent and simply check if the receiving socket has the TCP Stealth integrity protection mode enabled and call the function named at the beginning of this paragraph in order to verify the integrity of the payload. If the verification fails, the packet is dropped and the connection is reset.

Modifications to the setsockopt system call The setsockopt handler is extended to recognize three new socket options:

1. The `TCP_STEALTH` option to specify the TCP Stealth secret. It first checks if the user supplied at least as many bytes as the `MD5Transform` function needs to operate on (see section 3) and copies exactly one block of data (`MD5_MESSAGE_BYTES` bytes) into

the `struct stealth` associated with the socket on which the setsockopt was called on. To prevent potential race conditions, the data is first copied into a local buffer on the stack, then the socket is locked by a mutex, then the data is copied into the buffer in `struct stealth`, then the flag that signals the TCP Stealth authentication mode is set on the socket finally followed by the unlocking of the socket.

2. The `TCP_STEALTH_INTEGRITY` option to specify the TCP Stealth payload which first locks the socket followed by checks if authentication is enabled on the socket and if the length of the supplied payload is bound by sensible values. Afterwards a memory block is allocated and the user payload is copied in. The function `tcp_stealth_integrity` calculates the integrity hash from the supplied payload and assigns the result to the `struct stealth`. Eventually, the integrity protection mode bit is set in the TCP Stealth state information before the allocated memory is freed and the socket is unlocked.

3. The `TCP_STEALTH_INTEGRITY_LEN` to activate integrity checking on the server side. This setsockopt handler sets `TCP_STEALTH_MODE_INTEGRITY_LEN` flag as the operation mode of TCP Stealth and assigns the number of bytes to protect to `struct stealth`. Note that no locking is done in the handler itself as there is a combination of `lock_sock` and `release_sock` calls surrounding the whole *switch* statement.

4.1.4 New Compiler Macros

Besides the function executed at run-time the patch also adds several compiler macros:

1. The constants passed to `setsockopt` `TCP_STEALTH`, `TCP_STEALTH_INTEGRITY` and `TCP_STEALTH_INTEGRITY_LEN` are defined as three consecutive numbers following the last defined `setsockopt` option number. At the time of writing (kernel 3.16) the values 26 to 28 are used.

2. The macros `be32_isn_to_be16_av` and `be32_isn_to_be16_ih` which extract the authentication vector AV or the integrity hash IH from a given sequence number in network byte order. As the names of the constants indicate, the result also is in network byte order.

4.1.5 Additional Modifications

Amongst the insertion of new functions and the modification of existing functions, the patch adds two more code snippets.

The first one is a kernel structure representing a TCP socket (`struct tcp_sock`) is extended to hold additional TCP Stealth information as shown in Listing 6.

```
struct {
        #define TCP_STEALTH_MODE_AUTH              BIT(0)
        #define TCP_STEALTH_MODE_INTEGRITY         BIT(1)
        #define TCP_STEALTH_MODE_INTEGRITY_LEN     BIT(2)
        int mode;
        u8 secret[MD5_MESSAGE_BYTES];
        size_t integrity_len;
        u16 integrity_hash;
        u32 tsval;
} stealth;
```

Listing 6: *Definition of the* `struct stealth` *structure in the kernel representing TCP Stealth state information*

The names of the members of the structure are more or less self-explanatory. The `int mode` is a bitfield which indicates if only authentication or additionally integrity checking is desired. The corresponding possible values for `mode` are defined directly above where `TCP_STEALTH_MODE_INTEGRITY` and `TCP_STEALTH_MODE_INTEGRITY_LEN` are mutually exclusive as they define if the associated socket acts as a TCP Stealth client or server, respectively. The array `u8 secret` and the variables `size_t integrity_len`, `int integrity_hash` and `u32 tsval` are the locations where the TCP Stealth secret, the number of bytes to be integrity protected, the integrity hash itself and the (received or generated) timestamp are stored.

The second (and last) snippet that is being inserted by the patch is a new kernel configuration option called `TCP_STEALTH`. Using this option the user can choose whether the kernel should be compiled with TCP Stealth support enabled. The option can be found in `Networking Support > Networking Options > TCP/IP networking > TCP: Stealth TCP socket support` – by default the kernel is compiled without TCP Stealth support.

4.2 User Space

As mentioned earlier, there also exists user space code called *libknockify*. The user land part is available in form of a dynamic library which can be used at compile or run time to enable applications to use TCP Stealth without changing a single line of code. For the usage of this library please refer to section 6.3, this section only outlines the implementation details of the library itself.

4.2.1 Design

The design of *libknockify* handles two scenarios: The case in which TCP Stealth authentication should be added to a program and the more complex case in which additionally integrity protection should be added.

The first case wraps the `socket` call of the network API such that after the creation of a TCP socket `setsockopt` with option `TCP_STEALTH` is called. The library passes the user defined secret to the kernel in order to activate the authentication mode of TCP Stealth. This applies symmetrically to the client and the server side.

The second case additionally aims to provide integrity protection. With reference to the designated `setsockopt` call the reader may well ask how the application can specify payload to be protected that is not known by the time the socket is not even connected yet. The trick here is to wrap the `connect` call such that it indicates success to the application even though the connect internally never has taken place. By the time the application starts sending data over the allegedly connected socket, *libknockify* buffers the data. As soon as a sufficient amount of data has been buffered, *libknockify* executes `setsockopt` with option `TCP_STEALTH_INTEGRITY` and specifies the buffered payload to the kernel. Afterwards it catches up on calling `connect` and `send` to complete the activation of TCP Stealth authentication and integrity protection.

A TCP Stealth server which is to communicate with integrity protection enabled calls `setsockopt` with option `TCP_STEALTH_INTEGRITY_LEN` in the wrapper function of the `listen` call.

4.2.2 Overridden Functions

All in all, ten functions are wrapped with the following (briefly described) functionality:

1. A call to `setsockopt` is made after the `socket` call to enable authentication and the socket is added to the list of TCP Stealth sockets.
2. `connect` pretends that the connect has taken place even though it is delayed internally if integrity protection should be activated.
3. `listen` is prepended by a `setsockopt` if integrity protection is desired.
4. The `send` call is responsible for buffering data received from the application (if a `connect` is delayed) and for performing the actual connect as soon as enough data has been provided.
5. `write` is emulated by `send`.
6. `sendto` is emulated by `send`.
7. `getsockopt` is modified to always return a zero (no error) if the application tries to determine the error state of a socket on which a `connect` has been delayed.
8. If the application calls `close` on a socket it is removed from the list of TCP Stealth sockets.
9. The result of the `epoll_wait` call is filtered to indicate that any socket with a delayed `connect` is ready to be written to and not ready to be read from
10. The result of the `select` call is filtered to indicate that any socket with a delayed `connect` is ready to be written to and not ready to be read from.

The functions overridden by *libknockify* are listed and described in more detail in Table 6. For even more details, refer to the following section containing a full flow chart which depicts the inner workings of *libknockify*.

4.2.3 Flow Chart

The flow charts depicted in Figures 29 and 30 provide further details on the implementation of *libknockify*. The usual elements are used: Boxes indicate states whereas diamonds symbol decisions within the control flow. Unconditional jumps are colored black, the branches of a conditional jump are colored red and green. Control is returned to the application during the phase between the yellow boxes.

