An Approach Towards Anomaly Based Detection and Profiling Covert TCP/IP Channels

Patrick A. Gilbert

A Thesis

In

Concordia Institute

For

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science (Information Systems Security) at

Concordia University

Montreal, Quebec, Canada

August 2009

Your file Votre référence
ISBN: 978-0-494-63156-0
Our file Notre référence
ISBN: 978-0-494-63156-0

Abstract

An Approach Towards Anomaly Based Detection and Profiling Covert TCP/IP Channels

Patrick A. Gilbert

Firewalls and detection systems have been used for preventing and detecting attacks by a wide variety of mechanisms. A problem has arisen where users and applications can circumvent security policies because of the particularities in the TCP/IP protocol, the ability to obfuscate the data payload, tunnel protocols, and covertly simulate a permitted communication. It has been shown that unusual traffic patterns may lead to discovery of covert channels that employ packet headers. In addition, covert channels can be detected by observing an anomaly in unused packet header fields. Presently, we are not aware of any schemes that address detecting anomalous traffic patterns that can potentially be created by a covert channel.

In this work, we will explore the approach of combining anomaly based detection and covert channel profiling to be used for detecting a very precise subset of covert storage channels in network protocols. We shall also discuss why this method is more practical and industry-ready compared to the present research on how to profile and mitigate these types of attacks. Finally, we shall describe a specialized tool to passively monitor networks for these types of attacks and show how it can be used to build an efficient hybrid covert channel and anomaly based detection system.

Acknowledgements

I'd like to thank my family for giving me the time to research and write this thesis and the encouragement and support that they gave me. I'd also like to express my gratitude to my supervisor, Dr. Prabir Bhattacharya for his flexibility, support and encouragement during my thesis work. His advice and guidance kept me on track and motivated.

Table of Contents

List of Figures viii

List of Table ix

Chapter 1 1

Overview 1

1.1 Introduction 1

1.2 The network security problem 2

1.3 Terminology 3

1.4 The prisoners' problem 4

1.5 A taxonomy for covert channels 5

1.6 TCP/IP and Internetworking theory 7

1.6.1 How Layering Works 8

1.6.2 Frames and Packets 9

1.6.3 The IP Header 10

1.6.4 Transport Layers 13

1.7 Thesis Organization 16

Chapter 2 17

2 Background 18

2.1 Introduction - Early defence mechanisms 18

2.2 Weaknesses in the TCP/IP protocol 20

2.3 Payload tunnelling 22

2.4 Countermeasures and network security 24

2.5 Research on detection 25

Chapter 3 28

3 Proposed method 28

3.1 Preprocessor analysis 31

3.2 Protocol profiling, sanitization and packet analysis 33

3.3 Profiling characteristics 33

3.4 Case: Tunneling 35

Chapter 4 37

4 Implementation 37

4.1 Packet backlog 37

4.2 Network capture 37

4.3 Embedded security features 38

4.4 Summary of special features 38

4.5 Understanding rulesets 39

4.5.1 Comments 40

4.5.2 Interface 40

4.5.3 Include files 40

4.5.4 Variables 40

4.5.5 General format 41

4.5.6 Ruletype 43

4.5.7 Output 43

4.5.8 Mandatory ruletypes 45

4.5.9 Rules and alerts classes 49

4.5.10 Filters 49
4.6 Preprocessors 68
4.6.1 arp_validate preprocessor 68
4.6.2 defrag preprocessor 70
4.6.3 stream preprocessor 71
4.7 Streams profiling 72
4.7.1 http_decode preprocessor 73
4.7.2 telnet_decode preprocessor 73
4.7.3 rpc_decode preprocessor 74
4.8 Log output 74
4.8.1 Log entries 74
4.9 Stream preprocessor alerts 74
4.10 IP defragmentation preprocessor alerts 76
4.11 ARP preprocessor alerts 76
4.12 Ipacket tool 77
Chapter 5 78
5 Performance evaluation 78
6 Conclusions and Future work 79
References 81

List of Figures

Figure 1 Alice is sending a message (E) to Bob while trying to evade suspicion from Eve. She has many methods at her disposal: a shared secret key (K), a private random source (R), plain text (C) (an overt message) or encrypted cipher text (S) (a covert escape plan). Eve can monitor the passing traffic for covert channels, and can even alter the traffic to disrupt. Bob must be able to recover E from his knowledge of the cipher text S and from the key K. 5

Figure 2 A basic setup using the HTTP protocol to circumvent firewall security policies. 7

Figure 3: Figure of TCP/IP networking layers from a sending host to a receiving host with intermediating nodes. 9

Figure 4: Figure demonstrating the nesting of the different headers in relation to the different layers, when a user transmits a packet. 10

Figure 5: An IP packet has a header that includes the source and destination IP addresses, version, type, and service information, options, and a data section. 11

Figure 6: TCP connection establishment handshake 15

Figure 7: Flow diagram describing four steps to our proposed method: Network capture and reassembly, preliminary analysis via preprocessors, packet analysis via rules and output responses. 31

List of Table

Chapter 1

Overview

1.1 Introduction

TCP/IP was not designed with security in mind and thus has very few security components by default. The protocols lack many features that are desirable or needed on an insecure network, such as authentication or encryption. The designers did not spend a great deal of effort securing this protocol, because after all, the goal wasn't to support communications that required a high degree of confidentiality or integrity. The goal of the project was to develop a protocol for the open exchange of information in a small community and users generally trusted each other.

As the usage of the Internet and the TCP/IP protocols increased, it became apparent that security would become more and more problematic. The TCP/IP protocol was being used for a wide variety of applications and even implemented into devices as a migration path from serial connections and proprietary interfaces. Today, a plethora of devices are now internet protocol (IP) enabled and we must compose with these devices and applications.

Given the increasing complexity, it has become a daunting task to define a network security policy that ensures that you are permitting the proper protocols between a host and destination. Permitting a proprietary protocol could open a security breach because a different TCP/IP communication could be tunnelled through. As an example, permitting the SSH protocol to communicate with any host on the internet opens up the possibility of tunnelling anything and everything through this channel, whereas the network's stated

security policy intended only that connections must be encrypted if they are to leave the corporate network. The goal of the policy is met, but has the adverse effect of allowing convert TCP/IP channels.

In this work, we will explore the approach of anomaly based detection and profiling to be used for detecting a very precise subset of covert storage channels in certain network protocols. It is well known that in computer networks, overt channels such as network protocols are used as carriers for covert channels [1] [2].

This reduced scope fulfills the precise requirements as stated by my sponsor and serves as a basis for future work for detecting and profiling more elaborate covert channels. An elegant and more elaborate practical example is stated in [3]. As pointed out by Zander [4] and Van Horenbeeck [5], there exists a gap between academic research on this topic and real world applications; the applicability of the some approaches on detection in the current academic research is definitely questionable.

Our approach differs from the current academic research on this topic because it is industry-ready and being used with appreciable results in a very large production environment with a high-speed gigabit network. The approach has taken into account certain constraints in order to not possibly disrupt or degrade performance, a limitation that doesn't exist in the current academic theory and proof of concepts.

Finally, while our approach has been successful, it requires constant refinement to detect improvements made to covert channels rendering them more difficult to detect. Is it expected that this trend will continue.

1.2 The network security problem

Network security attempts to mitigate weaknesses introduced by network-connected

devices. Since the 1990's companies have incorporated security in their network infrastructure by means of firewalls, detection systems and its derivatives.
These types of systems have emerged because it is increasingly difficult to determine if an information system will be free of vulnerabilities. Ladnwehr [6] shows that information systems suffer from security vulnerabilities regardless of their purpose, manufacturer or origin and that is it technically difficult as well as economically costly to ensure that they are not susceptible to attacks.
In addition, certain environments must also compose with legacy devices that communicate using non-standard or basic client-server protocols.

The common security measure was to permit the communication, without any regards as to if the protocol semantics were followed or even the correct methods were used. When once a proxy-application based firewalled was suitable, other factors such as performance and compatibility came into play. With the advent of packet filtering we also inherited its drawbacks, one of which was not inspecting packets for protocol correctness. Under a packet filter, a rule permitting "telnet" for example could easily be a covert channel bridging the entire network to the internet, where as a proxy-application based firewall could not be deceived so easily.

1.3 Terminology

The term covert channel is used to refer to the hiding of information in network protocols. The information transmitted across the covert channel is referred to as covert information. When a transmission of information is conducted over a public channel, we can refer to this mechanism as an overt channel. In general, we can state that a covert

channel is considered a 'transmission channel that may be used to transfer data in a manner that violates security policy'.

As cited in Ladnwehr [6], covert channels were classified into storage and timing channels.

- Covert storage channels involve the direct or indirect writing to a storage location by the sender and the direct or indirect reading of the storage location by the receiver. Covert storage channels typically involve a finite resource that is shared by two subjects at different security levels. This will be the main type of channel that we will address.
- Covert timing channels involve the sender signaling information to the receiver by modulating its own use of system resources (e.g., central processing unit time) in such a way that this manipulation affects the real response time observed by the receiver.

1.4 The prisoners' problem

The prisoners' problem was first posed by Simmons in 1983 and serves as a model for covert channel communication [7]. Two people, Alice and Bob, are thrown into prison and intend to escape. To agree on an escape plan they need to communicate, but the warden Eve monitors all their messages. If Eve detects any signs of suspicious messages, she will place Alice and Bob in solitary confinement making their plan to escape impossible and severing all future communications between them. Alice and Bob must exchange innocuous messages containing hidden information that should hopefully not arouse any suspicion from Eve. According to Craver [8], there are three types of wardens:

• A *passive* warden can only spy on the channel, but cannot alter any messages.

• An *active* warden is able to slightly modify the messages, but without altering the semantic context.

• A *malicious* warden may alter the messages at will.

If we were to transpose this scenario to computer networks, Alice and Bob use two computers for communication, the network as a public channel, and transmit two types of messages: innocuous plain text (C) messages and covert messages containing escape plans (S). Alice and Bob have a share secret key (K), which is useful for determining covert channel encoding parameters and encrypting and authenticating covert messages. Eve can monitor the passing network traffic for covert channels, slightly modify the traffic or even alter the messages being transmitted between Alice and Bob in an attempt to unmask their escape plans. The figure 1.1 below describes the model (Alice sending to Bob, with Eve as a warden).

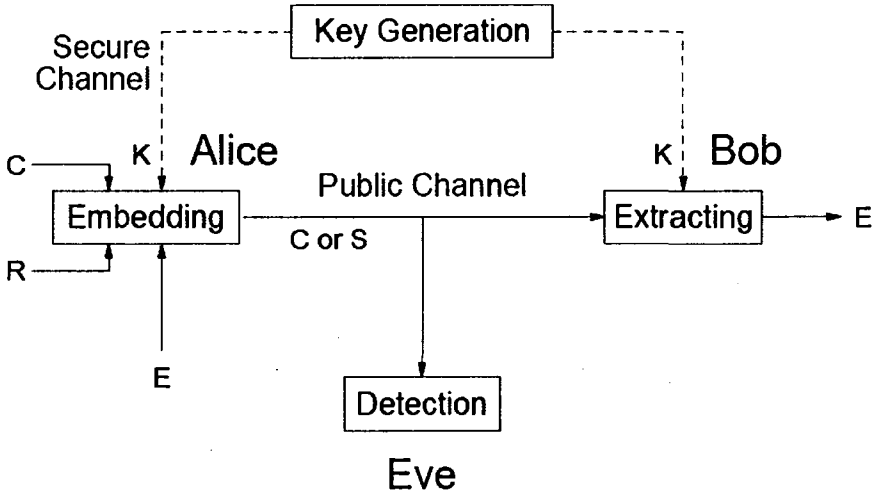


Figure 1 Alice is sending a message (E) to Bob while trying to evade suspicion from Eve. She has many methods at her disposal: a shared secret key (K), a private random source (R), plain text (C) (an overt message) or encrypted cipher text (S) (a covert escape plan). Eve can monitor the passing traffic for covert channels, and can even alter the traffic to disrupt. Bob must be able to recover E from his knowledge of the cipher text S and from the key K.

1.5 A taxonomy for covert channels

The following taxonomy according to Van Horenbeeck [5] is proposed that further differentiates covert channels by objective:

- Data smuggling: Covert channels can be used to smuggle data into or out of a corporate network, in which this data is classified as determined by the security policy. The type of data loss prevention is not the focus of this paper.
- Automated code execution: Covert channels can allow for the transfer and execution of code on internal hosts that should not normally be executed according to the security policy.
- Protocol tunnelling: allows internal entities to connect to host and services on the internet that are not permitted by the security policy. This technique permits the retrieval through permitted protocol and ports, content that would normally pass by many layers of protection (anti-virus, content filtering, etc) therefore exposing the internal network to a plethora of threats. A more serious threat is the establishment of a covert channel tunnel that allow the creation of arbitrary data transfer channels in the data streams authorized by a firewall thus permitting a "backdoor" for external entities to enter the internal network at will [9].

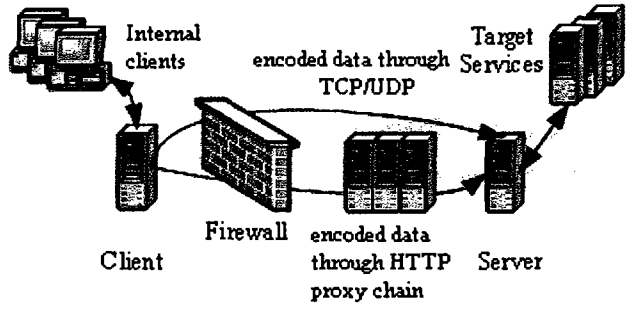


Figure 2 A basic setup using the HTTP protocol to circumvent firewall security policies.

1.6 TCP/IP and Internetworking theory

In order to fully understand the problem, a full understanding of TCP/IP and internetworking are necessary. The infrastructure of the Internet consists of a number of routers and interconnected network devices whose function is to forward information from its source to its destination. In packet switching networks, routing directs forwarding, the transit of logically addressed packets from their source toward their ultimate destination through intermediate nodes; typically hardware devices called routers, bridges, gateways, firewalls, or switches. Network devices and other devices that communicate over the Internet are known as hosts. The Internet is a public packet-switched network made up of hosts that can send packets onto the network destined for any other host. Transport Control Protocol/Internet Protocol (TCP/IP) is termed the 'language of the Internet'. Any device or software that uses TCP/IP can communicate on the Internet. Its design is based on five protocol layers, each of which is responsible to provide services to layers above it.

