to filter traffic are not much different than those of stateful packet filters, despite requiring significantly more resources. Circuit gateways may require user names, passwords, and other information from clients before creating connections to the outside, potentially allowing them to filter based on local user and application, but they still have no knowledge of remote users, nor of application-layer protocols.

Circuit gateways are generally used as network firewalls; host-based circuit gateways would be pointless. A common circuit gateway system is SOCKS [KK92, LGL+96].

Application gateways

Application gateways [CBR03, p. 185] work as proxies at the application layer. As with circuit gateways, connections are not end-to-end but are made by the gateways on request. In addition to the information available to circuit gateways, application gateways are able to filter based on application-layer information, and possibly also user and application information, but, due to the diversity and complexity of application-layer protocols, separate gateways are required for each application.

Typical uses of application gateways are to filter for malware, malformed messages that could represent attacks against servers, and content that should not be entering or leaving the protected network, such as pornography or trade secrets. The tradeoff for this power is that application gateways require large amounts of memory and processing time and frequently cannot be made fully transparent to users. Most common web proxies and SMTP servers are able to function as application gateways for their respective protocols; gateways for more obscure or proprietary protocols can be difficult to find.

Distributed firewalls

Traditionally, network firewalls have been the most common method for protecting large numbers of hosts; they are easy to deploy and manage. However, they do have a number of disadvantages:

• Unauthorized or improperly protected network links can allow users to bypass firewalls entirely.

• Laptops and other roaming hosts move in and out of protected areas; while outside of the network perimeter, they are not protected by its firewalls.

• Firewalls at network choke-points are single points of failure and possible performance bottlenecks.

• In networks with more than one route to the Internet, it is possible for packets to leave over one link and their responses to arrive on another. Stateful packet filters and gateways will not be able to re-assemble connections under these conditions.

As a way of improving on these weaknesses, Bellovin [Bel99] proposed using distributed firewalls in which each host in a protected network runs its own firewalling software but receives configuration from a central management server. Such a system has a number of advantages:

• The firewall is independent of network topology; hosts no longer need to be classified as “inside” or “outside” of a security perimeter.

• There is no longer a single point of failure or performance bottleneck; if one host goes down, others are unaffected.

• The firewall can base its decisions on additional information that is not available to most network firewalls, such as user and application names and dynamically assigned ports, without needing to resort to computationally expensive application layer processing.

• Hosts that move between networks or otherwise change addresses frequently, such as laptop computers, are easily protected, regardless of their current locations.

The independence of network topology provided by this model is particularly important: many of the threats faced by modern networks are from the inside (spyware, worms carried into protected areas on laptops, malicious insiders, etc.). The traditional model of walled-in networks with barbarians outside that forms the basis of most firewall architectures provides little to no defense against internal attackers.

One disadvantage of distributed firewalls is that hosts cannot make any assumptions about spoofed IP addresses on the local network and don’t know which hosts to trust. Bellovin [Bel99] suggests working around this by using IPsec to cryptographically verify host identities. In any case, this is a stronger form of authentication than relying on IP addresses, and works even if IP addresses change. However, IPsec adoption is nowhere near the point of making this practical over domains larger than corporate networks, so hosts will still have to

rely on address-based authentication for communication with hosts outside of their administrative domains. Also, many older and less-capable systems are still in use for which IPsec support is not available.

Other disadvantages of distributed firewalls lie in the difficulty of managing configuration. Management servers may either attempt to push configuration updates to all hosts, in which case any hosts that are currently unreachable do not receive the updates, or rely on hosts polling for and pulling updates, in which case hosts that neglect to poll regularly (or are prevented from doing so) do not receive them. It is difficult to ensure that all hosts that should be running firewalling software actually are; hosts whose firewalls are malfunctioning or not present may not be protected at all. Finally, some malware (such as Y3K Rat 1.6 [Whi01]) attempts to disable local security software on hosts that it infects; if this succeeds, then the firewall is disabled at the moment when it is needed most. In one implementation [IKBS00] of Bellovin's concept [Bel99], simply disabling the policy daemon will disable the firewall. Since there are many fewer avenues for malware to be executed on network firewalls, this attack is much less of a threat for traditional firewalling systems.