Function	Wrapper Functionality
`socket`	executes the original call, checks if the newly created socket is a TCP socket, calls `setsockopt` and adds the socket to the list of TCP stealth sockets
`connect`	checks if the socket belongs to the list of TCP Stealth sockets, executes the original call if integrity protection is disabled or stores the function arguments and omits the `connect` otherwise
`listen`	checks if the socket belongs to the list of TCP Stealth sockets, executes `setsockopt` if integrity protection should be enabled, finally executes the original call
`write`	emulated by calling `send`
`send`	checks if the socket belongs to the list of TCP Stealth sockets and if integrity protection is active, returns minus one if there is no delayed connect on the socket, stores the parameters of `send` for later transmission, executes the delayed connect if enought data has been provided, finally executes the original call
`sendto`	emulated by calling `send`
`getsockopt`	checks if the socket belongs to the list of TCP Stealth sockets, returns zero if the error state of the socket is queried, executes the original `getsockopt` otherwise
`close`	checks if the socket belongs to the list of TCP Stealth sockets and removes it from the list if applicable, finally executes the original call
`epoll_wait`	executes the original call, loops over all sockets passed to the original function, marks all sockets as writeable only where all of the following is true: the socket belongs to the list of TCP Stealth sockets, a `connect` has been delayed on the socket and the socket has been marked readable or writeable by `epoll_wait`
`select`	executes the original call, loops over all sockets passed to the original function, marks all sockets as writeable only where all of the following is true: the socket belongs to the list of TCP Stealth sockets, a `connect` has been delayed on the socket and the socket has been marked readable or writeable by `select`

Table 6: *List of functions overridden by* libknockify

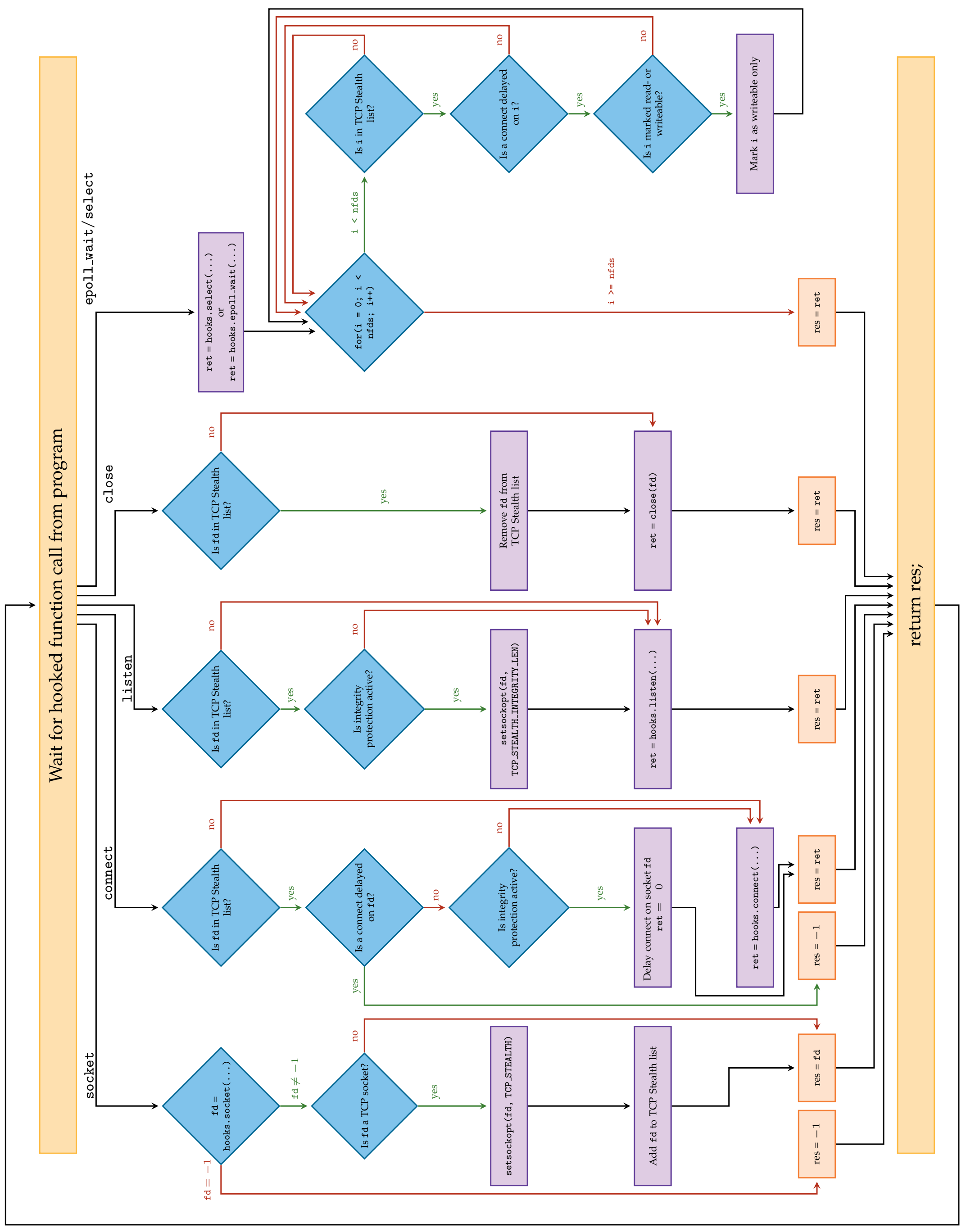

Wait for hooked function call from program
socket
connect
listen
close
epoll_wait/select
fd = hooks.socket(...)
fd = −1
fd ≠ −1
Is fd a TCP socket?
no
yes
setsockopt(fd, TCP_STEALTH)
Add fd to TCP Stealth list
res = −1
res = fd
Is fd in TCP Stealth list?
no
yes
Is a connect delayed on fd?
yes
no
Is integrity protection active?
no
yes
Delay connect on socket fd ret = 0
ret = hooks.connect(...)
res = −1
res = ret
Is fd in TCP Stealth list?
no
yes
Is integrity protection active?
no
yes
setsockopt(fd, TCP_STEALTH_INTEGRITY_LEN)
ret = hooks.listen(...)
res = ret
Is fd in TCP Stealth list?
no
yes
Remove fd from TCP Stealth list
ret = close(fd)
res = ret
ret = hooks.select(...) or ret = hooks.epoll_wait(...)
for(i = 0; i < nfds; i++)
i < nfds
i >= nfds
Is i in TCP Stealth list?
no
yes
Is a connect delayed on i?
no
yes
Is i marked read- or writeable?
no
yes
Mark i as writeable only
res = ret
return res;

Figure 29: *Control flow diagram for the* `socket`, `connect`, `listen`, `close`, `select` *and* `epoll_wait` *calls in* libknockify