- Physical layer provides the network device with direct access to the transmission medium and is also responsible for encoding and decoding signals over this medium , such as a wire or an optical fiber;
- Data link layer logically passes information between two points over the physical media, basically moving data across a LAN
- Network layer is responsible for global addressing and routing packets between physical network segments;

- Transport layer is responsible for communication between processes, and optionally reliable connections
- Application is responsible for encoding and decoding information in formats understood by applications, as well as providing any necessary services not provided by lower layers.

TCP/IP is an open standard that is free, robust and flexible. TCP/IP doesn't specify how the Physical and Data Link Layers work; it just expects them to provide enough functionality to link two or more network devices together into a Local Area Network. That is, because TCP/IP is an Internetwork Protocol suite, it specifies how data can make its way from a host on to another host. The Internet Protocol or IP is the standard language for the packets that flow across the Internet. Services such as the Web, SMTP, and DNS are all built on top of this protocol (IP).

1.6.1 How Layering Works

Each layer in the TCP/IP suite operates by using the data portion of the layer above it; information being sent is passed down from the application layer to the physical layer. Another way of looking at the protocol stack is to start from the highest level and work your way down. As the data goes down through the TCP/IP networking layers, information specific to that layer is added to the data until it reaches the bottom, at which point it is sent out over the communications link (see figure 3). When it is received on the other side, the process is reversed, with each layer removing the data specific to that layer, until it is presented to the ultimate recipient of the data.

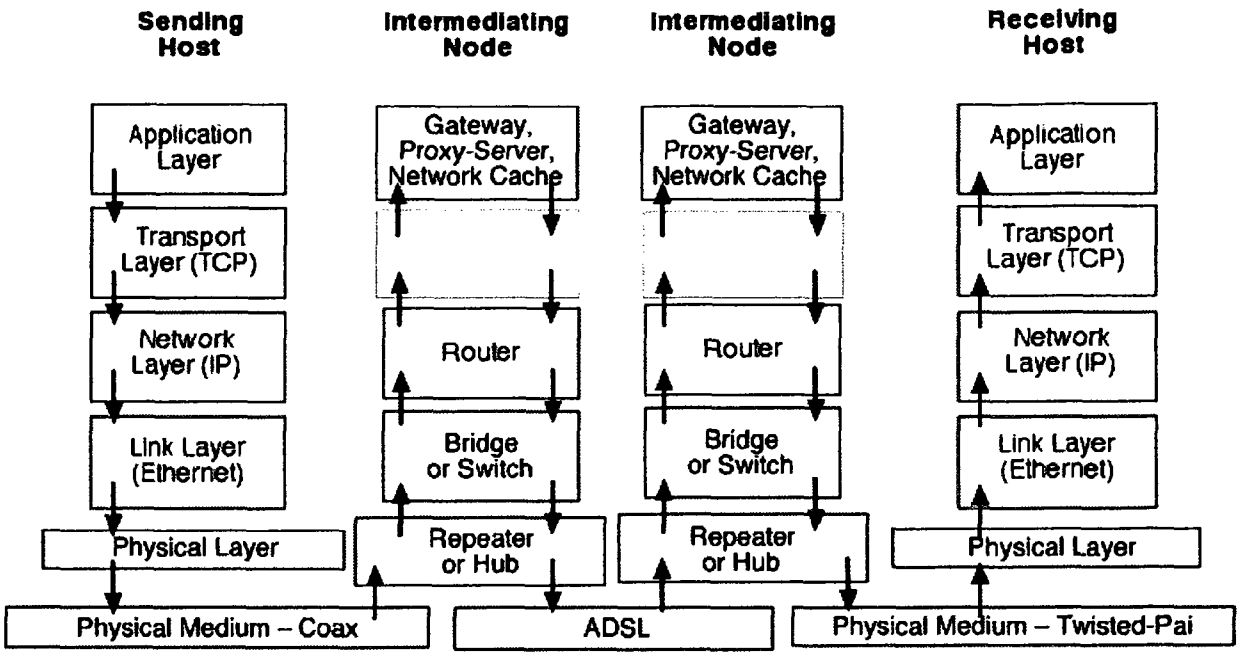


Figure 3: Figure of TCP/IP networking layers from a sending host to a receiving host with intermediating nodes.

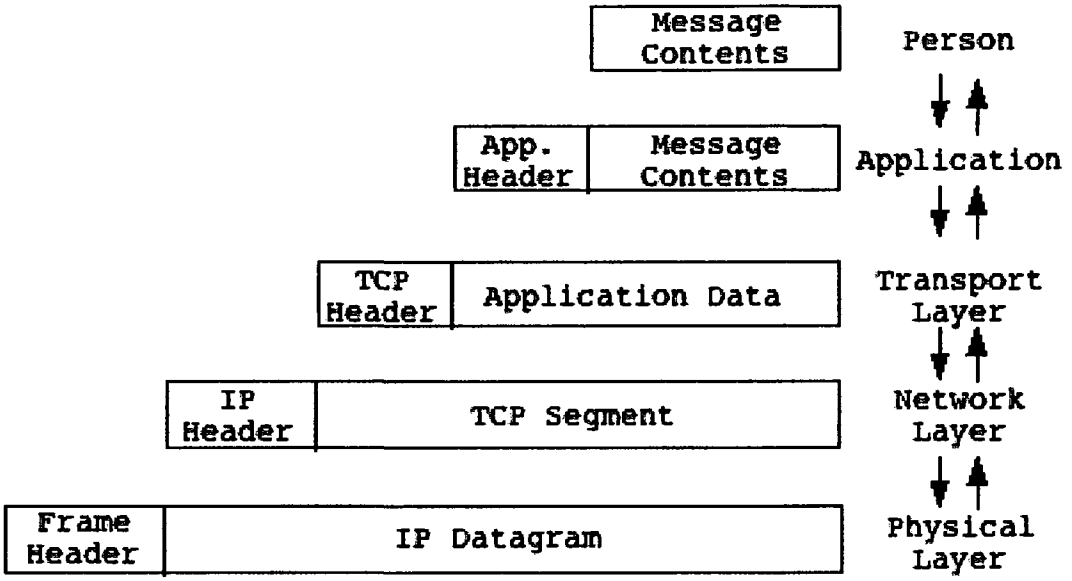
1.6.2 Frames and Packets

The basic unit of Logical Link Layer data transmission is the frame. The Internet Protocol has a similar basic unit of data transmission: the packet. The standard network protocol on the Internet is known as the Internet Protocol, version 4 (IPv4) [10]. IP provides an unreliable datagram service: it breaks the information that it transmits into packets (also known as datagrams) and routes each to its destination independently, but provides no guarantees that packets are properly delivered.

An IP Packet is quite similar in structure to an Ethernet frame, with source and destination addresses, packet description and option fields, checksums, and a data portion. Because of the way IP is layered on top of the Logical Link Layer, all of the

packet structure is nested inside the data portion of the logical link frame (Ethernet, for example).

There are many different types of packets exchanged in a TCP/IP network, starting with the ARP and RARP packets and including IP packets. The generic structure of an IP packet is illustrated below.



1.7

Figure 4: Figure demonstrating the nesting of the different headers in relation to the different layers, when a user transmits a packet.

1.7.1 The IP Header

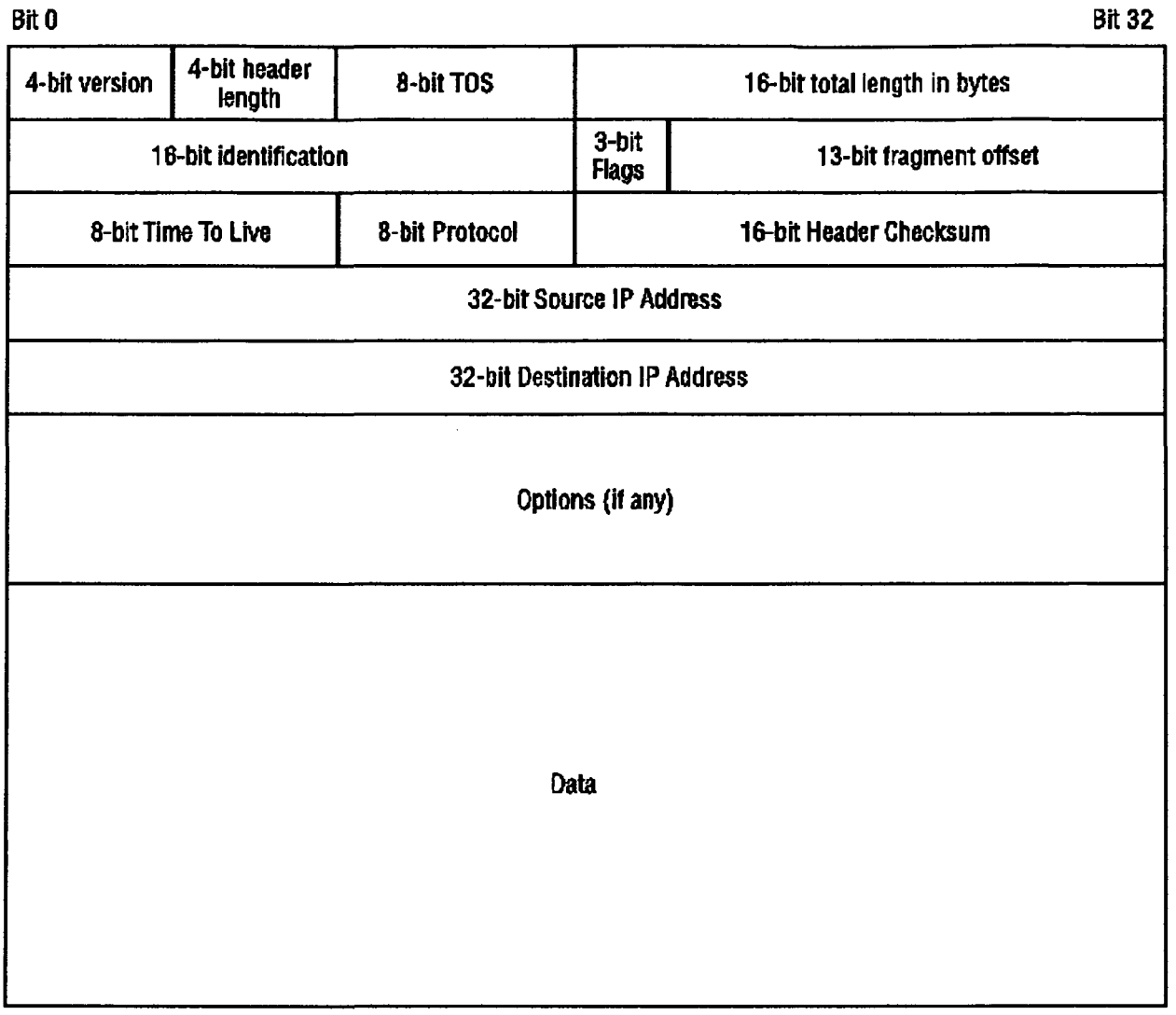


Figure 5: An IP packet has a header that includes the source and destination IP addresses, version, type, and service information, options, and a data section.

As shown in the above figure, the IP header is typically 20 bytes long, but can be up to 60 bytes long if the packet includes IP options. The non-optional fields are as follows:

Version These four bits identify which version of IP generated the packet. The current IP version is 4.

Header length This four-bit value is the number of 32-bit words in the header, and by default is 20.

Precedence (TOS) These eight bits are an early attempt at implementing quality of service for IP. They are comprised of three bits for packet precedence (ignored by modern implementations of IP), four Type of Service bits, and a bit to be left at zero. Only one of the four Type of Service bits can be turned on. The four bits are: minimize delay, maximize throughput, maximize reliability, and minimize cost. All zeros mean the network should use normal priority in processing the packet. RFCs 1340 and 1349 specify TOS use. Most implementations of IP don't allow the application to set the TOS value for the communicated data, which limits the usefulness of this field.

Datagram Length This is the total length of the IP datagram in bytes. Since this is a 16-bit field, the maximum IP packet size is 165535 bytes in length, even if the Data Link Layer frame could accommodate a larger packet.

Identification To guarantee that packets are unique and to assist in the reassembly of data streams that have been broken down into packets, the sending computer numbers the packets, incrementing the value when each packet is sent. This value doesn't have to start at zero, and isn't necessarily incremented by one on all implementations of IP.

Flags Flags define whether or not this packet is a fragment of a larger packet.

Fragment Offset This defines where in the chain of fragments this fragment belongs when reassembling a fragmented packet.

Time To Live This declares the number of routers through which the packet may pass before it is dropped. This prevents packets from getting caught in routing loops. This field is typically set to 32 or 64. When a packet is dropped due to TTL expiring, an ICMP message is sent to the sender.

Protocol This field shows which IP protocol (TCP, UDP, ICMP, etc.) generated the packet and should be the recipient of it at the destination.

Header Checksum All of the 16-bit values in the packet header are summed together using one's complement addition and placed in this field.

Source This is the device sending the IP packet.

Destination This is the intended recipient of the IP packet.

Network Byte Order (Big Endian)

Internet Protocol packets are sent in Network Byte Order, where the most significant bit of a word is sent first. For example, the 32-bit word 0xDEADBEEF would have the byte 0xDE sent first, and the byte 0xEF sent last. For this reason, Network byte order is also called Big-Endian. In contrast, in Little-Endian byte ordering, the least significant bits are sent first.

1.7.2 Transport Layers

A number of transport layers exist, each providing different services. ICMP ("Internet Control Message Protocol") is used for delivering many types of error message from the

network and transport layers as well as performing a variety of administrative functions [11]. UDP ("User Datagram Protocol") provides an unreliable datagram service between applications [12]; the only important feature that it adds to IP is application addressing. TCP ("Transmission Control Protocol"), the most common transport protocol on the Internet, provides reliable bidirectional streams between applications. Other transport protocols exist, providing other services. TCP breaks streams down into segments which are then encapsulated into IP Packets. Like IP packets, TCP segments contain both headers and payload data. TCP assigns sequence numbers to every byte of payload data sent and requires that every byte be acknowledged; if any segments are lost or corrupted, either they will be resent or an error will be detected. Duplicate and out-of-order packets can also be detected and corrected using the sequence numbers. TCP headers contain application-layer addresses, data and acknowledgement sequence numbers, and control flags, among other fields. Segments used to initiate connections have the SYN ("synchronize") flag set; those used to finalize connections have the FIN flag set. The ACK flag indicates that a segment is acknowledging data received from the other end of the connection. Opening a TCP connection requires that both endpoints exchange initial sequence numbers (ISNs) using an algorithm known as the three-way handshake, shown in Figure 1.6.