Hybrid firewalls

Bellovin [Bel99] recognized some of the weaknesses of distributed firewalls in his original paper and suggested that hybrid firewalls, combinations of distributed and choke-point elements, could address some of them. There are a number of ways that such a system could be structured:

• A network of non-mobile hosts, each protected by distributed firewalling software but without cryptographic authentication of peers, could use simple network firewalls to drop inbound packets with spoofed internal addresses. This would prevent many spoofing attacks under the assumption that no internal hosts are engaging in spoofing.

• A network that uses distributed firewalling could employ network firewalls as a secondary protection mechanism: this would protect hosts whose firewalling software is missing, malfunctioning, or not configured correctly. If the network firewall failed, it could fail open without leaving the network completely exposed.

• Networks that contain both mobile and stationary hosts could use network firewalls with knowledge of network topology to protect stationary hosts, while using a distributed firewall and an IPsec gateway to allow protection for and secure communication with mobile hosts.

2.11 Intrusion Detection Systems

Unlike firewalls, intrusion detection systems (IDSs) take a reactive approach, attempting to identify attacks in progress or to detect evidence of past attacks and taking appropriate countermeasures. IDSs can use either statistical methods or pattern-matching to detect intrusions, can operate by monitoring logs or events on individual hosts or by monitoring network traffic, and can run periodically or in real time [DDW99].

Responses made by IDSs depend on the type of IDS and the type of attack. Single host signature matchers, such as some anti-virus engines, that detect successful intrusions may attempt automated clean-up operations or may simply notify an operator that clean-up is needed. Worm traffic detected on a local network may indicate that an intrusion has taken place; an IDS detecting such an event may attempt to isolate the infected host(s) in order to contain the worm [MSVS03]. Port scans detected at a perimeter firewall may result in all traffic from the scanning hosts being blocked [Sol98] or subjected to rate limitations [Wil02]. Unfortunately, automated responses can often cause unintended side effects: false positive detections may cause more harm over the long term than undetected intrusions and, even when attackers cannot hide from IDSs, they may attempt to trigger inappropriate responses. Signature-based detectors can be fooled into making inappropriate responses by deliberately matching the signature of different attacks, port scanners can spoof large numbers of source addresses in order to make port scan detectors block off large segments of the Internet, and worms can use spoofed source addresses to trick IDSs into quarantining too many hosts [PN98]. In many cases, the only safe automated response to intrusions is to inform the operator.

Intrusion Prevention Systems (IPSs) use statistical or pattern-matching methods to attempt to automatically detect and disrupt attacks in real time [KVV05]. Due to their similarities in function, some IDSs (including Snort [MJ]) are

also capable of functioning as IPSs. Many "deep" packet filters are essentially combinations of stateful packet filters and simple IPSs [Ran05].

2.12 Network Address Translators

Network Address Translators (NATs) [SE01, SH99] are devices that re-write the source IP addresses of traffic leaving a network and the destination addresses of traffic entering it. This process is known as network address translation.

The most common reason using for network address translation is to share a single IP address among many devices. Devices on a network are assigned IP addresses that are only valid inside the network (usually chosen from the RFC 1918 private address blocks [RMK+96]); they can communicate amongst each other using these addresses, but the addresses are not recognized by the Internet at large. In order to connect such a network to the Internet, all traffic entering and leaving the network is routed through a NAT, which re-writes the source addresses of all outbound packets to its own external IP address (as in Figure 2.3(a)), which is valid on the Internet. In order for responses to outbound packets to be correctly received, the NAT keeps state information allowing it to re-write the destination addresses of packets it receives to the appropriate internal addresses (as in Figure 2.3(b)). In such a system, the internal addresses are known as private addresses and the externally-valid address of the NAT is known as the public address. Since TCP and UDP connections are typically identified by the combination of source and destination addre
26
and source and destination ports, such a N
cannot handle two internal hosts making connections to the same external host and port from the same source ports; to work around this, a variation called network address port translation, which also re-writes source port numbers on outbound packets, is used. Other variations on this system allow NATs to use more than one public address; some NATs support one-to-one mappings between private and public addresses.