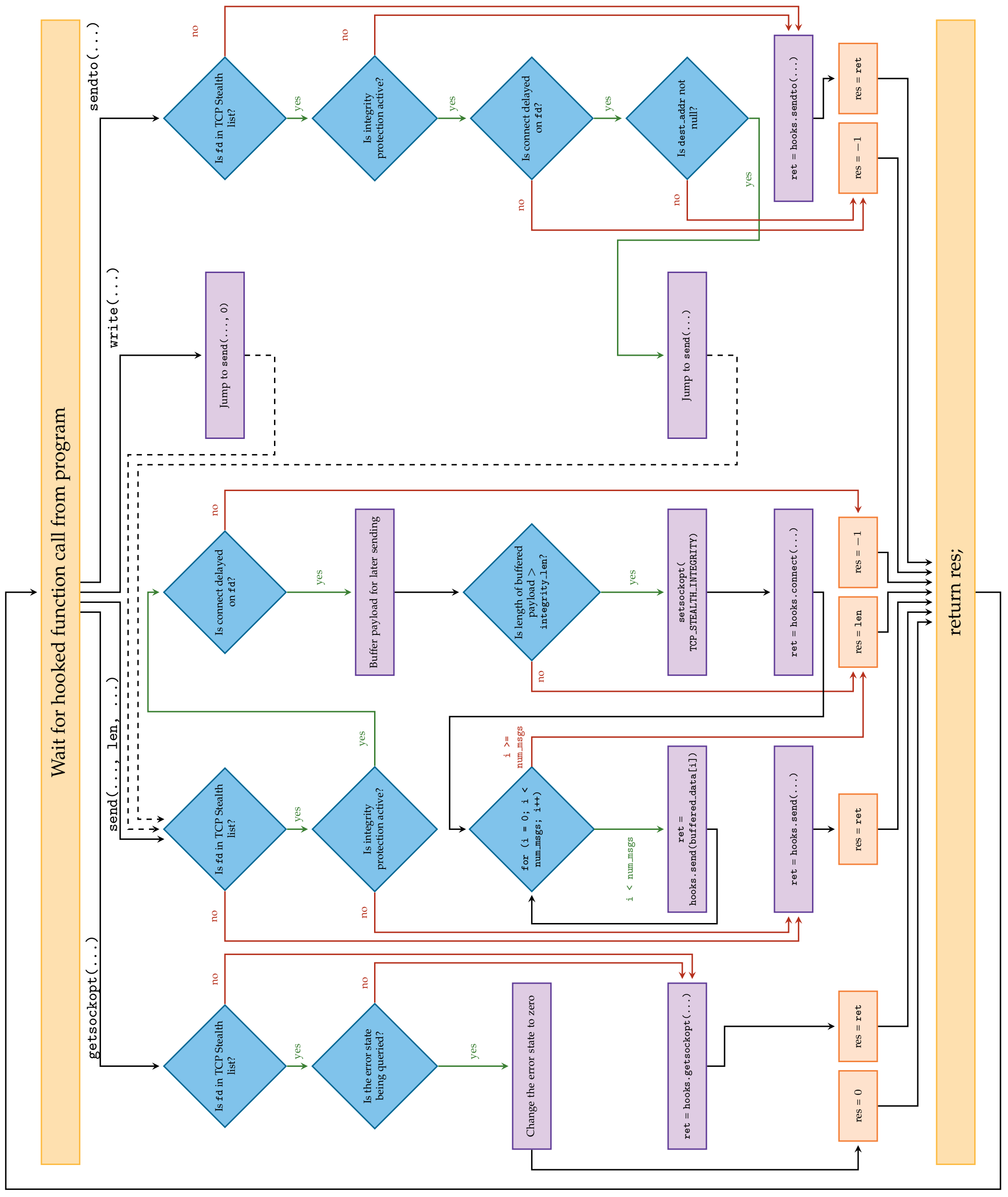

Wait for hooked function call from program
sendto(...)
write(...)
send(..., len, ...)
getsockopt(...)
Is fd in TCP Stealth list?
Is integrity protection active?
Is connect delayed on fd?
Is dest_addr not null?
ret = hooks.sendto(...)
res = ret
res = −1
Jump to send(..., 0)
Jump to send(...)
Buffer payload for later sending
Is length of buffered payload > integrity_len?
setsockopt(TCP_STEALTH_INTEGRITY)
ret = hooks.connect(...)
res = len
for (i = 0; i < num_msgs; i++)
i >= num_msgs
i < num_msgs
ret = hooks.send(buffered_data[i])
ret = hooks.send(...)
Is the error state being queried?
Change the error state to zero
ret = hooks.getsockopt(...)
res = 0
yes
no
return res;

Figure 30: *Control flow diagram for the* `getsockopt`, `send`, `sendto` and `write` *calls in* libknockify

5 Experimental Results

As outlined in Section 3.1 one of our goals is that TCP Stealth works well with existing middleboxes. As the essential operation of a NAT box [8] is to change the source IP address and (potentially) the source port for any packet that leaves the private network, the algorithms outlined in Section 3.3 and 3.4 do not incorporate the former in the ISN generation. However, NAT boxes (or middleboxes in general) are not bound to *only* change the aforementioned fields. It might be possible that some NAT implementations out there change any of the fields used in the ISN generation.

However, there are only certain special circumstances (such as tracking SIP and FTP protocols) under which there are good technical reason for NAT implementations to do this. The only NAT implementation that we know of which replaces ISNs independent of the application-layer protocol 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.

Aside from the use of NAT, transparent proxies may also cause the ISN to change. However, a transparent proxy is equivalent to a man-in-the-middle attacker, so one would probably not want TCP Stealth to support connections with such a device performing active interception anyway.

5.1 Prior Work

The behavior of NAT boxes with respect to modifying ISN numbers and TCP timestamps was previously studied in [13] by Honda et al. Honda et al. measure the fraction of ISN values that pass unmodified through the internet dependent on the destination port. The result can be found in Table 7. They state that 90% of the SYN segments within their data passes NAT boxes without modification to the ISN for the unassigned TCP port 34343. This share decreases to 85% and 73% when connecting to the well known ports 443 (https) and 80 (http) respectively.

5.2 Test Design

We also performed a similar experiment, asking users to run a small program that would assess the behaviour of the user's connection and report it to our server. Note that when we talk about NAT, we exclude DNAT where the middlebox changes the IP address or

	TCP Port		
Behavior	34343	80	443
Unchanged	126 (93%)	116 (82%)	128 (90%)
Mod. outbound	5 (4%)	5 (4%)	6 (4%)
Mod. inbound	0 (0%)	1 (1%)	1 (1%)
Mod. both	4 (3%)	13 (9%)	7 (5%)
Proxy (probably mod. both)	0 (0%)	7 (5%)	0 (0%)
Total	135 (100%)	142 (100%)	142 (100%)

Table 7: *Changes made to the ISN by middleboxes dependent on the destination port as measured by Honda et al. [13]*

TCP port of the destination. We also do not consider middleboxes that change the TCP payload. Our experiment focuses on middleboxes deployed near the client or by network providers (carrier-grade NAT). Thus, the goal of the experiment is simply to investigate if the ISN or the TCP timestamp option is changed while a segment travels through the Internet, before reaching the service provider.

In order to gain statistical relevance, we decided to run a public test and hence designed a test which could be run on different operating systems (GNU/Linux, Android, OSX, Windows) on different platforms.

The test sends out a TCP SYN packet (a probe) to one of our testing machines on the Internet (which are known to be connected via routers that do not manipulate the IP traffic) and extracts the following information from the packet as it is put onto the wire using `libpcap`:

- The source address of the IP header. We need this to determine if the packet as it is received by the server has been NATed at all.
- The ISN field of the TCP header
- The TSval field of the (optional) TCP timestamp header
- The Ethernet address of the network gateway which most likely performs the address translation, as we would like to be able to group the results by manufacturer of the NAT box and detect duplicate submissions. If a contributor for privacy reasons decides not to transmit the Ethernet address, she can signal this to the test program which then only transmits the *organizationally unique identifier* (OUI) [22] part of the Ethernet address.

This 4-tuple gets transmitted to the server which performs the following steps:

1. Listening for SYN packets sent to the designated port
2. Extraction of the received source IP, ISN and TSval fields
3. Extraction of the sent source IP, ISN, TSval and the gateway's Ethernet address from the first packet carrying the payload sent by the client

4. Checking if the probe is valid. This is done by calculating Fletcher's checksum [10] on the data to prevent random port scans and potential exploits carrying data intended for a different network service from polluting the results.

5. Checking if the source IP in the payload does not match the source IP observed by the server (i.e. that the probe has travelled through a NAT at all). If the source IP did not change, the probe and all data is discarded.

6. Performing a Geo IP lookup to get the originating country of the probe

7. Storing of the tuple T which holds the following information:

$$T = (\text{hwaddr}_{\text{sent}}, \text{ISN}_{\text{sent}}, \text{ISN}_{\text{recv}}, \text{TSval}_{\text{sent}}, \text{TSval}_{\text{recv}}, \\ \text{UNIX\_tstamp}, \text{checksum\_valid}, \text{geo\_string})$$

Afterwards the connection is closed as all relevant data has been collected.