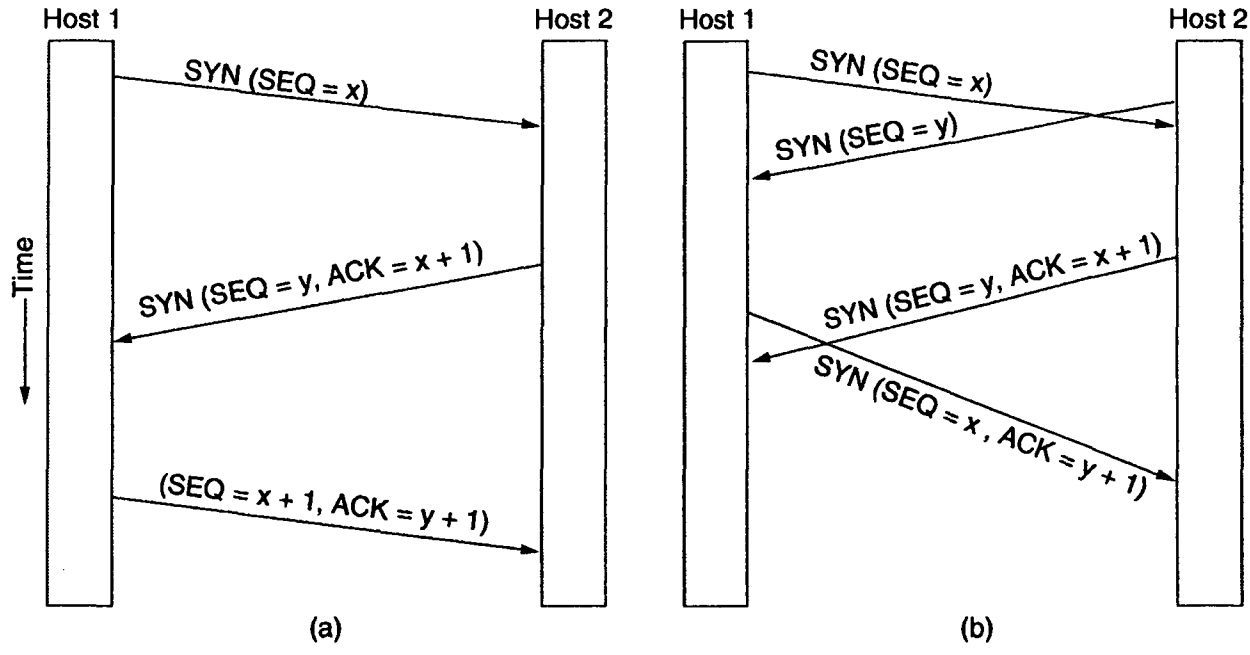


Figure 6: TCP connection establishment handshake

The client starts the handshake by choosing an ISN and sending it to the server in a segment with the SYN flag set. If the server chooses to accept the connection, it responds with its own ISN in a segment carrying the SYN flag; it also signals that it received the client's ISN by setting its acknowledgement number to the client's ISN plus one and setting the ACK flag. The client then acknowledges receipt of the server's ISN with an ACK segment carrying the server's ISN plus one in its acknowledgement number field. After completing this exchange, a TCP connection is established and both the client and server can send and receive data.

Both TCP and UDP use ports for addressing applications. A port is simply a 16-bit integer where any application can bind itself to listen and send packets. Some operating systems restrict the port range that can be used by "user land" applications. It is

common for ports < 1024 to be used exclusively for services offered by the operating system. For instance, SSH ("Secure Shell", used for secure remote logins, file transfer, and tunneling other protocols) servers typically listen on port 22/TCP, HTTP (used for retrieving documents from web servers) uses port 80/TCP, and DNS ("Domain Name System", responsible for mapping easy-to-remember names to numeric IP addresses) servers listen on port 53/UDP. On most operating systems, applications use the socket interface to send and receive messages. Unprivileged applications are typically limited to using TCP and UDP; headers, with the exception of addressing information, are automatically generated by the operating system, as are TCP connection establishment and finalization messages. Privileged applications can use the raw socket interface to generate arbitrary packets [13].

Neither IPv4, TCP, nor UDP provide any authentication or privacy services; any host can generate any packet and any host observing packets at the network layer can read the packets' payloads. If these services are important, then they can be provided by application-layer protocols, TCP extensions such as TLS [14], IP extensions such as IPSec [15], or the next-generation Internet Protocol, IPv6 [16].

1.8 Thesis Organization

This thesis is organized in the following chapters. Some basic concepts, terminology, and TCP/IP and internetworking concepts have been introduced in Chapter 1. Chapter 2 is a background study of early defense mechanisms, weaknesses in the TCP/IP protocol and current countermeasures and some current methods on detection. In Chapter 3, we present our approach of anomaly based detection and profiling to be used for detecting a very precise subset of covert storage channels in certain network protocols. Chapter 4 and

5 discuss implementation and a performance evaluation. Finally the conclusion and discussion are presented in Chapter 6.

Chapter 2

2 Background

2.1 Introduction - Early defence mechanisms

The first line of defence consisted of deploying firewalls and restricting the network traffic allowed to enter or leave local networks by defining interior and exterior interfaces, while still allowing approved traffic to pass. This was to be done at the network level in routers and firewalls that would provide the intended connectivity to the Internet. Firewall technologies were not abundant, and very few firewall policies would implement egress filtering. Perimeters were established, delimitating zones to leave the intruders "out", and DMZ zones offered services that the "public" internet user can utilize, namely http, mail and ftp services. Firewalls knew that if a packet proclaims to come from the "internal" network and is received on an external interface, something was definitely wrong. The best practice would be to allow traffic that is explicitly permitted while blocking everything else, taking as parameters source, destination, port and protocol. In actuality, there is much more detailed information being exchanged.
One type of firewall technology, network layer firewalls or also called packet filters [17]; work on the low-level of the TCP/IP protocol stack and attempt to filter inbound and outbound traffic by restricting the ports to which users may connect. As an example, disallowing outbound connections to anything except TCP ports 80 (HTTP), 443 (HTTPS), and 20 and 21 (FTP) would be a standard policy in most corporate environments. This may provide a false sense of security, because services may be setup on non standard ports on a remote host (such as the secure shell service on port 443

instead of 22 for example), and the firewall would allow for this type of connection to persist.

A form of packet filtering named "Deep Packet Inspection" examines the data and header part of the packet searching for protocol non-compliance, viruses, malicious code, SPAM and other threats or any other predefined criteria to decide if the packet can pass or not. This is in contrast to stateful packet inspection which only validates the header portion of a packet [18], after which an initial check is passed; a flow is stored in a table so that quick routing of subsequent packets can occur. These packets are not inspected for malicious content.

With the ability to inspect Layers 2 through 7 of the OSI model, Deep Packet Inspection [19] seems like a promising new method to identify and classify traffic based on signatures-based comparisons, packet flows, heuristic, statistical, or anomaly-based techniques, or some combination of these. The idea was to move the inspection of the data in packets to the same application that does the packet filtering rather than have this task offloaded on another system.

Application-layer firewalls work on the application layer of the TCP/IP stack and by the use of application proxies can easily filter traffic that doesn't match the expected protocol or port; they effectively are implementing a correctness check upon the application protocols they gateway [20]. When the firewall is inspecting all packets for improper content, the task becomes rapidly complex given the variety of applications and the diversity of content that can be allowed in each sequence of packets. If we are also to take into account performance degradation and the various security flaws in the TCP/IP protocol itself, we have as a result, a difficult implementation in any environment.

Protocol scrubber or "Traffic normalisation": This approach consists of an active interposition mechanism that attempts to homogenize network flows by identifying and removing attacks in real-time. The difference in this approach is to continuously remove malicious content in the traffic flow by normalising protocol headers, padding and extensions as described by Malan et al. [21], Handley et al. [22] and Fisk et al. [23] for the lifetime of the flow. This approach does have its merit, but has the same weakness as described for application-firewalls, namely network traffic disruption and degradation. The main weak point of this approach would be the handling of "non-well-behaved" clients which would have their flows tampered with and also suffer the corresponding packet loss given that the scrubber throws away any packets that could lead to inconsistencies [21]. It has been our experience that any tool that is actively interposing itself in a production environment, i.e. modifying traffic flows and subsequently breaking applications face a certain security vs. usability debate. We prefer to use a passive approach in an active production environment with IDS evasion detection techniques and anomalous network flows in mind.

2.2 Weaknesses in the TCP/IP protocol

Security was not an integral part of the TCP/IP design process and it has been determined that many flaw and design weaknesses exist. A model of TCP/IP networks in regard to some well-known security threats is presented in [24]. This model characterizes the topology of TCP/IP security to enable a better understanding of the related vulnerabilities. A classic paper, [25] points out serious security flaws in the TCP/IP protocol suite with details on a variety of attacks.

For instance, the protocols governing TCP/IP are not designed to ensure integrity of the messages being transferred. An important weakness of IP packets is that they contain no strong authentication information; therefore hosts usually cannot distinguish between packets sent by an authorized user and packets sent by an unauthorized intruder.

For example, the IP option of source routing (loose source routing) in the IP header is controlled by the sender. This is problematic, because the sender of a packet can specify the route that a packet should take through the network. An attacker needs only to send a single valid packet for that new route to be used for the entire TCP connection. This means that an otherwise unreachable device can now be reached by a specified route by the attacker; however we must note that this type of attack is less effective when performed on the Internet.

Another weakness in the protocol is masquerading as another host via IP spoofing. Sequence number prediction is a fatal flaw that permits an attacker to masquerade as another host and therefore abuse trust relationships. A Secure Networks Inc paper details the flaw in detail [26].

Protocols used to distribute routing information and map physical addresses to IP addresses on local networks are not secure, allowing attackers to intercept and manipulate traffic destined to other hosts [25],[27]. Furthermore, flaws in the ICMP Router Discovery Protocol have no form of authentication, therefore by spoofing IRDP Router Advertisements, and attacker can remotely add default route entries to a remote system. Hosts trusting ICMP messages are vulnerable to the same kinds of attack enabled by source routing [28].

IP and TCP packets can be fragmented at (almost) arbitrary boundaries, allowing

attackers to split attack signatures between multiple packets. Furthermore, IP fragments and TCP segments are allowed to overlap; it is up to the receiver to decide how to re-assemble them. As a result, firewalls and detection systems that re-assemble streams before checking them may miss attacks against hosts that use different re-assembly strategies [29].

Another type of attack is by altering the TTL of IP packets; it is possible to make the detection system see packets that will not arrive at the target of the attack. By inserting fake data into the stream, an attacker can interleave the attack with bogus information, thus hiding the attack from the detection system while the target correctly reconstructs this attack data and reacts to it.

2.3 Payload tunnelling

A tunnel acts as a relay point between two connections without changing the messages; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages. Various protocols exist that allow IP packets to be encapsulated inside protocols that run on top of IP. This encapsulation of protocols is known as tunnelling. These types of channels are especially useful to circumvent firewalls that limit outbound traffic to only a few select application protocols, a security problem that is quite present in corporate networks today. For example, an attacker can use a non-HTTP protocol to transfer and steal information because the firewalls cannot provide additional assurance that another protocol is not being used on an HTTP port.

One of the first tunnelling techniques to appear was Loki, an approach that tunnelled data in the payload of ICMP echo messages [30]. Similar approaches were implemented by Stodle [31] and Zelenchuk [32], the later using an indirect approach with a bounce host. Later approaches to tunnelling included HTTP tunnelling where a tool was developed by Padgett [33] to tunnel SSH over HTTP proxies. Subsequent refinements where also developed by Dyatlov and LeBoutillier where their implementation provided tools to tunnel UDP or TCP over HTTP [34], [35]. As an interesting side note, Lundstrom implemented a tool named "MailTunnel" that can establish a bidirectional tunnel over the exchange of email [34]. Indeed, interesting methods of tunnelling have been engineered over the years, namely covert channels in the IPID and sequence number fields of TCP/IP headers [37], tunnelling TCP connections to a remote host using ICMP echo request and reply packets [38], tunnelling using only acknowledgement packets [39], [40] or portions of TCP and IP Headers [41] and finally IP tunnels using DNS queries and replies for IP packet encapsulation [42]. A tool even exists to allowing the creation of arbitrary data transfer channels in the data streams authorized by a network access control system [43], [44]. Network covert shells have been used by attackers to communicate with compromised hosts, particularly in distributed denial of service attacks [45]. Many tools exist for setting up these shells using a variety of protocols including TCP, IP, HTTP, ICMP, AODV, and MAC [46]-[54]. Common practice is to encapsulate information in a legitimate protocol to bypass firewalls and content filters.

There are legitimate examples of tunnelling that are well documented and would be in accordance to the network's stated security policy. For instance, IPSec (a security architecture for IP) [15] can create encrypted tunnels for IP and other network protocols

on top of IP, whereas GRE ("Generic Routing Encapsulation") [55] can create plain-text tunnels. What our tool will attempt to identify are specific unauthorized tunnels that do not comply with the network's stated security policy.

2.4 Countermeasures and network security

In the production environments that we have examined, possible countermeasures include blocking unnecessary ports and protocols at network boundaries or at least limiting the destinations of particular permitted protocol and port.

An example would be to use egress filtering on the ICMP protocol, a common measure that is used to mostly prevent DoS type attacks originating from servers and clients inside the network. It also happens to prevent a particular type of covert channel as described by daemon9 with his tool Loki [30], PingTunnel [31] and Skeeve [32].

A Split-DNS architecture is used to limit outgoing DNS requests from only a few name servers internally and treating external requests with bastioned DNS servers with limited public DNS records. This setup limits techniques that could be used such as DNS tunnel [56].

Techniques such as bouncing covert channels as specified by Rowland [57] are easily thwarted by good router hygiene and appropriate egress/ingress filtering.

As described by Padlipsky [58], securing networks against wiretapping and securing routers against compromise prevents some covert channel scenarios in which covert senders or receivers act as middlemen [58].

Very few countermeasures exist for preventing covert channels on permitted protocols and ports, HTTP for example. This has led to many tools being available [33]-[35] to

circumvent security policies [59] and has also fueled different approaches on how to detect this problem.