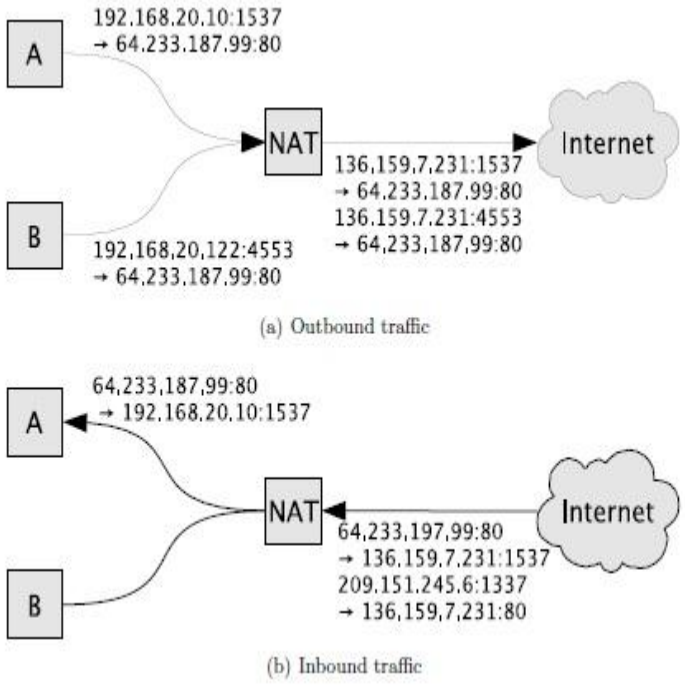

Figure 2.12: Traffic passing through a NAT.

As a side-effect of network address translation used in this manner, outside hosts cannot normally initiate connections through a NAT: without state information to indicate which inside host should receive a packet, the NAT will simply drop it (as with the 2nd packet arriving at the NAT in Figure 2.3(b)). Therefore, though NATs are not designed as network security devices, they can function as simple packet filters. Since networks employing NATs require that all outbound traffic pass through them, they are ideal locations to place network firewalls; most available NATs have at least limited packet-filtering capabilities, and many existing firewalls can also function as NATs. The state-tracking mechanisms used by NATs are similar to those used by stateful packet filters and have the same weaknesses with regards to stateless transport protocols.
27

Network address translation can also be used in reverse in order to share a single IP address among many network servers or to load-share between them: this technique is often called port forwarding. The term destination network address translation is also used; in this case, re-writing source addresses of outbound packets is known as source network address translation.

NATs resemble circuit gateways in that outside hosts see only the NATs' addresses, rather than those of the protected hosts, but differ in that protected clients and servers communicate

directly with outside hosts. NATs only re-write addresses; packet boundaries and other network-layer semantics are preserved.

2.13 VPNs and Encrypted Channels

Many attacks against network services are only possible because of IPv4's lack of protection against sniffing and modification and its lack of verification that packet source addresses are correct. All of these problems can be solved by proper application of cryptography: encryption of payload data makes sniffing useless; data integrity checks can prevent insertion and modification attacks, and user or host authentication allows spoofed source addresses to be detected.

Virtual Private Networks (VPNs) are private networks that allow confidential communication over insecure public networks. VPNs are normally implemented by tunneling ordinary, insecure protocols inside encrypted channels on top of standard network and transport protocols. This can be done at different protocol layers. TLS [DR06] is normally used for encrypting application protocols on top of TCP, but can also be used to tunnel IP on top of TCP or UDP [Yon06]. IPsec [KS05] and other protocols tunnel IP packets inside a secure channel on top of IP. All of these are complicated cryptographic protocols that provide host authentication, key exchange, and confidential, integrity-protected channels.

The benefits of VPNs and encrypted channels come at a cost. Encryption is CPU intensive, adding a cost to communication that may be prohibitive for low-powered or busy hosts. VPN software often requires operating system support, which may not be available for some platforms. Tunneling of any sort makes it impossible to block access to applications by port number; a growing trend on the Internet is the tunneling of various protocols on top of HTTP [Alb04, BP04]. Furthermore, encryption makes firewalling difficult: unless a firewall knows all encryption keys in use, it cannot decrypt tunneled traffic to make filtering decisions. This leaves security up to destination hosts themselves, just as it would be without firewalls; some firewalls may block encrypted traffic for this reason. Thus, it is beneficial to use a technology that provides the minimum required security at the minimum cost, rather than using more powerful VPN systems, wherever possible.