5.3 Analysis of the Collected Data

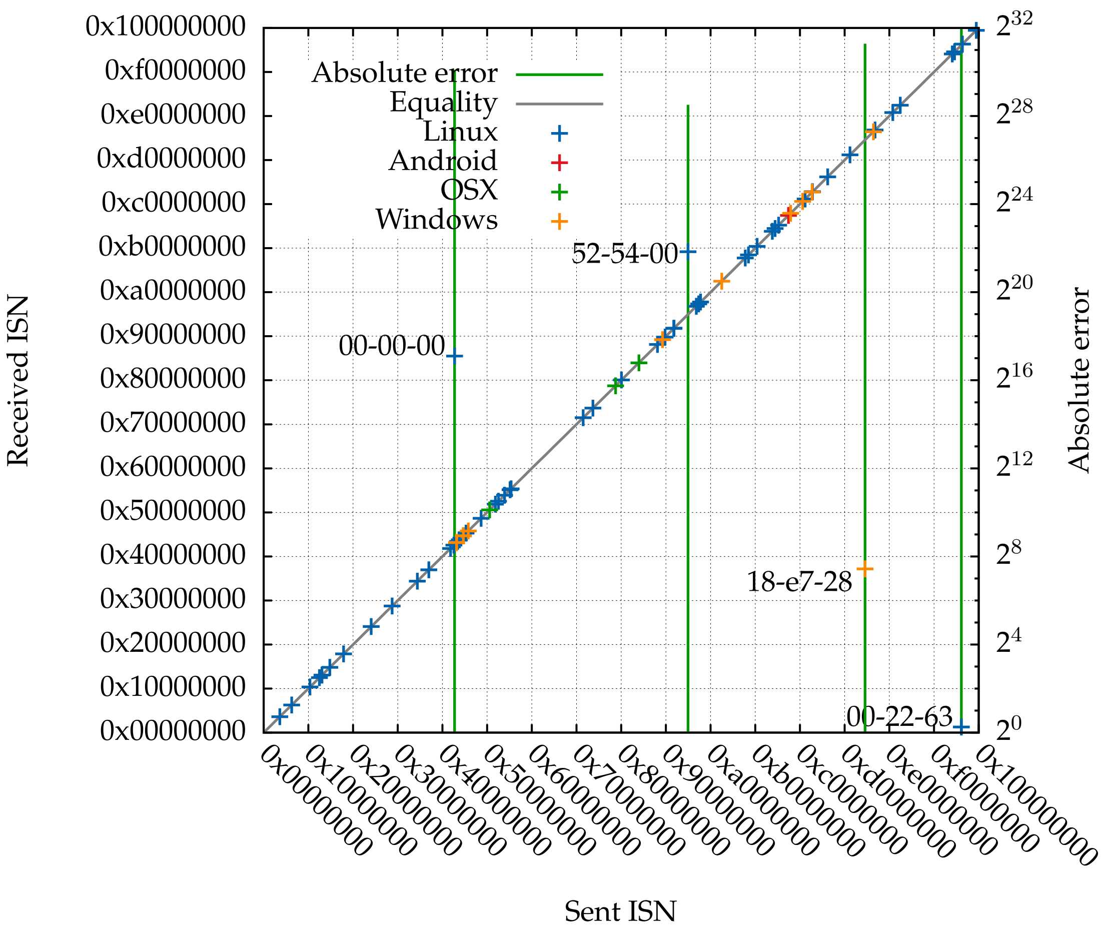


Figure 31: *Plot of the absulute value and the absolute error of the ISN of each collected sample*

Figure 31 shows the qualitative change of the ISN occurring to any of the collected probes. The x-axis denotes the ISN as it was sent by the client and the left y-axis denotes the ISN as it was received by the server. The logarithmic right y-axis measures the deviation of the sent and the received ISN. The black groups of hexadecimal digits right next to the error bars indicate the organizationally unique identifier (OUI) of the gateway of the sending side. Table 9 translates all OUIs found in this and the following graph to their associated names. As can be seen from the diagram, the observed ISNs approximately follow a uniform distribution across all possible values. Additionally, it can be observed that there are four probes which arrived with a changed ISN. The diagram suggests that there is no (obvious) correlation between the value of the sent ISN and the amount it deviates from the original value when received by the server.

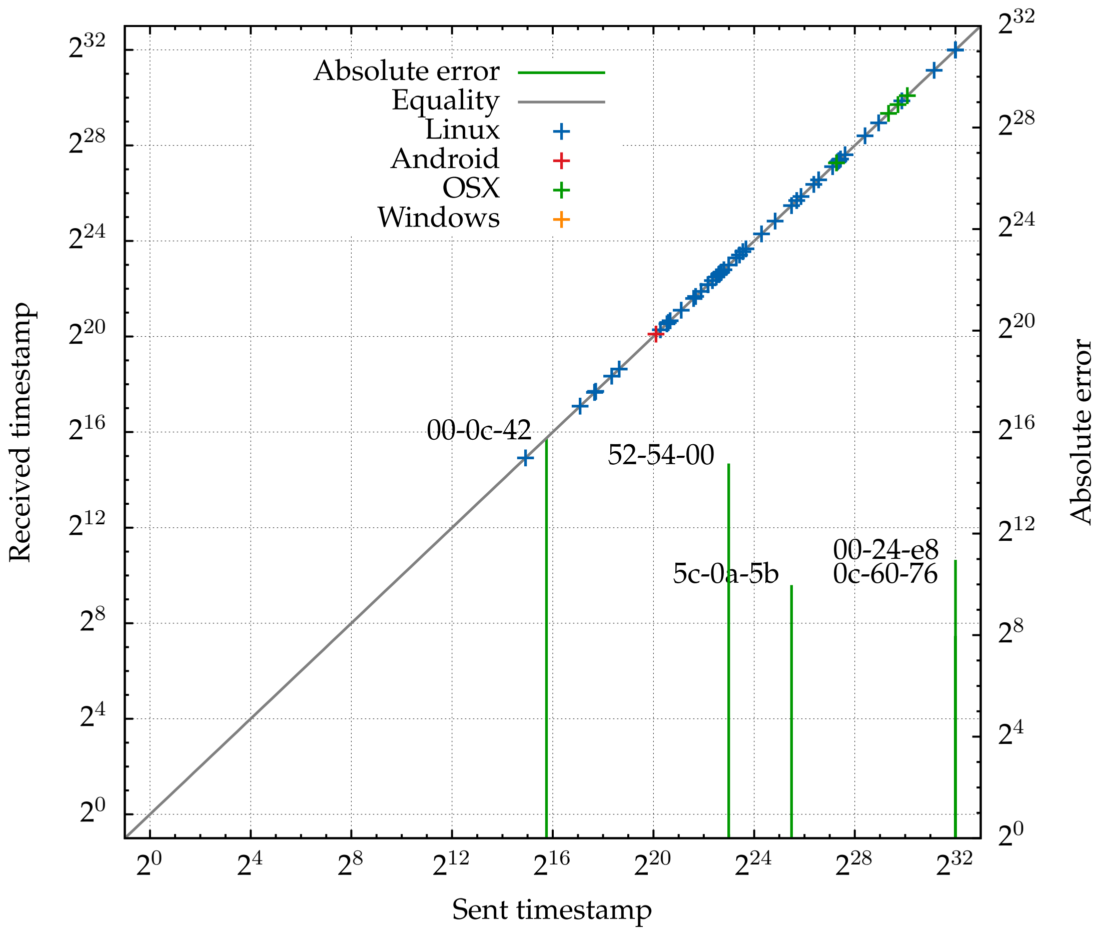


Figure 32: *Plot of the absolute value and the absolute error of the TSVal of each collected sample*

Similarly, Figure 32 shows how the timestamp value changed for the probes travelling through the internet (note the logarithmic scales). However, what's worth mentioning is that the average absolute error of the timestamp value ($17214 \approx 2^{14.1}$) is much lower than the absolute error observed for the ISN ($1660517875 \approx 2^{30.6}$). Additionally, the timestamp values are *not* uniformly distributed across all of the 2^{32} possible values. This is natural as the TCP timestamp correlates to the uptime of the sending host.

All in all, the collected data implies that TCP Stealth will work as expected when used

Country	None	ISN only	Stamp only	Both	Sum
Germany	22	1	1	1	25
France	6	1	2	0	9
United States	7	1	0	0	8
Hungary	4	0	0	0	4
Netherlands	4	0	0	0	4
Great Britain	2	0	0	0	2
Poland	2	0	0	0	2
Argentinia	1	0	0	0	1
Bermuda	1	0	0	0	1
Brazil	1	0	0	0	1
Switzerland	1	0	0	0	1
Czech Republic	1	0	0	0	1
Spain	1	0	0	0	1
Finland	1	0	0	0	1
Norway	0	0	1	0	1
Ireland	1	0	0	0	1
Italy	1	0	0	0	1
Russia	1	0	0	0	1
Slovakia	1	0	0	0	1
Unknown	2	0	0	0	2
Total	60	3	4	1	68

Table 8: *Observed changes made by middle boxes*

OUI	Organization
00-00-00	XEROX CORPORATION
00-0c-42	Routerboard.com
00-22-63	Koos Technical Services, Inc.
00-24-e8	Dell Inc.
0c-60-76	Hon Hai Precision Ind. Co.,Ltd.
18-e7-28	Cisco
52-54-00	*unknown*
5c-0a-5b	SAMSUNG ELECTRO-MECHANICS CO., LTD.

Table 9: *Observed OUI numbers and their associated organizations*

on the Internet. Table 8 shows that out of 68 collected samples, 3 (4%) arrived where solely the ISN changed, whereas 4 (6%) samples indicated only a modification of the timestamp. In one (additional) case (1%) both values changed. Consequently, 12% of the collected samples arrived with at least one of the relevant values being altered.