2.5 Research on detection

Historically, three main detection approaches exist. The signature-based approach involves building and updating a signatures database and notifying the administrator when a known signature is found in the network data streams. The protocol-based approach focuses on protocol anomalies or violations and incases the monitored communications turn out to be "abnormal", the operator is notified. The behavior-based approach involves creating behavioral user profiles (users, workstations, servers, network streams, etc.) and using statistical methods to determine if the observed data stream is suspicious [60]. Detecting covert channels are desirable to discourage the use of these channels and secondly, it is widely recognized that covert channels are impossible to eliminate entirely, even in highly optimized network protocols [61]. It is therefore crucial to monitor their activity.

While there are known techniques for detecting covert channels in layer 3 and 4 protocols [62], [63], and [64], very few exist for layer 7 channels.

In one approach, Schear *et al.* [65] proposed eliminating covert channels in HTTP responses by enforcing RFC protocol-compliant behavior and restricting usable response headers to a fixed set in a particular order, and by verifying response header fields against the corresponding object metadata and client request.

Sohn *et al.* [66]-[68] used a Support Vector Machine based approach to evaluate the accuracy of detecting covert channels embedded in ICMP echo packets and identification field of IP header and the sequence number field of TCP header.

Techniques on data hiding in IP fields can be found here [69]. While our interest is detecting covert channels in the payload channel, Sohn *et al.* did achieved classification accuracies of up to 99 percent when training a classifier on normal and abnormal packets generated by Loki [30]. Murdoch [70] argues that

TCP and IP specifications exhibit sufficient structure to define what is normal, and that learning methods such as SVM are overkill and they present a suite of tests to detect whether the IP ID and TCP ISN generating processes are in compliance with the specifications.

Pack *et al.* proposed detecting HTTP tunnels by using behavior profiles of traffic flows [71]. This type of approach is independent from payload inspection and is based on social behavior of hosts by looking at their connection patterns [72]-[74]. Profiles are based on a number of metrics such as the average packet size, ratio of small and large packets, change of packet size patterns, total number of packets sent/received, and connection duration. If the behavior of a flow under observation deviates from the normal HTTP behavior profile it is likely to be an HTTP tunnel.

Borders *et al.* developed a tool for detecting covert channels over outbound HTTP tunnels based on a similar approach [75]. It analyses HTTP traffic over a training period, and is then able to detect abnormal HTTP flows using metrics such as request size, request regularity, time between requests, time of the day, and outbound bandwidth usage. Other similar approaches have also been developed by Crotti [76] and Erman [77].

Hjelmvik *et al.* [78] developed a statistical protocol identification algorithm utilizing various statistical flow and application layer data features in order to identify application layer protocols by comparison of probability vectors to protocol models of

known protocols. Although the results are preliminary, their approach is to use a hybrid technique, utilizing efficient generic attributes, which can include deep packet inspection elements and treating them the same way as statistical flow properties.

Cabuk *et al.* [79]-[81] investigated the detection of a class of network covert channels that use the timing of IP packets to transmit a secret message over the network. Their detection scheme used compressibility and showed that we were able to distinguish the covert channel traffic from WWW, FTP-Data, and SSH traffic with more than 95% detection rates. It was noted that more regular the data set, the higher the compressibility. Their results show that the compressibility scores for the SSH, WWW, and FTP-Data data sets are much less than IP simple covert channel compressibility scores.

3 Chapter 3

3.1 Proposed method

In order to classify our proposed method, we have used the dual approach of knowledge – based (often referred to as misuse detection [82], [83]) for detecting specific anomalies and streams-based intrusion detection (often referred to as anomaly detection [82]) for our covert channel profiling. Anomaly detection is usually based on a model of acceptable patterns. The model is obtained either through a learning process [5], [84] or given a priori [85], [86]. The incoming events are then compared with the model. If they deviate from the model, they are considered anomalous. If the model is too sensitive it will produce many false alarms and if its configuration is too lax it will miss attacks. We have not taken the model approach, and for performance and versatility considerations, opted for the signature and protocol based approach. It would be difficult to model any and every new unusual traffic pattern, our approach would consistently be try to learn and constant human interaction would require us to calibrate the model to avoid false positives.

It has been shown that higher connectivity bandwidths and growing numbers of Internet users also lead to increased misuse and anomalous behavior [87]. It has also been shown that unusual traffic patterns may lead to discovery of covert shells that employ packet headers. For example, multiple ping requests within a small time interval may indicate the presence of an ICMP covert shell such as Loki [30]. In addition, covert shells can be detected by observing an anomaly in unused packet header fields [22]. According to Cabuk [79] these schemes have been fairly successful in detecting covert shells but none

can detect the type of covert channels that use packet timings in distributed systems. This is because in such systems packet headers are already protected by policy (i.e., Eve cannot access any information that is transmitted using Alice's network packet headers) and none of the above schemes address detecting anomalous traffic patterns that can potentially be created by a covert channel. It is our opinion that combining anomaly detection and covert channel detection can enable us to ascertain a better qualification of those events.

According to Tombini [86], no component so far can be considered 100% correct and complete, therefore qualification conflicts arise. EMERALD [88] explicitly handles these conflicts through a correlation component.

By analyzing a TCP/IP communication we can identify the type of traffic, its volume and duration, the application and user exchanging it (if it can be determined), the size of the payload and its content, and to some extent protocol compliance to RFC's and other constraints. The proposed method relies on the capability of examining network traffic with regular expressions and packet inspection logic to detect covert channels. The system also contains knowledge accumulated about TCP/IP weaknesses and unusual traffic patterns and can therefore look for attempts to exploit these weaknesses and raise an alarm if such attempts are made. Any action that is not explicitly recognized as anomalous , the system logs pending further offline investigation for potential covert activity whenever there is suspicion. This poses a problem of scalability that we discuss in Chapter 5.

In terms of accuracy, we can consider this part of our proposed method to be acceptable. In terms of completeness, it requires that the system be regularly updated

with new knowledge of attacks and unusual traffic patterns. As an advantage, this type of system has a very low false-alarm rate, given that it is quite precise and the context can be analyzed in great detail [89].

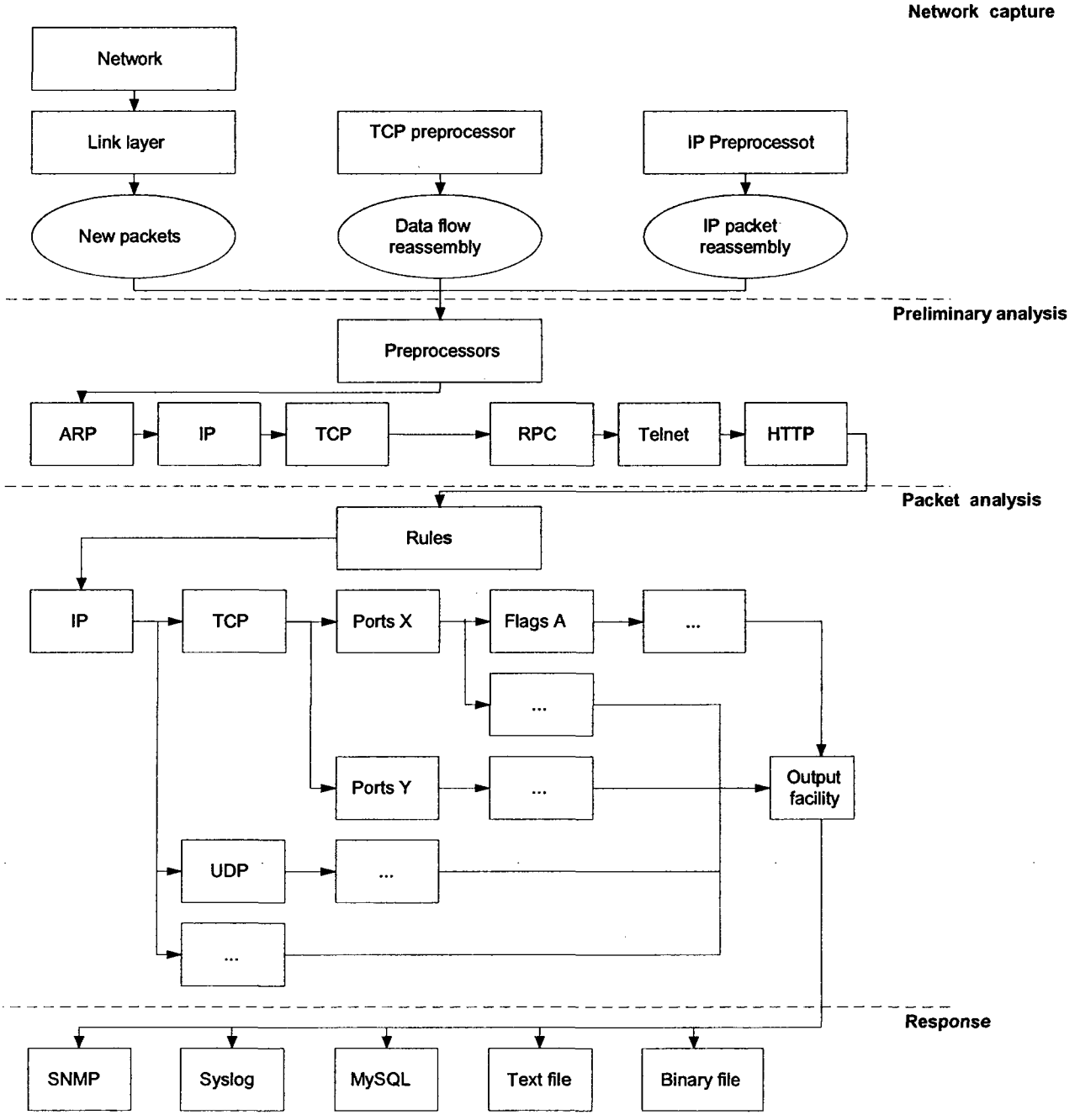


Figure 7: Flow diagram describing four steps to our proposed method: Network capture and reassembly, preliminary analysis via preprocessors, packet analysis via rules and output responses.

3.2 Preprocessor analysis

The proposed method relies on capturing link layer frames from the network and accumulates new packets. Using the TCP and IP preprocessors each IP and TCP packet is reassembled to avoid fragmentation attacks.

Analysis can now begin; the next step is to pass these packets through a series of preprocessors for each of the protocols starting by ARP, IP and TCP. A correctness check is then enabled by a series of more specialized preprocessors, each having a particular purpose that may generate alerts to indicate network's stated security policy violations.

The first specialized preprocessor is the ARP preprocessor. Its purpose is to validate the source Ethernet address with the source IP address of an IP datagram to detect spoofing. We are shall trigger an alert if an IP datagram with an invalid MAC address is detected. This may indicate a misconfigured host or a spoofing attempt.

The second specialized preprocessor is the IP preprocessor which reassembles fragmented IP datagrams into one IP datagram to help analyze its content. Different types of situations are of interest here, namely overlapping fragments (two IP fragments contained data that overlapped) and an out of bounds datagram (A set of IP fragments that would create an IP datagram with a length superior to 65535 bytes if reassembled.

The third specialized preprocessor is the stream preprocessor, which reassembles TCP application data sent in different segments into one continuous buffer to help analyze its content. We are looking for suspicious traffic that might indicate an attack or covert channel. A possible attack could be detected by analyzing the TCP stream and finding a

packet not belonging to any known TCP stream that could be used in stealth scanning. Other types of suspicious traffic could be a packet with the TCP SYN flag set or the TCP SYN and ACK flags set contained in application data. A similar attempt would be a packet with TCP RST flag set. This would violate protocol semantics, because this type of packet simply doesn't exist in the wild.

A particular reassembly evasion attempt exists where two packets contained overlapping data did not match. This is frequently used in reassembly evasion attempt. Here is an example of this type of evasion:

Let's suppose that a detection system has a rule which searches for the string "/php/finger.cgi" to protect a finger script on a web server. An attacker then establishes a connection to this web server and sends 4 TCP segments:

TCP sequence 100: get /p

TCP sequence 107: p/finger.cgi http/1.1

TCP sequence 106: a (doesn't make it to web server because of a low TTL)

TCP sequence 106: h (accepted by web server)

The resulting data seen by a detection system should be "get /pap/finger.cgi http/1.1" which will not trigger the rule. However, our proposed implementation intends to detect and report the overlap to help discover what an attacker really did.

Another type of attack is when two packets having the same sequence numbers have flags which could not have been sent in the order which they were received. This can happen if an attacker sends a TCP packet with the TCP RST or the TCP FIN flag set

and then continues sending data. An attack also exists where application data was sent with a sequence number greater than a TCP FIN or a TCP RST.

Two other conditions exist that should be considered an anomaly, where TCP application data was sent by a host which previously sent an RST and application data was sent with a sequence number greater than a TCP FIN. This can happen if an attacker sends a TCP packet with the TCP FIN flag set and then continues sending data.

3.3 Protocol profiling, sanitization and packet analysis

The second part of the preliminary analysis concerns particular protocol profiling and sanitization. An http preprocessor translates escaped characters in HTTP requests into their equivalent so they can be searched by content filters; the same also applies to the telnet protocol. The RPC preprocessor reassembles fragmented RPC traffic so it can be searched by content filters.

The next step consists of packet analysis; we intend to use rules to filter by keywords by using a simple syntax to achieve our purpose. Covert channels on any of the following protocols (ftp, telnet, http, smtp) will be detected by a variety of metrics. For example, a criterion for clear text protocols (not using any encryption) is the statistical distribution of characters that the protocol semantics demands. For example, normal http traffic is composed of a series of methods and standard responses. Indications of non-RFC compliance and statistical deviations would indicate that the conversation may not in fact be HTTP, but a probable convert channel. In our tests, we have defined a lower limit threshold of eighty percent compliance to protocol RFC's to be reliable.

3.4 Profiling characteristics

As an example, profiling characteristics of a telnet session for example would be comprised of a defined set of characters, small client-to-server packets and known RFC methods. The FTP protocol would also be composed of a defined set of characters, small client to server packets and a defined inactivity period. Here too the methods are known.

Other indicators, such as a high payload size might also indicate that the client is executing http requests but in fact establishing a steady stream of data transfer. This combined with the duration of the connection with a particular site should be sufficient to determine a convert tunnel with a high level of probability.

In general there are three classes of countermeasures: eliminating the covert channel, limiting the covert channel capacity and deterring potential users by demonstrating easy detection of the covert channel [90].

In our proposed method, we intend to demonstrate easy detection with rules to filter by keywords and using a simple syntax to achieve our purpose. Eliminating the channel is impossible, and limiting the channel capacity could be an option once the channel is identified. A QoS mesure could be attributed to the network flow, but this study isn't in the scope of this paper.