Stealthy Authentication Mechanisms

A common goal in firewall policy design is to limit which remote users can connect to particular services. There are many reasons for implementing such access restrictions, including

• strengthening a defense in depth: adding an extra layer that attackers must break through before reaching anything important,

• protecting systems with known un patched vulnerabilities from attackers until patches are available, while still allowing access to certain authorized users, and

• adding a measure of user authentication to legacy or proprietary systems with inadequate integrated security measures.

A side benefit of limiting access to services at a firewall in this manner is that unauthorized users will have difficulty even learning of the existence of the protected service; port scans against the service from unauthorized hosts cannot tell the difference between a port that appears closed because nothing is using it and a port that appears closed because a firewall is intervening. This adds a degree of security to the service: attackers are unlikely to attack services that they don't know about, so measures that make services more difficult to detect will reduce the number of attackers aware of the service and therefore the number of attacks made. Since the number of attackers is finite, a reduced number of attackers and attacks reduces the probability of a security breach.

Regrettably, most existing firewalls aren't very good at implementing such restrictions. One common method is to assume that trusted users connect only from certain small sets of trusted computers with known IP addresses, and allow connections only from these addresses. This has many limitations: attackers can spoof trusted IP addresses or hijack trusted hosts and trusted users may attempt to connect from hosts not in the trusted set. Since many computers on today's Internet do not have static IP addresses, instead relying on DHCP servers to assign addresses that change over time, allowing access from one particular computer may require granting access to many thousands of IP addresses; in the case of a worm outbreak, there is a strong chance that an attacking worm will reside on a computer with one of these addresses. Adjusting the set of trusted IP addresses usually requires manual re-configuration by a firewall administrator. The

other method available is to use a world-accessible service that uses some form of user authentication to identify trusted users and grant them temporary access to the protected service, by creating a temporary association between the trusted user and its IP address. Unfortunately, this approach typically has drawbacks as well. Exploitable flaws in security and authentication software are discovered regularly [UC06a, UC06b, UC07a, UC07b]. Since the authentication service is visible to the world, it can be attacked by anyone, and since it controls firewall rules, successful attacks could be used to completely bypass the firewall.

Clearly, since IPv4 headers include no information about users, there is little that can be done with the approach of filtering by fixed sets of source IP addresses. The model of authenticating to a world-accessible service and requesting access has potential, but needs enhancements to correct the weaknesses described above. In particular:

• Since the whole point of the firewall access restrictions is to keep unauthorized users from connecting at all, the authentication service should be no easier for attackers to communicate with than the protected services. It must still be world accessible, but it could be hidden in some manner. One way to accomplish this is for communication with it to use a covert channel.

• Since any attacker who discovers and is able to communicate with a hidden authentication service has already displayed significant skill and resources, the authentication service must be cryptographically secure. Also, it should be as simple as possible, so that it can easily be audited and reasonable assertions made about its vulnerability to attack.

Using a complex, highly visible mechanism would present a risk of attack not significantly different than that of the original, now protected, service; using a hidden service that provides strong authentication and is simple enough to easily audit provides a target that is both less likely to be compromised if attacked and less likely to be attacked at all.

This chapter will begin by discussing covert channels over networks and then describe several ways that covert channels could be used by firewall authentication services. Two existing techniques, port knocking and single packet authorization, will be discussed in detail, including their design issues, strengths, weaknesses, and current implementations.

3.1 Single Packet Authorization

Single Packet Authorization, or SPA, has the same goals as port knocking, but, instead of encoding authentication information in a series of port numbers, it encodes it in the payload of a single UDP datagram (see Figure 3.2). This allows for authentication messages of several kilobytes to be used without concern for packet reordering.

The information encoded in an SPA message is generally similar to what might be encoded in a port knocking sequence (see Section 3.2.1). A message containing a plain-text secret could be used, although most existing implementations use some sort of encrypted or one-time message. Unfortunately, the phenomenon of misused cryptography and broken authentication protocols is not limited to port knocking. Cryptknock [Wal04] (actually an SPA system, despite its name) uses an unauthenticated Diffie-Hellman exchange to generate a session key which is then used to encrypt a shared secret; if the server accepts the secret, then it allows unrestricted access to the originating host. Since this protocol doesn't associate the shared secret with the client's IP address, an attacker could break it by re-writing the client's IP headers to make it appear to the server that they originated at the attacker, then forwarding the server's responses back to the client. Alternately, the standard man-in-the-middle attack against Diffie-Hellman [Sta03] would allow an attacker to recover the shared secret. Tailgate TCP (TGTCP) [BHI+02], Doorman [War05], tumbler [GC04], and fwknop [Ras06] implement SPA with more robust authentication schemes.