6 User Guide

This section is intended for users who wish to enable TCP Stealth on their machines. Below follow detailed instructions on how the kernel can be patched and how user programs can profit from the new functionality.

6.1 Patching the Kernel

As the mainline Linux currently does not offer support for *Knock*, the kernel of the machine which should be using *Knock* needs to be patched. Note, however that there is one distribution, *Parabola Linux* [4], which incorporated support for *Knock*. If you are running this distribution, you can simply install *Knock* using your favourite package manager.

For all other distributions, patching the kernel is straightforward:

1. First, obtain the sources of the desired kernel version from `https://www.kernel.org` if you intend to use a vanilla running kernel. Note that many distributions make adoptions to the kernel and therefore provide custom kernel sources, so one might want to check for the curstomized kernel sources.

2. Once the kernel sources are available download the suitable *Knock* patch from `https://gnunet.org/knock`. Note that if you intend to run a kernel version which is not explicitly listed on the *Knock* website, the best option is to try out the patches of the closest version provided (i.e. `tcp_stealth_3.10.diff` for all kernels up to 3.11 and `tcp_stealth_3.12.diff` for all kernels which are more recent than 3.11).

3. Change to the directory where the kernel sources reside (replace the `<your-version>`-part according to your selection of the kernel- and the patch-version) and apply the patches (you can find more information on how to apply and revert patches on the kernel source in the *kernel.org* archives [14]):

```
~$ cd linux-<your-version>/
~/linux $ patch -p1 < /path/to/knock/patch/tcp_stealth_<your-version>.diff
```

4. Get the configuration of the currently running kernel. There are two widely used methods which can be used interchangeably:

(a) *Debianoids* maintain a copy of the kernel configuration parameters in the `/boot` directory. You can copy the config to your current kernel sources using the following command:

```
~/linux $ cp /boot/config-$(uname -r) .config
```

(b) Many other distributions compile the kernel with the possibility to read the running kernel's configuration from the `/proc/ file system:`

```
~/linux $ zcat /proc/config.gz > .config
```

(c) `If none of the cases above applies for your distribution, you can try to use the default kernel configuration by entering`

```
~/linux $ make defconfig
```

`however do not expect a convincing kernel to result from this, as far as performance and stability are concerned.`

5. Choose the defaults for all configuration parameters which are not in your current configuration. A different kernel version might introduce new compile configuration options:

```
~/linux $ yes "" | make oldconfig
```

6. Enable *Knock* in your current configuration by selecting `Networking Support > Networking Options > TCP/IP networking > TCP: Stealth TCP socket support`:

```
~/linux $ make menuconfig
```

7. The kernel is now ready for compilation. Enter

```
~/linux $ make bzImage && make modules
```

to compile the kernel and all additional modules. Be prepared that this step can take a long time, especially on older (weaker) computers. If you have a machine with more than one processor core, you can adjust the number of build threads using the `-j` option to both `make` commands.

8. If compilation succeeded (it should), install the new kernel and all modules. Afterwards automatically create a new *initramdisk* for your newly compiled kernel. If you have sudo installed, enter

```
~/linux $ sudo make install && sudo make modules_install
```

otherwise enter the these commands into a root prompt leaving both `sudo`s.

9. Once the kernel is installed and modules are copied over make sure that the `TCP_STEALTH` constants are exported to the userspace such that `setsockopt` knows about the new functionality of the kernel. As above, make sure you have root privileges and enter

```
~/linux $ sudo make headers_install INSTALL_HDR_DIR=/usr
```

to install the headers into the `/usr/include/linux` directory. If, for some reason, the headers can or should not be installed, simply copy the definition of the three `TCP_STEALTH` socket options from `linux-<your-version>/include/uapi/linux/tcp.h` to `/usr/include/linux/tcp.h`.

10. Reboot the machine and instruct your boot manager to boot into the new kernel. You now have a *Knock* aware machine. In order to understand how to use *Knock* in userspace, please refer to Section 6.2 of this document.

6.2 Minimal Knock Program

This section introduces minimal examples of a server and a client which make use of *Knock*'s basic functionality.

6.2.1 Minimal Knock Server

A minimal server program which makes use of *Knock*'s features needs to call *setsockopt* twice: Once to enable authentication and a second time to deliver the number of payload bytes for integrity protection. Listing 7 shows the relevant part of a minimal server. Code for a full *Knock* server can be found in Listing 9 in Appendix A. (The definitions of the two constants used in *setsockopt* can also be found there.)

```
	char secret[64] = "This is my magic ID.";
	int payload_len = 4;

	sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if (sock < 0) {
		printf("socket() failed, %s\n", strerror(errno));
		return 1;
	}

	if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH, secret, sizeof(secret))) {
		printf("setsockopt() failed, %s\n", strerror(errno));
		return 1;
	}
	if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH_INTEGRITY_LEN,
			&payload_len, sizeof(payload_len))) {
		printf("setsockopt() failed, %s\n", strerror(errno));
		return 1;
	}

	if (bind(sock, (struct sockaddr*) &addr, sizeof(addr))) {
		printf("bind() failed %s\n", strerror(errno));
		return 1;
	}

	if (listen(sock, 10)) {
		printf("listen() failed, %s\n", strerror(errno));
		return 1;
	}

	client = accept(sock, NULL, 0);
```

Listing 7: *Minimal Knock server (truncated)*

The sequence of the *socket*, *bind*, *listen* and *accept* *libc*-calls should be familiar. The only part which is relevant for the functioning of *Knock* are the two *setsockopt* calls. The `TCP_STEALTH` socket option is used to tell *Knock* the pre-shared secret which will be used for authentication. As can be seen from the code, the secret needs to be a block of data consisting of exactly 64 bytes. In this example the secret string “This is my magic ID” is being used. Only applications which know the secret string can thus connect to this server. In a real-world-scenario, the secret string could be set to something more secure such as a byte sequence but it is important to notice that this is entirely the freedom of the application programmer.

If only authentication of the communicating nodes should be guaranteed, no further socket options need to be set. However, if additionally integrity checking is desired, *Knock* also needs to be told the number of bytes which should be verified upon the arrival of the first data packet. We explicitly state that any sane protocol should start with the exchange of an ephemeral key in which case the server application would verify all bytes of the key that is sent in the first packet of the client. To enable integrity checking the programmer thus needs to hand over the expected key length of the protocol to *Knock* by calling *setsockopt* with the parameter `TCP_STEALTH_INTEGRITY_LEN`. We strongly encourage any programmer who plans to use *Knock* to provide a key which is longer than the one the example shows (4 bytes).

6.2.2 Minimal Knock client

All clients which should be capable of connecting to a *Knock*-protected service need to call *setsockopt* twice. Similar to the server, the first call enables authentication whereas the second call delivers the actual payload to the kernel to enable integrity checking. Naturally, if integrity checking is a requirement then both – the client and the server – need to execute the correct *setsockopt* calls as shown in Section 6.2.1. The core sequence of *libc* calls is shown in Listing 8, for the full listing, the reader is asked to refer to Appendix A, Listing 10.

```
        char secret[64] = "This is my magic ID.";
        char payload[4] = "1234";

        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock < 0) {
                printf("socket() failed, %s\n", strerror(errno));
                return 1;
        }

        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH, secret, sizeof(secret))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }
        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH_INTEGRITY, payload,
                        sizeof(payload))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }

        if (connect(sock, (struct sockaddr*) &addr, sizeof(addr))) {
                printf("connect() failed %s\n", strerror(errno));
                return 1;
        }
```

Listing 8: *Minimal Knock client (truncated)*

In Listing 8, the usual *libc* calls *socket* and *connect* can be observed. After creating the socket, the first *setsockopt* specifies the pre-shared secret using the *Knock*’s `TCP_STEALTH` socket option number. As explained earlier, the secret needs to be a location in memory holding exactly 64 bytes. As *Knock* authentication checking is performed symmetrically by the server and the client, the *Knock* secret on the client side needs to match the secret of the server.