Covert channels on any of the following protocols (ftp, telnet, http, smtp) will be detected by a variety of metrics. For example, a criterion for clear text protocols (not using any encryption) is the statistical distribution of characters that the protocol semantics demands. Standard http traffic is composed of a series of methods (GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT) and standard responses. Indications of non-rfc compliance and statistical deviations would indicate that the conversation may not in fact be HTTP, but a probable convert channel. Our approach is

similar to Schear et al. [65] but it isn't as intrusive and disruptive. While interesting, the approach is rather intrusive for it must act as a network bridge at the perimeter network. Interposing another device in a chain of proxies and firewalls that can affect traffic flow complicates thing considerably for large scale corporate networks, especially in terms of interoperability and performance. The vetting process is noteworthy, in that it prevents the transmission of data either overtly in the *payload channel* of layer 7 protocols such as HTTP or FTP or in hidden covert channels in the *protocol channel* of these protocols. Unfortunately, it is also impractical in that no data can be transferred if it hasn't be vetted; a condition that simply cannot be applied in existing corporate networks. Also, the approach towards verification of HTTPS by performing in-line decryption of all traffic raises particular privacy concerns for regular corporate employees. Our approach must be examined further and is a consideration for future work.

3.5 Case: Tunneling

For example, tunneling (which is used to describe when one network protocol called the "payload protocol" is encapsulated within a different protocol) various protocols through port 80 or 443, normally used for HTTP/HTTPS traffic, has become quite common and well documented [91], [92], [93]. Even in the case of an application-layer firewall, the tunneling tool [94] evades suspicion by using correct protocol semantics, for example: an SSH connection tunneled through the HTTP protocol would look like this:

POST /index.html?test=19 HTTP/1.1

Host: example.com:80

Content-Length: 102400

Connection: close

SSH-2.0-OpenSSH_5.0p1

Consequently, the correctness check has become less of a guarantee of security because the protocol is followed and the application-layer firewall cannot discern a disguised query from a legitimate one (a network-layer firewall wouldn't even bother to look at the protocol semantics). In addition, these tunneling tools provide encryption which renders most application layer filters useless. In essence, how can we assure that the traffic that is generated from a host is indeed what the firewall is permitted to pass? If two hosts can communicate, fundamentally, they will be able to transmit and receive information as they please, by covert channel or otherwise. The use of Active Wardens in an attempt to remove, modify or detect any carriers of covert channels [95] was examined, but later dismissed given the unverified results on network traffic and legacy applications. The concept of Minimal Requisite Fidelity, discussed in [96], describes a method for network applications to slightly alter or distort live data passing through them in an attempt to disrupt covert channels.

A distinct possibility exists for violating a security perimeter and bypassing the security controls entirely by tunneling a connection from outside the security perimeter in. The following paper explores the subject in detail [97]. We believe that our proposed method and consequent implementation can partially answer this question.

4 Chapter 4

4.1 Implementation

Our implementation is the creation of a passive monitoring tool. It is a hybrid between a covert channel detection mechanism and an anomaly detector. By capturing link layer frames from the network, it provides the ability to detect known attacks via signatures, to collect statistics, to provide evidence of intrusion and to warn the security administrator in real time of any attack that might be disrupting the network's activities. The tool, named Syncd , can operate in an IP-less mode, meaning that the interface on which it listens has no IP address, but can still retrieve packets on the network. This phantom loghost prevents intruders from sending packets to it in an attempt to destroy the logs.

4.2 Packet backlog

Syncd has a special feature called the packet backlog. When a packet is logged, its characteristics are kept on the packet backlog. If another similar packet is logged on the same network within a predetermined amount of time, then it will not be logged and no message will be produced. When the packet exits the packet backlog, a message is produced telling how often it was repeated. This protects against flooding of messages and packets logs if there is a flood of packet on the network.

4.3 Network capture

Syncd can monitor a number of networks simultaneously and has an enhanced scripting language which permits system administrators to write rules to log suspicious activities and allow normal network traffic. It includes a tool called ipacket which permits

inspection of logged packets using rules from the same scripting language. It can be configured with multiple configuration files to facilitate having a different configuration and a different set of rules for each one of the networks it monitors. Furthermore, packets can branch to secondary rulesets for further inspection after matching a particular rule. It can respond to a matching rule by: sending SNMP traps (this feature also works in IP-less mode), sending a message to syslog or to a file, logging the packet and sending a message to syslog or to a file, executing a program, stopping packets from being processed by the other rules, branching to another ruleset for further inspection, reassembling packets into a TCP stream, and, finally, doing nothing.

4.4 Embedded security features

Syncd can detect many types of network flood and can reassemble fragmented IP datagram and detect IP fragmentation denial of service attacks. Syncd can protect against ARP cache poisoning and can detect unauthorized TCP daemons accepting incoming connections. It can differentiate incoming traffic from outgoing traffic. This permits treating packets differently depending on if they come from the local network or not. Syncd has the possibility of reassembling TCP streams and detects many TCP stream reassembly evasion attempts. It also includes a tool named istream which permits inspection of TCP streams logged because of suspicious activities.

4.5 Summary of special features

Syncd distinguishes itself from other available tools with unique features such as:

- Covert channel detection: Syncd creates stream profiles to detect if transmitted data on HTTP, HTTPS, TELNET, SMTP and FTP ports is regular traffic or a

covert channel.

- ARP and IP spoofing detection: Syncd can detect IP datagrams coming from wrong Ethernet addresses.
- IP fragmentation attacks detection: Syncd can detect overlapping IP fragments.
- TCP streaming evasion detection: Syncd can detect many complex evasion attempts.
- Flexible flood protection: Syncd can be configured to detect virtually any kind of traffic or packet flood to hosts or networks.
- Syncd does not require the libpcap library.

Furthermore, it incorporates many features found in similar tools such as:

- HTTP Unicode decoding
- RPC defragmentation
- TELNET escape code removal in TCP streams
- Flexible logging of messages and packets to a SNMP server, a MySQL database, a text file, a binary packet file and or to syslog. If syncd logs to a MySQL database, any interface such as can be used to view the generated alerts and packets.

4.6 Understanding rulesets

4.6.1 Comments

Comment lines must start with a "#" and finish at the end of the line.

 # This is a comment

This is not # and neither is this

4.6.2 Interface

Interfaces are of Ethernet type and are declared in the following way: "**interface** *name*".

interface le0

will declare the interface named *le0* to be used by **Syncd**.

4.6.3 Include files

Files can be included using the '**include** "*filename*"' command. The filename must be surrounded by quotes.

include "dns-server.rules"

will include the file named *dns-server.rules* in the ruleset.

4.6.4 Variables

Variables are declared using the following format: "**var** *name value*"

var dns_server 10.0.0.2

alert tcp from any any -> $dns_server any (flags: S; msg:"TCP connection to DNS

server";)

is equivalent to *alert tcp from any any -> 10.0.0.2 any (flags: S; msg:"TCP connection to DNS server";)*

4.6.5 General format

The general format of a rule is the following:

ACTION PROTOCOL SOURCE_IP SOURCE_PORT DIRECTION DESTINATION_IP DESTINATION_PORT (FILTERS)

ACTION is one or many actions which will take place if the rule matches. It can be any user-defined ruletype (explained in the next section) or the keyword *drop*. The keyword *drop* is used to stop the packet from being processed by other rules. It can thus be used to let packets only match one rule or to setup rules which will stop normal traffic from being processed by following rules.

PROTOCOL is the packet protocol to be used. It can be *tcp* or *udp* or *ip* or *icmp*

SOURCE_IP is the source IP address to be used. It can be a single IP address such as *1.2.3.4* or a list of bracket-surrounded and comma-separated IPs such as *[1.2.3.4,1.2.3.5]*. Furthermore, netmasks can be applied to IPs such as *1.2.3.0/24* or *1.2.3.0/255.255.255.0* and a negation operator *!* can be added to match any other IP *!5.6.7.8*. These can be combined to select almost any range of IPs *![1.2.3.0/24,1.2.4.1]*.

SOURCE_PORT is the TCP or UDP source port to be used. It can be a single port such as *1024* or a range of ports like *21:25*. The negation operator *!* can also be added to

ports. It must be noted that ranges such as *:1023* would select any port from 0 to 1023 and *50000:* would select any port from 50000 to 65535.

DIRECTION is the direction in which packets are going. It can be -> to indicate that packets must be coming from SOURCE to DESTINATION, <> to indicate that packets can be coming from SOURCE to DESTINATION or DESTINATION to SOURCE and <- to indicate that packets must be coming from DESTINATION to SOURCE.

DESTINATION_IP is the destination IP address to be used. It has the same format as *SOURCE_IP.*

DESTINATION_PORT is the TCP or UDP destination port to be used. It has the same format as *SOURCE_PORT.*

FILTERS is one or many filters described in the following section. These filters must be surrounded by parentheses, even if there is no filter present.

Here are two examples:

warning tcp from !1.2.3.0/24 any to 1.2.3.5 80 ( )

info icmp from any any to 6.7.0.0/16 any (itype: 0; icode: 8; msg: "Echo request";)

The first rule will trigger on any *TCP* segment from *any host not in the 1.2.3.0/24 network* on *any port* to *host 1.2.3.5 on port 80.* There are *no filter present*, so there is *no message defined* for the alert and *no further tests* are performed before triggering the *warning* action.

The second rule will trigger on any ICMP datagram from *any host* to *any host in the*

6.7.0.0/16 network with *ICMP type 0* and *ICMP code 8* (which is an echo request). The message of the alert will be *Echo request* and the *info* action will be triggered.

4.6.6 Ruletype

Actions are defined with the ruletype command which has the following format:

ruletype *name*

{

output-statements

}

where *name* is the name of the action to declare and *output-statements* is a list of output commands (described in the next section) on separated lines. Comments are not allowed inside the braces.

4.6.7 Output

There are currently 5 different output types that can be used by syncd: packet, syslog, message, snmpv1 and database. Each type is different and will log events in a particular way:

packet: receives packets logged

message: receives alerts in text format

syslog: receives alerts in text format

database: log packets that triggered a rule which contains a *sid* and a *rev* field in a database (currently, only MySQL is supported)

snmpv1: sends SNMPv1 traps containing alerts in text format

Here is how they are configured:

output packet: *filename*

where *filename* is the file which will receive packets logged

output message: *filename*

where *filename* is the file which will receive alerts in text format

output syslog: *facility*

where *facility* is the syslog facility which will receive alerts in text format. Available facilities are: *auth*, *cron*, *daemon*, *kern*, *lpr*, *mail*, *news*, *syslog*, *usr*, *uucp*, *local0*, *local1*, *local2*, *local3*, *local4*, *local5*, *local6* and *local7*.

output database: *type_str*, **sensor_name**=*sensor_str* **dbname**=*db_str* **user**=*user_str* **password**=*password_str* **host**=*host_str* **port**=*port_str*

where *type_str* is the database type (which can only be *mysql* for the moment), *sensor_str* is a unique name for the machine on which **Syncd** is running, *db_str* is the name of the database to log to, *user_str* is the user to use to log on the database, *password_str* it the

password to use to log on the database, *host_str* is the host on which the database is running and *port_str* is the port on which to connect on the host.

output snmpv1: community=*community_str* **portdst**=*portdst_str* **ipsrc**=*ipsrc_str* **ipdst**=*ipdst_str*

where *community_str* is the name of the community to use, *portdst_str* is the port to send traps to, *ipsrc_str* is the IP of the machine on which **Syncd** is running and *ipdst_str* is the machine to which SNMPv1 traps should be sent.

A very basic configuration is:

```
ruletype alert
{
        output packet: /var/log/Syncd/packet
        output message: /var/log/Syncd/message
}
```

This will log alerts in text format in */var/log/message* for human-analysis and packets logged in */var/log/packet* for further analysis using **ipacket** if necessary.

4.6.8 Mandatory ruletypes

Syncd allows the user to specify where system messages (such as startup and termination

notices) should be sent. Furthermore, it also allows the user to specify where preprocessor messages should be sent. Syncd thus reserves the following ruletype names for output redirection:

syncd: system message such as startup, termination or possible error messages

stream : stream preprocessor messages

defrag : IP defragmentation preprocessor messages

arp : arp_validate preprocessor messages

http : HTTP decode preprocessor messages

It must be noted that if a preprocessor is used, a corresponding ruletype must be declared or alerts will not be reported. A warning will however be given to the user.

Example of output configuration

This is an example of configuration that could be setup. Here, syslog could use remote logging to send important message to a host monitored by a system administrator (thus, he would receive alerts, system messages and preprocessor alerts). The stream preprocessor, which can generate more false alarms and which is more useful in post-incident analysis if any stream reassembly evasion techniques are used, sends it messages and packets to separate files.

ruletype alert

{

output message: /var/log/Syncdf/messages

output packet: /var/log/Syncdf/messages_packet

output database: mysql, sensor_name=dmz

dbname=user=password=host=10.4.4.2 port=6000

output syslog: local6

}

ruletype warning

{

output message: /var/log/Syncdf/messages

}

ruletype syncd

{

output message: /var/log/Syncdf/system

output syslog: local6

}

ruletype stream

{

output message: /var/log/Syncdf/stream

output packet: /var/log/syndf/stream_packet

}

ruletype arp

{

output message: /var/log/Syncdf/messages

output packet: /var/log/Syncdf/messages_packet

output syslog: local6

}

ruletype http

{

output message: /var/log/Syncdf/messages

output packet: /var/log/Syncdf/messages_packet

output syslog: local6

}

4.6.9 Rules and alerts classes

Rules and the alerts they generate can be categorized in classes to specify their severity. Alert classes must first be defined using the *config classification* command and each rule belonging in a particular category must contain the *classtype* filter (described later on). The format of the *config classification* command is:

config classification: *name, description, priority*;

where *name* is the name of the class, *description* is a short description of the class and *priority* is the priority of the class.

4.6.10 Filters

Here is a list of supported filter keywords: *msg, flags, seq, ack, window, itype, icode, icmp_id, icmp_seq, id, tos, sameip, ip_proto, ipopts, ttl, fragbits, content, uricontent, nocase, dsize, offset, depth, rpc, floodp, floodb, classtype, sid, rev* and *reference*. Each of them is describe in the next section.

A colon, its arguments and a semi-colon must follow every filter that takes one or more arguments.