SPA servers that use packet sniffers or related technologies may need to limit packet sizes to the path MTU between client and server (typically a minimum of 576 bytes [MD90]) in order to avoid fragmented packets. However, this is still ample room for authentication information in messages of this size.

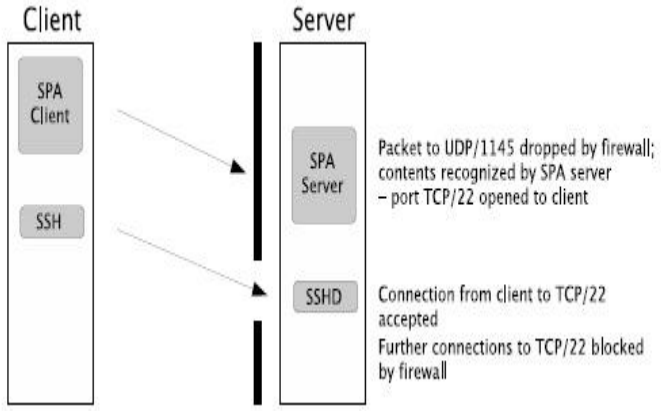

Figure 3.1: SPA example. A port is opened in the firewall in response to an authentication packet.

Advantages of SPA

As Rash [Ras06] points out, SPA is easier to implement and less failure-prone than port knocking and is probably preferable in most circumstances.

1. An SPA server can be written as a normal network service on an open port. Since UDP services are not required to respond to messages that they receive, and the protocol does not automatically generate any response, a non-responding UDP service on an open port on a system that silently drops unexpected packets is indistinguishable from a closed port to a port scan. An SPA server can therefore be written as a normal network service, without needing to resort to packet sniffers or any platform-specific mechanisms. (However, such a design may put constraints on the available mechanisms to manipulate firewall states.)
2. Since only one packet must be sent before opening a connection, an SPA authentication exchange takes much less time and is less vulnerable to packet loss than port knocking, besides being immune to packet reordering.
3. NATs and stateful firewalls between SPA clients and servers will only have to allocate resources for at most one logical connection, rather than one for each knock as required with port knocking.
4. Compared to port knocking, SPA can use relatively large authentication messages without sacrificing performance and reliability.

Disadvantages of SPA

Despite being much less sensitive to packet ordering than port knocking, SPA systems will still fail if a connection attempt reaches the firewall before the authentication packet has been received and processed. They will also fail if an authentication packet, or a fragment of one, is dropped or corrupted in transit.

SPA servers typically cannot be implemented as log readers, since SPA systems need to access packet payloads and firewalls generally don't log any more than packet headers. However, this is not usually a problem, since log reading is an inefficient design compared to its alternatives (see Section 3.2.2) and is seldom used for SPA.

Egress filters may not pass outbound traffic destined to unusual UDP ports, but SPA servers could run on ports regularly used for "normal" traffic. For instance, most egress filters will permit DNS traffic; SPA messages bound to a server running on port 53/UDP would likely pass unmolested.

3.2 Variations on SPA

It is possible to encode SPA messages into the payloads of any protocol. In [Ras06], Rash suggests using the payloads of ICMP or GRE messages. In theory, raw IP messages with no transport headers at all could also be used. Although such systems have the potential to be extraordinarily stealthy (ICMP echo-request, ("ping") messages are very common in the Internet background radiation [PYB+04], although GRE and raw IP messages are rather unusual), they do present some implementation challenges.

User applications cannot usually read ICMP, GRE, or raw IP payloads, requiring that servers using such encodings hook into network stacks at a lower level (for example, using packet sniffers). More importantly, unprivileged user programs cannot directly send such messages, thus requiring clients to be privileged applications. Currently, fwknop [Ras06] and Cerberus [Epp04] implement SPA over ICMP.