If additionally integrity checking should be employed, *setsockopt* with option `TCP_STEALTH_INTEGRITY` needs to be called secondly. This call provides the data whose integrity should be protected. In a real-world example, an application would put an ephemeral key here in order to protect it against alteration. However, the example defines the data `1234` to be integrity protected. Two consequences follow:

1. The first message the client is to send out using one of the designated *libc* calls must start with those four bytes
2. The server side needs to be told that it should verify exactly four bytes of the first payload data received from the client by using the *setsockopt* option `TCP_STEALTH_INTEGRITY_LEN`.

Afterwards, the application can communicate as desired.

6.3 Enabling Knock Using LD_PRELOAD

Please note that the modification of the application itself is still preferable to the method explained in this section. Minimal code examples which make use of *Knock* are given in sections 6.2.1 and 6.2.2. These could be ported easily to existing code.

As mentioned earlier, *Knock* can also be used without having to modify the source code of the program. This can be useful in cases where the source code is not available or when inserting the needed *libc* calls is infeasible (for example due to restrictions imposed by the application logic).

In order to use *Knock* in existing applications, a dynamic library is provided which is referred to as *libknockify* (`libknockify.so`). *libknockify* can either be used at compile time during linking or alternatively at runtime overriding the basic network API calls. Implementation specific details (such as a list of overridden functions) are given in section 4.2, this section only outlines how *libknockify* can be used by the end user.

6.3.1 The GNU linker

In GNU/Linux, the GNU linker *ld*[4] is responsible to resolve external symbols used by applications at run time (see Section 4.2 for details). During resolving *ld* supports overriding arbitrary symbols with arbitrary code from a chosen shared library. Overriding is done using the *LD_PRELOAD* environment variable which is read by *ld* to determine the shared object file which contains the declarations and definitions of the functions which should be overridden Section 4.2 covers the implementation specific details of an `LD_PRELOAD` object.

6.3.2 Basic Usage

The basic usage of the *libknockify* shared object to enable *Knock* for program `example_program` is thus as follows:

```
$ LD_PRELOAD=./knockify.so ./example_program
```

[4]`https://www.gnu.org/software/binutils/`

Name	Mandatory	Description
`KNOCK_LOGLVL`	No	Defines how much output `libknockify` should produce. Choose a value between 0 and 3 (inclusive) where three causes `libknockify` to print out out all possible messages and zero silences `libknockify` completely.
`KNOCK_SECRET`	Yes	Specifies the secret that should be used by `libknockify` to authenticate any connection initiated or accepted by the application. Four formats are supported: 1. `h:([0-9a-fA-F][0-9a-fA-F])+` Specifies the secret as a string of hexdigits 2. `f:file` Specifies a secret as the contents of file `file` 3. `No special prefix` Specifies a secret as raw bytes of a zero terminated string If by any of the above methods a secret of inappropriate size is specified, the given secret is either padded with zeroes or truncated after the 64th byte.
`KNOCK_INTLEN`	No	Specifies how many bytes of the first message of any communication should be protected. A zero is equivalent to running *Knock* with integrity checking disabled. The upper bound is protocol specific: A number greater than the minimal amount of data which the client is to send before the server is expected to answer can stall the connection (see section 3 to understand this behavior).

Table 10: *Configuration Options supported by* `libknockify`

If the application `example_program` communicates in a way that *Knock* can support (see section 3 for details) and `libknockify` is configured correctly (see section 6.3.3) then `example_program` has been successfully enabled to use Knock.

6.3.3 Options Supported by `libknockify`

Generally, arguments for `libknockify` can be specified in two ways: Either using environment variables or line by line as key-value pairs of the format `KEY=VALUE` in a hidden file called `.knockrc` which (by default) should reside in the current user's home directory. Table 10 explains the implemented instances of `KEY` and their respective associated `VALUE` field.

7 Related Work

Our design is closely related to SilentKnock [30], 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.

However, 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 in this way.

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 [28] 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 [19] is yet another scheme which 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 implementation is called *Knock* as it only uses a *single* SYN packet.

8 Discussion

8.1 Hashing the Source Address

Presumably to increase the entropy of the generated ISN values, previous schemes such as SilentKnock [30] generally included the source IP and source port in the calculation of the ISN value. As a result, these designs break when clients are connecting to the server from behind a router performing network address translation [8], as clients behind NAT typically cannot easily predict the source address that will be seen by the server.

As a result, including the source address prevents about 90% of clients on the Internet today from connecting to a hidden server protected by SilentKnock. Furthermore, an adversary that is placed to observe the TCP SYN packet can typically easily fake the source address, thus including it offers little additional security beyond making the ISN less predictable.

Thus, TCP Stealth does not include the source address in the MAC, as this fails to improve security against realistic adversaries while seriously limiting usability. To avoid generating predictable ISN numbers, TCP Stealth uses the value of the TCP timestamp option in the hash instead of the (random) source port. While the TCP timestamp value might be predictable, the resulting ISN is still unpredictable as it is hashed with the shared secret — as long as the delay between connections is large enough to allow the timestamp clock to advance.

In conclusion, TCP Stealth in combination with either TCP timestamps or content integrity protections over random content will generate (pseudo-)random, non-constant ISNs for each connection. Only if neither the application payload nor timestamps nor changes in the shared secret are available as a source of entropy, ISNs would remain unchanged between connections.

8.2 Replay Attacks

Compared to SilentKnock [30] 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 as clients would somehow need to coordinate their use of shared secrets. Furthermore, having a large number of shared secrets increases the chance of success for an adversary to use a valid ISN by chance.

Naturally, TCP Stealth still needs to defend against replay attacks, and saying that we do not include sequence numbers should not mean that replay attacks are not considered.

However, to discuss replay attacks properly, we need to distinguish two types of replay attacks. Typically, a successful replay attack of a port knock achieves two goals: it confirms the existence of the service and allows the adversary to send arbitrary data to the service, for example to exploit a vulnerability. Confirming the existence of a service to an adversary who observed a successful TCP handshake is not particularly significant, so for TCP Stealth we consider a replay attack as successful only if the attacker can actually successfully communicate with the service.

Note that a typical replay attack defense using sequence numbers only works against an adversary that is merely able to observe network traffic. For TCP Stealth, we are considering a stronger adversary which is able to perform a man-in-the-middle attack after the TCP SYN packet, this is a strictly stronger attack than replaying the TCP SYN packet. The payload integrity protection provided by TCP Stealth also mitigates replay attacks — assuming the application-level protocol begins its payload with key material that is used to authenticate and integrity-protect the rest of the data exchange.

Alternatively, applications that require strict single-use shared secrets can still achieve this by opening a fresh TCP server socket with a fresh secret after each successful connection. Thus, the design of TCP Stealth does not exclude the use of replay-prevention schemes of previous designs, and using single-use shared secrets is strongly recommended if TCP Stealth without payload integrity protection is used.

8.3 On the Use of MD5

Some community members were surprised by our choice of MD5 for the hash function. The primary reason for this choice was that a single round of 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 somewhat weak hash function has no real downside, as the 32 bit values of the ISN hardly require a high-qualtiy hash function, as the security is clearly primarily limited by the use of only 32 bits.

8.4 Kernel-space vs. User-space

A common objection against Knock is the need to modify the kernel. However, as long as TCP is implemented inside the kernel, this is the simplest method to make it trivial for users to deploy Knock. Implementing a TCP stack in user-space makes it easier to detect the use of TCP Stealth by timing attacks, and may creates performance issues. Not to mention that insecure networking code is a security problem inside and outside the kernel alike. Having insecure networking code in an administrative service is likely to grant administrative priviledges. Thus, we advocate that the focus should be on auditing the code and making it easy to use and secure, and not on where it should run.

8.5 Inter-Operability with SYN Cookies

SYN cookies are a technique to protect against SYN flooding DoS attacks. By design they prevent the server from allocating any state information for an incoming connection request until the other side has proven to be alive. The server responds to the connection request with a challenge ISN which needs to be acknowledged correctly by the client.

Inter-Operability with TCP Stealth is given because our design only relies on the ISN sent by the client: At the time time of authentication, the server simply checks the AV in the received SYN segment. If after the first data segment from the client integrity protection should be applied, the ISN (and by this the original value of the integrity hash) can be reconstructed by subtracting one from the sequence number of the data segment received from the client.