4.6.10.1 *msg* filter

The *msg* filter specifies the message corresponding to a rule: it is this message that will appear in logs when a rule triggers an alert. This filter must be followed by a quote-surrounded message and its format is:

msg: "*message*";

where *message* is the message to be used. As an example:

warning udp $external any -> $web_server 514 (msg: "Remote syslog datagram";)

An UDP datagram that triggers this rule would create a log message similar to this:

08-07-18 12:01:55 | Remote syslog datagram (UDP 20/8/120 4.3.3.4:1050 -> 4.2.1.3:514)

[19]

4.6.10.2 *flags* filter

The *flags* filter checks TCP flags of TCP segments using an operator and a flags mask. This filter's format is

flags: <*operator*> *flag_list*;

where *operator* is an optional operator described next and *flag_list* is a list of the following flags:

F	FIN flag
S	SYN flag
R	RST flag
P	PSH flag
A	ACK flag
U	URG flag
1	Reserved flag bit #1
2	Reserved flag bit #2

0 No flags

By default, flags are checked for exact match. This behavior can, however, be modified using one of the following operators:

+ Other flags than the ones specified may be set

* At least one flag in the ones specified must be set

! No flags specified must be set

As an example,

warning tcp $external any -> $web_server 80 (msg: "Connection attempt to web server"; flags: S;)

warning tcp $external any -> $telnet_server 23 (msg: "URG data to telnet server"; flags: +U;)

*warning tcp $external any -> $internal any (msg: "Reserved TCP flags set"; flags: *12;)*

The first rule looks for packets with only the SYN flag while the second rule searches packets with the URG flag set (any other flags may be set). Finally, the last rule checks for packet which have at least the 1 or 2 flag set.

4.6.10.3 *seq* filter

The *seq* filter checks TCP segments for a specific sequence number. The *seq* filter's format is:

seq: *number*;

where *number* is a sequence number between 0 and 2^32-1. As an example,

warning tcp $external any -> $internal any (msg: "TCP segment with sequence number 1234567"; seq: 1234567;)

This rule looks for any TCP segment with an sequence number of 1234567.

4.6.10.4 *ack* filter

The *ack* filter checks TCP segments for a specific acknowledgement number. This filter's format is:

ack: *number*;

where *number* is an acknowledgement number between 0 and 2^32-1. As an example,

warning tcp $external any -> $internal any (msg: "TCP segment with acknowledgement number 7654321"; ack: 7654321;)

This rule looks for any TCP segment with an acknowledgement number of 7654321.

4.6.10.5 *window* filter

The *window* filter checks TCP segments for a specific window size. The format of this filter is:

window: *size*;

where *size* is a window size between 0 and 65535. As an example,
warning tcp $external any -> $internal any (msg: "TCP segment with window size of 666"; window: 666;)

This rule looks for any TCP segment with a window size of 666.

4.6.10.6 *itype* filter

The *itype* filter checks ICMP datagrams for a specific ICMP type. The format of the *itype* filter is:

itype: *type*;

where *type* is an ICMP type between 0 and 255. As an example,

warning icmp $external any -> $internal any (msg: "ICMP destination unreachable"; itype: 3;)

This rule looks for any ICMP datagram with a type of 3.

4.6.10.6.1 icode filter

The *icode* filter checks ICMP datagrams for a specific ICMP code. The format of the *icode* filter is:

icode: *code*;

where *code* is an ICMP code between 0 and 255. As an example,

warning icmp $external any -> $internal any (msg: "ICMP echo request"; icode: 0; itype: 8;)

This rule looks for any ICMP datagram with a type of 8 and a code of 0.

4.6.10.7 *icmp_id* filter

The *icmp_id* filter checks ICMP echo datagrams (ICMP type of 8 or 0 and ICMP code of 0) for a specific identification number. The format of this filter is:

icmp_id: *id*;

where *id* is an identification number between 0 and 65535. As an example,

warning icmp $external any -> $internal any (msg: "ICMP datagram with ID of 666"; icmp_id: 666;)

This rule looks for any ICMP echo datagram with an identification number of 666.

4.6.10.8 icmp_seq filter

The *icmp_seq* filter checks ICMP echo datagrams (ICMP type of 8 or 0 and ICMP code of 0) for a specific sequence value. The format of the *icmp_seq* filter is:

icmp_seq: *seq*;

where *seq* is a sequence number between 0 and 65535. As an example,

warning icmp $external any -> $internal any (msg: "ICMP datagram with sequence of 65535"; icmp_seq: 65535;)

This rule looks for any ICMP echo datagram with a sequence number of 65535.

4.6.10.9 *id* filter

The *id* filter checks IP datagrams for a specific identification number. This filter's format is:

id: *number*;

where *number* is an identification number between 0 and 65535. As an example,

warning tcp $external any -> $internal any (msg: "TCP segment with IP ID of 0"; id: 0;)

This rule looks for TCP segments with an IP identification number of 0.

4.6.10.10 ip_proto filter

The *ip_proto* filter checks IP datagrams for a specific protocol number. The format of this filter is:

ip_proto: *protocol*;

where *protocol* is a protocol number between 0 and 255. As an example,

warning ip $external any -> $internal any (msg: "IP protocol 129"; ip_proto: 129;)

This rule looks for IP datagrams with an IP protocol number of 129.

Usual protocol numbers:

1 ICMP

6	TCP
17	UDP

4.6.10.11 *ipopts* filter

The *ipopts* filter checks IP datagrams for a specific IP option. This filter's format is:

ipopts: *option*;

where *option* must be one of the following:

rr	Record route
eol	End of list
nop	No operation
ts	Timestamp
sec	IP sec
lsrr	Loose source routing
ssrr	Strict source routing
satid	Stream identifier

As an example,

warning ip $external any -> $internal any (msg: "IP datagram with record route option";

ipopts: rr;)

This rule looks for IP datagrams with the "record route" IP option.

4.6.10.12 *ttl* filter

The *ttl* filter evaluates IP datagrams' time-to-live value with a specified operator and value. The format of this filter is:

ttl: <*operator*> *value*;

where *operator* is an optional argument which may be < or > (it is = by default) and *value* is the TTL value to compare against. As an example,

warning ip $external any -> $internal any (msg: "IP datagram with very low TTL"; ttl: < 3;)

This rule looks for IP datagrams with a TTL smaller than 3.

4.6.10.13 fragbits filter

The *fragbits* filter checks fragmentation flags of IP datagrams using an operator and a flags mask. The format of the *fragbits* filter is:

fragbits: <*operator*> *flag_list*;

where *operator* is an optional operator described next and *flag_list* is a list of the following flags:

M More fragments

D Don't fragment

R Reserved flag

By default, flags are checked for exact match. This behaviour can, however, be modified using one of the following operators:

\+ Other flags than the ones specified may be set

* At least one flag in the ones specified must be set

! No flags specified must be set

As an example,

warning ip any any -> any any (msg: "Reserved bit set in IP datagram"; fragbits: R;)

This rule looks for any IP datagram with the reserved bit set.

4.6.10.14 *dsize* filter

The *dsize* filter checks packets' payload length using an operator and a size. The format of this filter is:

dsize: <*operator*> *size*;

where *operator* is an optional argument which may be < or > (it is = by default) and *sizee* is the payload size to compare against. As an example,

warning icmp $external any -> $internal any (msg: "ICMP datagram with payload > 1000 bytes"; dsize: > 1000;)

This rule looks for ICMP datagrams with a payload of more than 1000 bytes.

4.6.10.15 *content* filter

The *content* filter searches packets' payload for a particular string. The format of the *content* filter is:

content: "*string*";

where *string* is the string to search. The following quote " character, the pipe | character and the semicolon ; character must be escaped. Binary data or escaped characters must be enclosed in pipe | characters and their value specified in hexadecimal. As an example,

warning tcp $external any -> $internal 80 (msg: "x86 nops sent to web server"; content: " |90909090|";)
warning tcp $external any -> $internal 80 (msg: "finger.cgi script access"; content: "finger.cgi";)

The first rule checks the payload of TCP segments for the 4 consecutive bytes 90h 90h 90h 90h while the second searches for the string *finger.cgi*.

4.6.10.16 uricontent filter

The *uricontent* filter is identical to the content filter, except that only URI portion of a request in a payload is searched. Please refer to the *content* filter for format.

4.6.10.17 *nocase* filter

The *nocase* filter affects the behavior of the last *content* or *uricontent* filter specified by making comparisons case insensitive. The format of this filter is:

nocase;

As an example,

warning icmp $external any -> $internal any (msg: "The two strings matches"; content: "Some string"; content: "another string"; nocase;)

This rule searches ICMP datagrams for the string "Some string" in a case sensitive

fashion and then searches again for the string "another string" in a case insensitive fashion.

4.6.10.18 *offset* filter

The *offset* filter affects the behavior of the last *content* or *uricontent* filter specified by starting any string search at a particular offset into the payload. The format of this filter is:

offset: *value*;

where *value* is the offset minus 1 from which to start a string search. As an example,

warning icmp $external any -> $internal any (msg: "Suspicious string found in ICMP datagram at byte 11 or later"; content: "Suspicious"; offset: 10;)

This rule searches ICMP datagrams for the string "Suspicious" from the 11^{th} (10+1) character of the payload until its end.

4.6.10.19 *depth* filter

The *depth* filter affects the behavior of the last *content* or *uricontent* filter specified by

stopping any string search after a particular number of characters inspected. The format of this filter is:

depth: *value*;

where *value* is the number of characters to be inspected. As an example,

warning icmp $external any -> $internal any (msg: "Suspicious string found in ICMP datagram"; content: "Suspicious"; offset: 10;)

This rule searches ICMP datagrams for the string "Suspicious" from the 11^{th} (10+1) character of the payload until its end.

4.6.10.20 *rpc* filter

The *rpc* filter checks UDP or TCP packets for RPC requests matching particular application, procedure and version. The format of the *rpc* filter is:

rpc: *application*,*procedure*,*version*;

where *application* is an application number, *procedure* is a procedure number or * to indicate any procedure and *version* is a version number or * to indicate any version. As an example,

warning udp any any -> $rpc_server 111 (msg: "RPC getport request"; rpc: 100000,*,*;
)

This rules cheks every UDP datagram going to a RPC server for getport requests.

4.6.10.21 *floodp* filter

The *floodp* filter checks for packet floods. It is a special type of filter: every time a rule in which it is present matches a packet, its count is increased by 1. If the count, during a 10 second period, exceeds the limit specified by the argument to the *floodp* filter, the message contained in the *msg* filter of the rule is sent to the *Syncd* ruletype. Any ruletypes specified in the action field of the rule will not be executed except for drop actions. The format of this filter is:

floodp: *count*;

where *count* is the maximum packet number before an alert is triggered. It is recommended to define an empty ruletype to use for *floodp* and *floodb* filters instead of using another user-defined ruletype which will not be used anyway. As an example,

ruletype none
{
}
none tcp any any -> $web_server 80 (flags: S; floodp: 500; msg: "SYN flood to web

server";)

This rule counts TCP segments with the SYN flag set going to the a web server. If more than 500 packets are counted during a 10 second period, a message is sent to the *Syncd* ruletype. A flood message might look like this:

08-07-01 11:52:56 | SYN flood to web server (516 packets totaling 39876 bytes in 10 seconds)

This alert means that 516 packets matching the rule in question (TCP segments to a web server on port 80 and with the SYN flag set) were detected in a 10 second period. These 516 packets made up 39876 bytes of traffic.

It must be noted that *floodp* and *floodb* filters can be combined in a single rule.

4.6.10.22 *floodb* filter

The *floodb* filter checks for traffic floods. It is a special type of filter: every time a rule in which it is present matches a packet, its count is increased by the total size of the packet (including the link layer). If the count, during a 10 second period, exceeds the limit specified by the argument to the *floodb* filter, the message contained in the *msg* filter of the rule is sent to the *Syncd* ruletype. Any ruletypes specified in the action field of the rule will not be executed except for drop actions. The format of the *floodb* filter is:

floodb: *bytes*;

where *bytes* is the maximum number of bytes before an alert is triggered. It is recommended to define an empty ruletype to use for *floodp* and *floodb* filters instead of using another user-defined ruletype which will not be used anyway. As an example,

```
ruletype none
{
}

none icmp any any -> any any ( floodb: 1000000; msg: "ICMP flood on network"; )
```

This rule matches every ICMP datagrams and calculates their total size. If more than 1000000 bytes are counted during a 10 second period, a message is sent to the *Syncd* ruletype. A flood message might look like this:

```
08-07-01 11:57:40 | ICMP flood on network (800 packets totaling 1009560 bytes in 10 seconds)
```

This alert means that 800 packets matching the rule in question (any ICMP datagram) were detected in a 10 second period. These 800 packets made up 10009560 bytes of traffic.

4.6.10.23 classtype filter

The *classtype* filter specifies which class an alert belongs and allows classification of alerts in different severity categories. The format of the *classtype* filter is:

classtype: *name*;

where *name* is the name of the class an alert belongs to (these can be declared using the *config classification* command). As an example,

warning ip any any -> any any (msg: "Reserved bit set in IP datagram"; fragbits: R; classtype: unknown;)

This rule classifies logged IP datagrams with the reserved bit set in the unknown category.

4.6.10.24 *sid* filter

The *sid* filter specifies a unique identification number to each rule. Its format is:

sid: *number*;

where *number* is the identification number of the rule.

4.6.10.25 *rev* filter

The *rev* filter specifies a revision number for a particular rule. Its format is:

rev: *number*;

where *number* is the revision number of the rule.

4.6.10.26 reference filter

The *reference* filter allows references to be added to a rule. These references provide information about a particular rule and may be in any of the following known reference systems: *bugtraq*, *arachnids*, *CVE*, *mcaffee* or *url* (which specifies a URL to rule-related information). The format of this filter is:

reference: *system*;

where *system* is the reference system of the rule.

4.7 Preprocessors

Preprocessors perform special operations on application data which help the system recognize particular attacks. Each preprocessor has its particular purpose and may generate alerts to indicate suspicious activities.

4.7.1 arp_validate preprocessor

The arp_validate preprocessor validate the source Ethernet address with the source IP address of an IP datagram to detect spoofing. It is used with the following syntax:

"**preprocessor arp_validate:** *filename*" where *filename* is the name of the preprocessor configuration file containing rules to be applied to IP datagrams. This file, which may content line-comments and use variables, has a special format which is:

POLICY SOURCE_IP **from** *SOURCE_ETHERNET*

POLICY is **allow:** or **deny:** depending on whether or not you wish to allow traffic that matches the rule to be allowed or denied.