8.6 TCP Stealth and SSH: Application Protocols Matter!

While the use of integrity protections with TCP Stealth is technically optional, port knocking without integrity protections offers little security against an adversary that observes network traffic and hijacks connections after the initial TCP handshake. Thus, future network protocols should be designed to exchange key material at the beginning of the first TCP segment.

Sadly, this is not the case for SSH, which instead exposes a banner with version information to an attacker way before the cryptographic handshake. Hence, design flaws in the SSH protocol currently require the use of an additional obfuscation patch [3] to effectively use TCP Stealth integrity protections with SSH.

8.7 On Modifying Applications

Some critics claimed that using Knock requires modifications to applications. The `libknockify` library demonstrates that this is false: by pre-loading `libknockify` existing legacy applications can be made to use Knock on GNU/Linux. However, for reasons of usability we do advocate that application developers should make the necessary and usually trivial modifications to explicitly support Knock.

8.8 Portability

We believe that the design of TCP Stealth is perfectly portable, and not limited to GNU/Linux. We hope that this thesis and the draft protocol for TCP Stealth submitted to the IETF [15] will help convince the community at large to develop and deploy modifications to modern operating systems to support TCP Stealth.

8.9 Security by Obscurity

Some opinions on port knocking in general and TCP Stealth in particular say that the design is based on security by obscurity. First, we want to make clear that security by obscurity refers to believing that a system can be made more secure just by hiding it from an attacker. While this is not inherently wrong (moving the SSH service away from port 22 certainly helps evading a large percentage of automated port scans), we explicitly state that the port knocking method employed by TCP Stealth is *not* based on obscurity: Only a client in possession of the TCP Stealth secret is able to calculate the cryptographically correct authentication vector. Thus, the security provided by TCP Stealth is based on a (shared) secret, not on obscurity.

A similar argument states that the problem should be solved using a virtual private network (VPN). Naturally, *if* users were to be able to deploy a secure VPN solution, that would also help. However, we argue for *defense in depth*. After all, an even better defense would be to just deploy secure application software. TCP Stealth is another tool, and security professionals should choose a combination of appropriate tools against the threat of colonization based on the needs, capabilities and budget of their organization.

8.10 Handling Server-Side NAT or Load Balancers

Currently, Knock does not work for services using server-side NAT (DNAT[5]) [25] or load balancers. However, in principle TCP Stealth support can be integrated into the rules for a DNAT or load balancer. Naturally, in this case it would not be the TCP server providing the shared secret to the kernel via a socket option, but the DNAT or load balancer configuration.

8.11 Attacks

Micahel Rash claimed[6] that it would be possible to write a detector for TCP Stealth "under the assumption that modern TCP stacks randomize ISN's". Under the assumptions that TCP Stealth is used in combination with the TCP timestamp option, and that sessions are not established at a frequency exceeding timestamp progression, and that the distinguishing attack is run against a kernel using a single round of MD5 to generate the pseudo-random ISN, we question that it is possible to write such a detector.

Alternatively, we claim that even if the TCP timestamp option were not available, but instead TCP Stealth's content integrity protection is used with an application protocol that begins with an integrity-protected random sequence, writing a passive distinguisher will also fail.

[5]`http://linux-ip.net/html/nat-dnat.html`
[6]`http://sourceforge.net/p/fwknop/mailman/message/31749987/`

9 Summary

We have presented TCP Stealth, a method for calculating TCP ISNs for port knocking and payload authentication, and Knock, an implementation of TCP Stealth for Linux. Using the techniques presented in this thesis, system administrators can harden their systems by protecting internal TCP services against the criminal activities pursued by ~~organized crime syndicates~~ national security services.

9.1 TCP Stealth is Stealthy

TCP traffic that is sent out by TCP Stealth differs in no way from normal patterns that emerge during communication. This is to prevent a passive attacker who is able to follow the entire communication from being able to tell an observed TCP Stealth enabled TCP session from an ordinary TCP session. As a result, TCP Stealth avoids easy detection by a passive attacker.

Naturally, an active attacker with observation capabilities may still be able to observe the handshake, attempt to connect to a protected server and to jump at conclusions about the use of TCP Stealth depending on whether the connection attempt was successful or not. Still, this requires significantly more effort than the situation today.

9.2 TCP Stealth Protects the Payload

An active attacker who knows about the use of TCP Stealth is not generally able to reuse parts of a recorded session of an authorized client to establish a connection to the TCP Stealth protected service.

Instead of making the shared secret single-use, replay protection in TCP Stealth is achieved by checking the integrity of the payload, which also provides protections against man-in-the-middle attacks that try to hijack connections after the TCP SYN.

9.3 TCP Stealth is Easy to Use

Avoiding single-use secrets also improves usability, as ordinary users cannot be expected to manage a set of rotating keys on multiple clients.

Another usability issue of previous designs is that they often require administrators to run additional services on the host (as in case of *knockknock* and *knockd*), require adjustments to firewall settings, or to tinker with configuration files. In contrast, TCP Stealth is designed to be trivial to configure and enable from both a user's and a developer's perspective in order to minimize possible errors from these sources.

9.4 TCP Stealth Belongs in the Kernel

Any kernel-level implementation of port-knocking includes the risk of incorporating devastating security flaws into the kernel. However, given the size of modern operating systems, an analysis focusing on this danger alone is too narrow.[7]

In particular, a correct kernel-level implementation also offers unique advantages compared to a user-space implementation, including ease-of-use for developers and system administrators, simplicity of the code, and performance, all of which indirectly help improve security.

Specifically, there are no inspection processes such as *libpcap* which would otherwise have to be permitted to snoop on traffic. Inside the kernel, all checks can be performed along the normal path of execution in the operating system's TCP stack. Furthermore, existing system calls can be extended to offer a simple and intuitive interface for applications, and system administrators do not need to worry about configuring and launching additional services.

9.5 TCP Stealth works with NAT

Nowadays, most end-user devices access the Internet from behind a gateway router which performs *network address translation* (NAT) (compare [29], Section 5.6.2). While TCP Stealth was designed to avoid the use of information that is commonly altered by NAT devices, some NAT devices modify TCP timestamps and ISNs and may thus interfere with the port knocking mechanism.

[7]Moxie disagrees: `http://www.thoughtcrime.org/software/knockknock/`

Appendix

A Minimal Examples of Server and Client Programs

```c

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/select.h>

#define TCP_STEALTH                     26
#define TCP_STEALTH_INTEGRITY_LEN       28

int main(int argc, char **argv)
{
        int sock, retval;
        int client;
        unsigned char buf[512];
        fd_set rfds;

        struct sockaddr_in addr;
        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        addr.sin_addr.s_addr=INADDR_ANY;

        char secret[64] = "This is my magic ID.";
        int payload_len = 4;

        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock < 0) {
                printf("socket() failed, %s\n", strerror(errno));
                return 1;
        }

        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH, secret, sizeof(secret))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }
        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH_INTEGRITY_LEN,
                        &payload_len, sizeof(payload_len))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }

        if (bind(sock, (struct sockaddr*) &addr, sizeof(addr))) {
                printf("bind() failed %s\n", strerror(errno));
                return 1;
        }

        if (listen(sock, 10)) {
                printf("listen() failed, %s\n", strerror(errno));
                return 1;
        }

        client = accept(sock, NULL, 0);
        if (client < 0) {
                printf("accept() failed, %s\n", strerror(errno));
                return 1;
        }

        do {
                FD_ZERO(&rfds);
                FD_SET(STDIN_FILENO, &rfds);
                FD_SET(client, &rfds);

                select(((sock > client) ? sock : client) + 1, &rfds,
                        NULL, NULL, NULL);

                if (FD_ISSET(STDIN_FILENO, &rfds)) {
                        int len = read(STDIN_FILENO, buf, sizeof(buf));
                        write(client, buf, len);
                }
                if (FD_ISSET(client, &rfds)) {
                        int len = read(client, buf, sizeof(buf));
                        if (!len) {
                                puts("Peer closed connection.");
                                break;
                        }
                        printf("%s", buf);
                }

        } while (retval > 0);

        return 0;
}
```

Listing 9: *Minimal Knock server*

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <sys/select.h>

#define TCP_STEALTH             26
#define TCP_STEALTH_INTEGRITY   27
#define SERVER_ADDR             "127.0.0.1"

int main(int argc, char **argv)
{
        int sock, retval;
        struct sockaddr_in addr;
        unsigned char buf[512];
        fd_set rfds;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(8080);
        inet_aton(SERVER_ADDR, &addr.sin_addr);

        char secret[64] = "This is my magic ID.";
        char payload[4] = "1234";