SOURCE_IP is one or a list of IP addresses to be used as source IP address for packets inspected. It is in the same format as *SOURCE_IP* and *DESTINATION_IP* used in the general rule format.

SOURCE_ETHERNET is one or a list of Ethernet addresses to be used as source Ethernet address for packets inspected. It can be a single Ethernet address such as *01:b4:05:ef:30:99* or a list of bracket-surrounded and comma-separated Ethernet addresses such as *[12:23:34:45:56,aa:bb:cc:dd:ee:ff]*. Furthermore, a negation operator can be added to match any other Ethernet address such as *!05:06:07:08:09:0a*.

As an example, let's consider a detection system listening on a DMZ with a router (Ethernet address 01:01:01:02:02:02) connected to the Internet, another router (Ethernet address 04:04:04:03:03:03) connected to a protected network (IP addresses of 1.2.4.0/24). Furthermore, the servers on the DMZ are in the 1.2.5.0/24 network. To protect against spoofing, the preprocessor configuration file should be:

# # # # #

Sample arp_validate preprocessor configuration file

Allow external traffic from the router connected to the Internet

allow: !1.2.4.0/23 from 01:01:01:02:02:02

Allow internal traffic from the router connected to the protected network

allow: 1.2.4.0/24 from 04:04:04:03:03:03

Allow DMZ server traffic from any other Ethernet address than those of the routers

allow: 1.2.5.0/24 from ![01:01:01:02:02:02,04:04:04:03:03:03]

Deny everything else

deny: any from any

4.7.2 defrag preprocessor

The defrag preprocessor reassembles fragmented IP datagrams into one IP datagram to help analyze its content. It is used with the following syntax: "**preprocessor defrag: memcap:** *max_memory*" where *max_memory* is the maximum amount of memory that may be allocated by the preprocessor. The "**memcap:** *max_memory*" argument is

optionnal and its default is 8000000 bytes.

The defrag preprocessor logs its alerts to the *defrag* ruletype. It is recommended to use the arp_validate preprocessor if the defrag preprocessor is used.

4.7.3 stream preprocessor

The stream preprocessor reassembles TCP application sent in different segments into one continuous buffer to help analyze its content. It is used with the following syntax: "**preprocessor stream: memcap:** *max_memory* , **timeout:** *timeout_delay* , **flush:** *flush_size*, **port:** *portlist*" where *max_memory* is the maximum amount of memory that may be allocated by the preprocessor, *timeout_delay* is the time after which a TCP connection will be declared dead, *flush_size* is the maximum number of bytes of application data that is kept before a packet is reassembled (if a TCP segment contains more application data than flush_size, it might not go through the packet reassembly, so it will not be profiled) and its content flushed and *portlist* is a list of space-separated port numbers (or the keyword *any*) on which TCP connections will be monitored. The "**memcap:** *max_memory*" argument is optional and its default is 16000000 bytes. The "**timeout:** *timeout_delay*" argument is optional and its default is 120 seconds. The "**flush:** *flush_size*" argument is optional and its default is 128 bytes. The "**port:** *portlist*" argument is optional and its default is no port (which means nothing will be done).

To permit packets to be reassembled, another command must be used: "**preprocessor stream_reassemble: source:** *src_portlist* , **destination:** *dst_portlist*" where *src_portlist*

is the list of TCP source ports from which packets must be to be reassembled and *dst_portlist* is the list of TCP destination ports to which packets must be going to be reassembled (it must be noted that these two lists are applied in an "OR fashion", meaning that packets must only be in one of the two lists to be reassembled). The "**source:** *src_portlist*" and the "**destination:** *dst_portlist*" arguments are optionnal and their default is no port (which means nothing will be done).

As an example, a system administrator wishing to reassemble TCP streams of clients connecting to port 21, 23, 25, 80, 110 and 111 and of server responses on ports 80 and 443 would setup the preprocessor like this:

preprocessor stream: port: 21 23 25 80 110 111 443

preprocessor stream_reassemble: source: 80 443 , destination: 21 23 25 110 111

The stream preprocessor logs its alerts to the *stream* ruletype. It is recommended to use the defrag preprocessor and the arp_validate preprocessor if the stream preprocessor is used.

4.8 Streams profiling

Stream profiling is automatically enabled if the stream preprocessor is used. However, stream profiling is only performed on packets reassembled. Each type of profiling needs

a different kind of packets to be reassembled. Here is a list:

HTTP and HTTPS profiling: TCP source ports 80 and 443

SMTP profiling: TCP destination port 25

telnet profiling: TCP destination port 23

FTP profiling: TCP destination port 21

4.8.1 http_decode preprocessor

The http_decode preprocessor translates escaped caracters in HTTP requests into their equivalent so they can be searched by *content* filters. It is used with the following syntax: "**preprocessor http_decode:** *portlist*" where portlist is a list of space-separated destination TCP ports. Any TCP application data going to these ports will be verified for escaped caracters.

The http_decode preprocessor logs its alerts to the *http* ruletype. It is recommended to use the stream preprocessor if the http_decode preprocessor is used.

4.8.2 telnet_decode preprocessor

The telnet_decode preprocessor removes escape sequences from TCP application data going to ports 21 and 23 so it can be searched by *content* filters. It is used with the following syntax: "**preprocessor telnet_decode**".

The telnet_decode does not generate alerts. It is recommended to use the stream preprocessor if the telnet_decode preprocessor is used.

4.8.3 rpc_decode preprocessor

The rpc_decode preprocessor reassembles fragmented RPC traffic so it can be searched by *content* filters. It is used with the following syntax: "**preprocessor rpc_decode:** *portlist*" where portlist is a list of space-separated destination TCP ports. Any TCP application data going to these ports will be inspected.

4.9 Log output

4.9.1 Log entries

Here is an example of text format log entry that would be sent to a message file, in a SNMPv1 trap or to syslog:

08-07-17 16:02:03 | SCAN Proxy attempt (TCP 20/28/0 1.2.3.4:1024 -> 5.6.7.8:1080 ----
--S-) [45]

This means the alert occurred on July 17th 2008 at 16:02 and 3 seconds. The alert was a proxy scan attempt. The protocol used was TCP, the IP header length was 20, the TCP header length was 28 and there was 0 bytes of application data. The scan was from 1.2.3.4 port 1024 to 5.6.7.8 port 1080. There was a TCP flag set, the SYN flag. The number in brackets at the end indicates that this packet was logged in a packet file with an identification number of 45. The packet can now be inspected using the ipacket tool (described later).

4.9.2 Stream preprocessor alerts

Here is a list of possible alerts generated by the stream preprocessor and their meaning. These types of unusual traffic patterns may lead to the discovery of covert channels.

SP1/possible-stealth-port-scan: A packet not belonging to any known TCP stream and that could be used in stealth scanning was detected.

SP2/data-on-syn: A packet with the TCP SYN flag set contained application data.

SP3/data-on-syn-ack: A packet with the TCP SYN and ACK flags set contained application data.

SP5/data-on-rst: A packet with the TCP RST flag set contained application data.

SP6/data-mismatch: Two packets contained overlapping data which did not match. This is frequently used in reassembly evasion attempt.

SP7/same-seq-different-flags: Two packets having the same sequence numbers had flags which could not have been sent in the order which they were received. This can happen if an attacker sends a TCP packet with the TCP RST or the TCP FIN flag set and then continues sending data.

SP9/data-past-termination: Application data was sent with a sequence number greated than a TCP FIN or a TCP RST. This can happen if an attacker sends a TCP packet with the TCP RST or the TCP FIN flag set and then continues sending data.

SP10/stream-alert: TCP packets reassembled after a stream has generated an alert are automatically logged for inspection.

SP11/data-after-rst: TCP application data was sent by a host which previously sent an RST.

SP13/data-past-fin: Application data was sent with a sequence number greated than a

TCP FIN. This can happen if an attacker sends a TCP packet with the TCP FIN flag set and then continues sending data.

SP14/covert-channel-over-ftp: A TCP connection on port 21 contained data which did not look like regular FTP traffic.

SP15/covert-channel-over-telnet: A TCP connection on port 23 contained data which did not look like regular TELNET traffic.

SP16/covert-channel-over-http: A TCP connection on port 80 or 443 contained data which did not look like regular HTTP or HTTPS traffic. False alarms are most often the result of non RFC-compliant HTTP servers. When this happens, a very quick inspection of the application data of the packet logged will tell immediately if the data is correct or not.

SP17/covert-channel-over-smtp: A TCP connection on port 25 contained data which did not look like regular SMTP traffic.

4.9.3 IP defragmentation preprocessor alerts

Here is a list of possible alerts generated by the IP defragmentation preprocessor and their meaning:

DP1/overlapping-fragments: Two IP fragments contained data that overlapped.

DP2/datagram-too-big: A set of IP fragments would create an IP datagram with a length superior to 65535 bytes if reassembled.

4.9.4 ARP preprocessor alerts

Here is a list of possible alerts generated by the ARP preprocessor and their meaning:

AP1/invalid-ARP-address: An IP datagram with an invalid ARP address was detected. This may indicate a misconfigured host or a spoofing attempt.

4.9.5 Ipacket tool

Ipacket is a tool created to inspect packets that generated an alert and were logged in a packet file. The tool can be used to dump packet payload, complete packet or a specific packet ID in various formats, and display each byte of application data in any specified chronological order.

5 Chapter 5

5.1 Performance evaluation

In our approach, network traffic is tapped at the perimeter boundary, typically in line with the corporate firewall. The traffic can be recorded and processed at different levels of granularity, from complete packet-level traces to statistical figures. Compared to other approaches that use active capture [65], only passive capture is used, since it is best suitable for analysis of Internet backbone traffic properties. Our passive traffic capture method is software-based and we have modified the Solaris operating system and device in order to obtain copies of network packets. This approach is inexpensive and offers good adaptability; we recognize that its possibilities to measure traffic on high speed networks are limited to 10 Gbit/s line speeds [96]. A hardware-based method was examined, but determined to be rather costly and less versatile.

Once network data is collected, it is processed online or in 'real time' and also stored offline for non time critical events and suspicious network data. This offers the possibility to correlate network traffic collected at different times and different locations. Our passive traffic capture operates on different protocol layers, the Internet Protocol (IP), located on the network layer and transport layer protocols, especially TCP and UDP, and application layer protocols are our main focus. Data gathered on different protocol layers can present different levels of granularity.

Our approach provides packet-level traces that include all information of each packet observed on a specific host or link.

Certain challenges that we faced in our approach is scalability. We realized that our packet capture and analysis was strictly limited by hardware performance, storage and processing capabilities, but we have taken steps to mitigate this problem. First, average backbone link throughput are far from line speed, and secondly, we are filtering only packets with specific properties. Another problem, and as shown in [98], is that routing symmetry on highly aggregated links is rare, which means that bi-directional flow data can no longer be assumed. We have yet to address this problem.

Our test setup consists of a sever running Solaris 10 with 8 core 1.2 GHz UltraSPARC T1 processors, 32 GB of memory, and gigabit Ethernet interfaces. The server is connected inline using a network tap with the corporate network security perimeter and follows the guidelines set by John [99] for passive internet traffic measurements. It is non-intrusive and can run continuously and provide results at any time scale. With this setup, we were able to scale to 3 Gbit/s without starting to drop packets and lessen our detection rate. This is the threshold we are considering the approximate maximum practical analysis rate for this hardware and this configuration and rulesets.

6 Conclusions and Future work

With increased measures being implemented to counter the more obvious security issues – such as the free transfer of USB sticks in- and out of an organization, the use of covert channels is certain to rise. In addition to this threat, it is important to realize that covert channels could also lead to direct circumvention of network security policies causing a potentially dangerous security incident.

This work presents a flexible and passive approach of profiling a multitude of covert TCP/IP channels. Specifically, the approach maintains client-server transparency because it passively captures link layer frames, thus not interfering with network throughput. The flexibility stems from the ability to be granular and use regular expression for many fields in the network flows. The approach consists of using covert channel detection schemes combined with anomalous network activity detection. We believe that this hybrid approach complements itself well and can be a basis for adding new and refined techniques in both the covert channel research area and the anomalous network activity area. Future work could include evaluating and integrating new HTTP preprocessors that use compression techniques [79] or the SPID algorithm [78] to detect covert HTTP more precisely and efficiently. New filters could be implemented that capture recent unusual network traffic patterns or detection evasion techniques.

We believe this approach will appeal to companies who are sensitive to these types of security incidents and are looking for a cost effective way to mitigate the risk of covert channels. Presently, large scale implementations of SIEM'S (security information and event management) are used to correlate many sources of logs in an attempt to discover these types of covert channels.

References

[1] C. G. Girling. Covert Channels in LAN's. *IEEE Transactions on Software Engineering*, SE-13(2):292-296, February 1987.

[2] T. Handel, M. Sandford. "Hiding Data in the OSI Network Model," in: *Proceedings of the First International Workshop on Information Hiding*, pages 23-38, 1996.

[3] S. Zander, G. Armitage, P. Branch. "Covert Channels in Multiplayer First Person Shooter Online Games," in: *Proceedings of 33rd Annual IEEE Conference on Local Computer Networks (LCN)*, October 2008.

[4] S. Zander, G. Armitage, P. Branch. Covert Channels and Countermeasures in Computer Network Protocols. *IEEE Communications Magazine*, 45(12):136-142, December 2007.

[5] M. Van Horenbeeck. "Deception on the Network: Thinking Differently About Covert Channels," in: *Proceedings of 7th Australian Information Warfare and Security Conference*, December 2006.

[6] C.E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi, "A taxonomy of computer program security flaws," in: *ACM Computing Surveys, vol. 26, no.4, pp. 211-254, Sept. 1994.*

[7] G. J. Simmons, "The Prisoners' Problem and the Subliminal Channel," in: *Proc. Advances in Cryptology*, 1983, pp. 51–67.

[8] S. Craver, "On Public-Key Steganography in the Presence of an Active Warden," in: *Proc. 2nd Int'l. Wksp. Info. Hiding*, Apr. 1998, pp. 355–68.

[9] J. Sawinski, (2004, March). Reverse Tunneling Techniques: theoretical requirements for the GW implementation. [Online], Available: http://www.gray-world.net/projects/papers/rtt.txt

[10] Jon Postel. RFC 791: Internet Protocol: DARPA Internet Program Protocol Specification, September 1981.

[11] Jon Postel. RFC 792: Internet Control Message Protocol, September 1981.