        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock < 0) {
                printf("socket() failed, %s\n", strerror(errno));
                return 1;
        }

        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH, secret, sizeof(secret))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }
        if (setsockopt(sock, IPPROTO_TCP, TCP_STEALTH_INTEGRITY, payload,
                        sizeof(payload))) {
                printf("setsockopt() failed, %s\n", strerror(errno));
                return 1;
        }

```

```c
	if (connect(sock, (struct sockaddr*) &addr, sizeof(addr))) {
		printf("connect() failed %s\n", strerror(errno));
		return 1;
	}


	do {
		FD_ZERO(&rfds);
		FD_SET(0, &rfds);
		FD_SET(sock, &rfds);

		select(sock + 1, &rfds, NULL, NULL, NULL);

		if (FD_ISSET(STDIN_FILENO, &rfds)) {
			int len = read(STDIN_FILENO, buf, sizeof(buf));
			write(sock, buf, len);
		}
		if (FD_ISSET(sock, &rfds)) {
			int len = read(sock, buf, sizeof(buf));
			if (!len) {
				puts("Peer closed connection.");
				break;
			}
			printf("%s", buf);
		}

	} while (retval > 0);

	return 0;
}
```

Listing 10: *Minimal Knock client*

B The Interface to the Dynamic Linking Loader

In order to enable TCP Stealth for existing applications, several functions of the *libc* and the network API need to be wrapped. This is achieved by using the functions provided by the interface to the dynamic linking loader (*ld* on traditional Linux systems). The process is in order to wrap a API function with user defined code is straightforward. Listings 11, 12 and 13 show a minimal example how overriding is done in code followed by a brief explanation.

```c
#include <stdio.h>

int main()
{
        puts("Hello world!");
        return 0;
}
```

Listing 11: *Source code of the* LD_PRELOAD *example program*

```
$ gcc program.c -o program
$ gcc -shared liboverride.c       \
      -ldl -fPIC                  \
      -o liboverride.so
$ ./program
Hello world!
$ LD_PRELOAD=./liboverride.so \
  ./program
Hello from shared object!
Hello world!
```

Listing 12: *Sequence of the needed shell commands used to compile and run the* LD_PRELOAD *example program and library*

```c
 1  #include <stdio.h>
 2  #define __USE_GNU
 3  #include <dlfcn.h>
 4  #include <stdlib.h>
 5
 6  static struct hooked_libc_functions {
 7          int (*puts)(const char *);
 8  } hooks = { 0 };
 9
10  int __attribute__ ((constructor)) init_hooks()
11  {
12          const char *err;
13          hooks.puts = dlsym(RTLD_NEXT, "puts");
14          err = dlerror();
15
16          if (err) {
17                  printf("Failed to resolve " \
18                          "symbol puts: %s\n", err);
19                  exit(-1);
20          }
21
22          return 0;
23  }
24
25  int puts(const char *s)
26  {
27          hooks.puts("Hello from shared object!");
28          return hooks.puts(s);
29  }
```

Listing 13: *Source code of the* LD_PRELOAD *example library*

1. The sample program in Listing 11 calls `puts` from the *libc* API which should be overridden with a custom implementation of `puts`.

2. An overriding library (Listing 13 would call `dlsym` with option `RTLD_NEXT` and the name of the function (`puts`) that should be wrapped (or any symbol residing in a dynamic library used by the program). It returns a pointer to the location with the provided symbol name within the original library on success which is stored as the function pointer `hooks.puts` in the example.

3. The overriding library provide a function with the identical signature as the function to be wrapped. (That is, name *and* arguments need to match the original function signature.)

4. In the following, the `puts` function in line 25 provided by the overriding library will be called each time the targeted program calls `puts`. The example implementation of the customized `puts` simply calls the original `puts` (`hooks.puts`) twice, once to output an extra message and once to output the original message passed to the `puts` function by the program.

Listing 12 shows how the program and the overriding library can be compiled. To document the functionality, the program first is called as generated by the compiler and afterwards with `liboverride.so` assigned to the `LD_PRELOAD` environment variable. The result is the additional message of the overriding library being output above the original message of the program.

The technique described above used in a similar manner to enable TCP Stealth in existing code.

References

[1] Belgacom Attack: Britain's GCHQ Hacked Belgian Telecoms Firm. http://www.spiegel.de/international/europe/british-spy-agency-gchq-hacked-belgian-telecoms-firm-a-923406.html, September 2013.

[2] Daniel Julius Bernstein. E-mail providing implementation details of syn cookies, October 1996. http://cr.yp.to/syncookies/archive, visited August 6th, 2014.

[3] brl. Obfuscated OpenSSH. https://github.com/brl/obfuscated-openssh, visited August 10th, 2014.

[4] André Fabian Silva Delgado. Package linux-libre-ltr-knock for Parabola Linux, July 2014. https://parabolagnulinux.org/packages/kernels/i686/linux-libre-lts-knock/, visited July 1st, 2014.

[5] Roger Dingledine and Nick Mathewson. Design of a Blocking-Resistant Anonymity System. Technical report, The Tor Project, Nov 2006.

[6] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*, August 2013.

[7] Wesley Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational), August 2007.

[8] Kjeld Egevang and Paul Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022.

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

[10] John Fletcher. An Arithmetic Checksum for Serial Transmissions. *Communications, IEEE Transactions on*, 30(1):247–252, January 1982.

[11] Barton Gellman and Ashkan Soltani. Nsa infiltrates links to yahoo, google data centers worldwide, snowden documents say. The Washington Post, October 2013.

[12] Andy Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385 (Proposed Standard), August 1998. Obsoleted by RFC 5925, updated by RFC 6691.

[13] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is It Still Possible to Extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[14] Jesper Juhl. Applying Patches To The Linux Kernel, August 2005. https://www.kernel.org/doc/Documentation/applying-patches.txt, visited July 1st, 2014.

[15] Julian Kirsch, Christian Grothoff, Jacob Appelbaum, and Holger Kenn. TCP Stealth – Draft, August 2014.

[16] Julian Kirsch, Christian Grothoff, Monika Ermert, Jacob Appelbaum, Laura Poitras, and Henrik Moltke. NSA/GCHQ: Das HACIENDA-Programm zur Kolonisierung des Internet. http://www.heise.de/ct/artikel/NSA-GCHQ-Das-HACIENDA-Programm-zur-Kolonisierung-des-Internet-2292574.html, August 2014.

[17] Martin Krzywinski. Port Knocking: Network Authentication Across Closed Ports. *SysAdmin Magazine*, 12:12–17, 2003.

[18] Robert Love. *Linux Kernel Development Second Edition*. http://www.makelinux.net/books/lkd2/ch10lev1sec3, visited August 7th, 2014.

[19] Moxie Marlinspike. *knockknock*, December 2009. http://www.thoughtcrime.org/software/knockknock/, visited May 5th, 2014.

[20] James Morris, David Miller, and Herbert Xu. Scatterlist Cryptographic API. http://lxr.free-electrons.com/source/Documentation/crypto/api-intro.txt, visited August 5th, 2014.

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

[22] The Institute of Electrical and Electronics Engineers. List of organizationally unique identifiers, July 2014. http://standards.ieee.org/develop/regauth/oui/oui.txt, visited July 28th, 2014.

[23] Laura Poitras, Marcel Rosenbach, and Holger Stark. A wie Angela. http://www.spiegel.de/spiegel/print/d-126267965.html, March 2014.

[24] Jon Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[25] Yakov Rekhter, Robert Moskowitz, Daniel Karrenberg, Geert Jan de Groot, and Eliot Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996. Updated by RFC 6761.

[26] Ronald Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992. Updated by RFC 6151.

[27] Craig H. Rowland. Covert Channels in the TCP/IP Protocol Suite. *FirstMonday*, 2(5), May 1997.

[28] 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, New York, NY, USA, 2011. ACM.

[29] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[30] Eugene Y. Vasserman, Nicholas Hopper, and James Tyra. SilentKnock: practical, provably undetectable authentication. In *International Journal of Information Security*, April 2009.