[12] J. Postel. RFC 768: User Datagram Protocol, August 1980.

[13] Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek. An Advanced 4.3BSD Interprocess Communication Tutorial. Computer Systems Research Group, Deptartment of Electrical Engineering and Computer Science, University of California, Berkeley, 1986

[14] T. Dierks and E. Rescorla. RFC 4346: The Transport Layer Security (TLS) Protocol, Version 1.1), April 2006.

[15] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol, December 2005.

[16] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, December 1998.

[17] S.M. Bellovin and W.R. Cheswick, "Network Firewalls," in: *IEEE Communications MAGAZINE, vol. 32, no. 9, pp. 50-57, Sept. 1994.*

[18] T. Porter. (2005, January). The Perils of Deep Packet Inspection. [Online], Available: http://www.securityfocus.com/infocus/1817

[19] I. Dubrawsky. (2003, July). Firewall Evolution – Deep Packet Inspection. [Online], Available: http://www.securityfocus.com/infocus/1716

[20] M. J. Ranum. (2005, May). What is "Deep Inspection"? [Online], Available:http://www.ranum.com/security/computer_security/editorials/deepinspect

[21] G. R. Malan *et al.*, "Transport and Application Protocol Scrubbing," in: *Proc. IEEE Conf. Computer Communications (INFOCOM)*, Mar. 2000, pp. 1381–90.

[22] M. Handley, C. Kreibich, V. Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in: Proceedings of 10th USENIX Security Symposium, August 2001.

[23] G. Fisk, M. Fisk, C. Papadopoulos, J. Neil, "Eliminating steganography in internet traffic with active wardens," in: Proc. of the 2002 International Workshop on Information Hiding. Oct. 2002.

[24] G. Vigna, "A topological characterization of TCP/IP security." Dipartmento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vonci, 20133 Milano, Italy, December 1996.

[25] S. M. Bellovin, "Security problems in the TCP/IP protocol suite," in: *Computer Communication Review*, vol. 19, pp. 32–48, April 1989.

[26] Oliver Friedrichs, A simple TCP spoofing attack. Security Advisory of Secure Networks Inc. February 10, 1997.

[27] M. de Vivo, G. O. de Vivo, and G. Isern, "Internet Security Attacks at the Basic Levels" in: *ACM SIGOPS Operating Systems Review, vol. 32 no. 2, pp. 4–15, April 1998.*

[28] P. Mudge, "ICMP Router Discovery Advisory", L0pht heavy industries, August 11, 1999.

[29] T. H. Ptacek and T. N. Newsham. (1998, January). Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. [Online]. Available: http://insecure.org/stf/secnet ids/secnet ids.html

[30] daemon9. LOKI2: The Implementation. *Phrack Magazine*, 7(51), September 1997.

[31] D. Stodle, (2009, January). Ping Tunnel. [Online], Available: http://www.cs.uit.no/~daniels/PingTunnel/

[32] I. Zelenchuk, "Skeeve — ICMP Bounce Tunnel," 2004, http://www.gray-world.net/poc_skeeve.shtml

[33] P. Padgett, "Corkscrew," 2001, http://www.agroman.net/corkscrew/

[34] A. Dyatlov, "Firepass — Is a Tunneling Tool," 2003, http://gray-world.net/pr_firepass.shtml

[35] P. LeBoutillier, "HTTunnel," 2005, http://sourceforge.net/projects/httunnel/

[36] M. Lundström, "MailTunnel," http://gray-world.net/tools/mailtunnel-0.2.tar.gz

[37] SynAckLabs,(2003, May). Stegtunnel. [Online], Available: http://www.synacklabs.net/OOB/stegtunnel.html

[38] D. Stodle, (2009, January). Ping Tunnel. [Online], Available: http://www.cs.uit.no/~daniels/PingTunnel/

[39] Ka0ticSH. Diggin Em Walls (part 3) - Advanced/Other Techniques for ByPassing Firewalls, April 2002. http://neworder.box.sk/newsread.php?newsid=3957.

[40] A. Vidstrom. ACK Tunneling Trojans, 2000. http://ntsecurity.nu/papers/acktunneling/

[41] A. Hintz. Covert Channels in TCP and IP Headers, 2003. http://www.defcon.org/images/defcon-10/dc-10-presentations/dc10-hintz-covert.ppt.

[42] F. Heinz,(2004, June). NSTX (the Nameserver Transfer Protocol). [Online], Available: http://nixbit.com/cat/system/networking/nstx/

[43] S. Castro, (2003, August). Covert Channel Tunneling Tool. [Online], Available: http://www.gray-world.net/pr_cctt.shtml

[44] A. Dyatlov, S. Castro. Exploitation of Data Streams Authorized by a Network Access Control System for Arbitrary Data Transfers: Tunneling and Covert Channels over the HTTP Protocol. Technical report, Gray-World, June 2003. http://gray-world.net/projects/papers/covert_paper.txt.

[45] Henry, P. A. 2000. Covert channels provided hackers the opportunity and the means for the current distributed denial of service attacks. Tech. rep., CyberGuard Corporation.

[46] Abad, C. 2001. IP checksum covert channels and selected hash collision. Tech. rep., University of California.

[47] Ahsan, K. 2000. Covert channel analysis and data hiding in TCP/IP. M.S. thesis, University of Toronto.

[48] Ahsan, K. and Kundur, D. 2002. "Practical data hiding in TCP/IP," in: *Proceedings of the Workshop on Multimedia Security (MMSEC'02)*, pp. 63-70.

[49] Matthias Bauer, "New covert channels in HTTP: adding unwitting Web browsers to anonymity sets," in: *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, October 30, 2003, Washington, DC.

[50] Giffin, J., Greenstadt, R., Litwack, P., and Tibbetts, R. 2002. "Covert messaging through TCP timestamps," in: *Proceedings of the Workshop on Privacy Enhancing Technologies (PET'02)*, 2482, 194--208.

[51] Hauser, V. 1999. Placing backdoors through firewalls. Tech. rep., The Hacker's Choice.

[52] Li, S. and Ephremides, A. 2004. "A network layer covert channel in ad-hoc wireless networks," in: *Proceedings of the 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)*, pp. 88-96.

[53] Rutkowska, J. 2004. The implementation of passive covert channels in the Linux kernel. Tech. rep., Chaos Communication Congress.

[54] Smith, J. C. 2000. Covert shells. Tech. rep., SANS Institute Information Security Reading Room.

[55] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. RFC 2784: Generic Routing Encapsulation (GRE), March 2000.

[56] O. Pearson. DNS Tunnel - Through Bastion Hosts, 1998. http://gray-world.net/papers/dnstunnel.txt.

[57] C. H. Rowland. Covert Channels in the TCP/IP Protocol Suite. *First Monday, Peer Reviewed Journal on the Internet*, July 1997.

[58] M. A. Padlipsky, D. W. Snow, P. A. Karger. Limitations of End-to-End Encryption in Secure Computer Networks. Technical Report ESD-TR-78-158, Mitre Corporation, August 1978. http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=A059221&Location=U2&doc=GetTRDoc.pdf.

[59] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan. Infranet: "Circumventing Web Censorship and Surveillance," in: *Proceedings of 11th USENIX Security Symposium*, August 2002.

[60] S. Castro. Covert Channel and Tunneling over the HTTP protocol Detection: GW Implementation Theoretical Design. Technical report, Gray World, November 2003. http://www.infosecwriters.com/hhworld/cctde.html.

[61] I. S. Moskowitz, M. H. Kang, "Covert Channels - Here to Stay?," in: *Proceedings of 9th Annual Conference on Computer Assurance*, pp. 235–244, 1994.

[62] J. Rutkowska. The Implementation of Passive Covert Channels in the Linux Kernel. In *Proceedings of Chaos Communication Congress*, December 2004.

[63] K. Ahsan, D. Kundur. "Practical Data Hiding in TCP/IP," in: *Proceedings of ACM Workshop on Multimedia Security*, December 2002.

[64] E. Tumonian, M. Anikeev. "Network Based Detection of Passive Covert Channels in TCP/IP," in: *LCN* (2005), pp. 802–809.

[65] N. Schear, C. Kintana, Q. Zhang, A. Vahdat. "Glavlit: Preventing Exfiltration at Wire Speed," in: *Proceedings of Fifth Workshop on Hot Topics in Networks (HotNets)*, November 2006.

[66] T. Sohn, J. Moon, S. Lee, D. H. Lee, J. Lim. "Covert Channel Detection in the ICMP Payload Using Support Vector Machine," in: *Proceedings of 18th International Symposium on Computer and Information Sciences (ISCIS)*, pages 828-835, November 2003.

[67] T. Sohn, J. Seo, J. Moon. "A Study on the Covert Channel Detection of TCP/IP Header Using Support Vector Machine," in: *Proceedings of 5th International*

Conference on Information and Communications Security, pages 313-324, October 2003.

[68] T. Sohn, T. Noh, J. Moon. "Support Vector Machine Based ICMP Covert Channel Attack Detection," in: *Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Networks*, pages 461-464, September 2003.

[69] E. Cauich, R. Gómez Cárdenas, R. Watanabe. " Data Hiding in Identification and Offset IP Fields," in: *Proceedings of 5th International School and Symposium of Advanced Distributed Systems (ISSADS)*, pages 118-125, January 2005.

[70] Murdoch, S. J., and Lewis, S. 2005. "Embedding covert channels into TCP/IP," in: *Proceedings of the Workshop on Information Hiding (IH'05)*, 3727, 247–261.

[71] D. Pack, W. Streilein, S. E. Webster, R. K. Cunningham. "Detecting HTTP Tunneling Activities," in: *Proceedings of Third Annual Information Assurance Workshop*, June 2002.

[72] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "Blinc multilevel traffic classification in the dark," *SIGCOMM*, 2005.

[73] W. John and S. Tafvelin, "Heuristics to classify internet backbone traffic based on connection patterns," *ICOIN*, 2008.

[74] M. Iliofotou, H. Kim, M. Faloutsos, M. Mitzenmacher, P. Pappu, and G. Varghese, "Graph-based p2p traffic classifcation at the internet backbone," *IEEE Global Internet Symposium*, 2009.

[75] K. Borders and A. Prakash. "Web Tap: Detecting Covert Web Traffic," in: *Proc. 11th ACM Conference on Computer and Communications Security, pages 110–120, Oct. 2004.*

[76] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," SIGCOMM Comput. Commun.Rev., vol. 37, no. 1, pp. 5–16, 2007.

[77] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," *SIGCOMM*, 2006.

[78] E. Hjelmvik, W. John, "Statistical Protocol Identification with SPID: Preliminary Results," in: Proc. of the 6th Swedish National Computer Networking Workshop. May 2009.

[79] S. Cabuk, C. E. Brodley, and C. Shields, "IP Covert Channel Detection," in: *ACM Transactions on Information and System Security*, vol. 12, no.4, article 22, Apr. 2009.

[80] S. Cabuk, C. E. Brodley, C. Shields. "IP Covert Timing Channels: Design and Detection," in: *Proceedings of 11th ACM conference on Computer and Communications Security (CCS)*, pages 178-187, October 25-29 2004.

[81] S. Cabuk, C. Brodley, C. Shields. "IP Covert Timing Channels: An Initial Exploration," in: *Proceedings of ACM Computer and Communications Security Conference*, October 2004.

[82] R. Jagannathan, T. Lunt, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H. Javitz, P. Neumann, A .Tamaru, and A. Valdes. System design document: Next-generation intrusion detection expert system (NIDES). Technical Report

A007/A008/A009/A011/A012/A014, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, March 1993.

[83] S. Kumar and E. Spafford, "A pattern matching model for misuse intrusion detection," in *Proceedings of the 17th National Computer Security Conference,* pages 11-21, October 1994.

[84] P. Uppuluri and R. Sekar, "Experiences with specification-based intrusion detection," in: *Proceedings of RAID'2001*, LNCS 2212, pages 172–189, Oct. 2001.

[85] J. Zimmermann, L. Me, and C. Bidan, "An Improved Reference Flow Control Model for Policy-Based Intrusion Detection," in: *Proceedings of ESORICS 2003*, Oct. 2003.

[86] E. Tombini , H. Debar , L. Me , M. Ducasse, "A Serial Combination of Anomaly and Misuse IDSes Applied to HTTP Traffic," in *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pp.428-437, Dec. 2004

[87] A. Householder, K. Houle, C. Dougherty, Computer attack trends challenge internet security, Computer 35 (4) (2002) 5–7.

[88] P. A. Porras and P.G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances" in: *Proceedings of 20th National Information Systems Security Conference*, 1997.

[89] H. Debar, "An Introduction to Intrusion-Detection Systems,", IBM Research, Zurich Research Laboratory, Rushlikon, Switzerland, Aug. 2000.

[90] A. B. Jeng, M. D. Abrams. "On Network Covert Channel Analysis," in: *Proceedings of Third Aerospace Computer Security Conference*, December 1987.

[91] Anonymous, "Bypassing a Restrictive Internet Proxy," in: *2600 The hacker quarterly,* pages 29-30, Spring 2008.

[92] C.C. Albrecht, "How Clean is the Future of SOAP? " in *Communications of the ACM, vol. 47 no. 2, pp. 66–68, Feb. 2004.*

[93] K. Borders and A. Prakash. "Web Tap: Detecting Covert Web Traffic" in *Proc. 11th ACM Conference on Computer and Communications Security, pages 110–120, Oct. 2004.*

[94] L. Brinkhoff.(2008, June) HttpTunnel, [Online], Available: http://www.nocrew.org/software/httptunnel.html

[95] P. Singh, Whispers on the Wire - Network Based Covert Channels, Proceedings of the Symposium on Security for Asia Network (SyScAN'05), 1st and 2nd of September 2005, Bangkok, Thailand

[96] G. Fisk, M. Fisk, C. Papadopoulos, J. Neil, "Eliminating steganography in internet traffic with active wardens," in Proc. of the 2002 International Workshop on Information Hiding. Oct. 2002.

[97] S. Ubik, P. Zejdl, "Passive monitoring of 10 gb/s lines with pc hardware," in: *TNC '08: Terena NetworkingConference*, Bruges, BE, 2008.

[98] M. Dusi, W. John, and K. Claffy, "Observing routing asymmetry in internet traffic," *http://www.caida.org/research/traffic-analysis/asymmetry/*, 2009 (accessed 2009-08).

[99] W. John and S. Tafvelin, "Experiences from passive internet traffic measurements," *Technical Report 2008-17, Chalmers University of Technology,* 2008.