Barham et al. [BHI+02] suggested a variation on TGTCP in which the authentication information would be attached as a payload to the SYN segment opening a TCP connection; this would prevent race attacks and be invulnerable to out-of-order delivery. However, this approach is not without weaknesses: it only works for TCP ports, it requires modifications to the client's network stack to attach authentication information to outgoing TCP SYN segments, it requires a kernel-level server that can process packets before they reach the transport layer of the

firewall's network stack, and some egress filters may drop SYN segments carrying data.

3.3 Active-covert SPA

Since SPA traffic is visible to packet sniffers and is not disguised as background noise, it cannot normally be considered active-covert. However, it is possible to give active-covert properties to SPA by encoding the payload as something that might normally be seen in background traffic and setting the destination port to match. For instance, authentication information encoded as ASCII text and sent to port 1026/UDP with the proper headers would resemble Windows Messenger spam [LUR03], and a message resembling Intel x86 machine code sent to port 1434/UDP might be mistaken for the Sapphire worm [MPS+03].

3.4 Concerns about "Security by Obscurity"

Port knocking systems in particular have often been accused [BBO05, MMETC05, Nar04] of being nothing more than "security by obscurity". Generally, these claims are based on assumptions that the security of port knocking authentication systems is based solely on them remaining hidden, or that concealing security-sensitive information is bad and that all details of security systems should be visible.

Beale [Bea00] describes a system relying on security by obscurity as one that relies on critical knowledge about the system's design being kept secret, though the secret information could be discovered by an outsider without unreasonable effort. The authentication systems used by traditional services such as telnet and FTP, which send passwords in plain text, are generally considered insecure by modern standards [Bel89], but their well-specified protocols and documented reliance on the secrecy only of small easily-changeable per-user passwords leave them well outside of the realm of security by obscurity. This is equally true for similarly-designed services such as SSH, which employ cryptography to protect secrets in transit.

The security of port knocking systems and other covert authentication schemes is also dependent only on the knowledge of small, easily changeable secrets; in the case of port knocking systems, these are port sequences. Systems that send their secrets are equivalent in security to telnet, whereas those that use cryptographic protocols are more akin to SSH. In neither case does the security of the authentication system depend on any other property. The covertness of the communication channels being used is not necessary; if the same information was transmitted across normal, open ports, the system would remain secure. Rather, the covertness only serves to increase the level of effort required to attack the systems. As Beale points out, concealing an already-secure service is not a weakness but rather has a number of advantages, reducing the number of attacks faced by the system and forcing attackers to do more work, which both slows them down and makes their actions more obvious.

Improvements to Port Knocking and SPA

As presented in the previous chapter, port knocking and SPA have a number of weaknesses. In this chapter, novel techniques for addressing some of these weaknesses and other improvements to port knocking are introduced. First, I introduce methods for conducting challenge-response authentication using SPA or port knocking, which improve on some of the limitations of the authentication algorithms described in Section 3.1 and enable authentication of the server as well as of the client. Next, a variety of strategies for ensuring that port knock sequences can be correctly decoded, even if delivered out of order, are discussed. Third, two alternate methods for encoding information into port sequences are introduced. Finally, several possible ways of associating authentication exchanges with connections and preventing race attacks are described.

4.1 Basic Unilateral Authentication

The ISO two-pass unilateral authentication [ISO95, MvOV96] is designed to authenticate a client A to a server B; no attempt to authenticate the server to the client is made. A slightly modified version of this algorithm, intended to be suitable for port knocking or SPA, is shown in Algorithm 4.1. In the following discussion, I refer to message 1 as the request, message 2 as the challenge, and message 3 as the response.

Algorithm 4.1: Challenge-response unilateral authentication

1: $A \rightarrow\vert B$: req
2: $B \rightarrow A$: N_B
3: $A \rightarrow B$: $MAC_{K_{req}}(N_B, ID_A, ID_B, req)$

where A is the client
B is the server
req is a request for authentication
N_B is a nonce chosen by B
K_{req} is a secret key shared by A and B
ID_X is the IP address of X
MAC is a cryptographic message authentication function
, (a comma) represents concatenation

A begins the sequence by sending a request, which serves both to initialize the protocol and identify the operation to be performed upon successful authentication. Cryptographically, this must be considered public information. B responds to a recognized request by issuing a unique nonce as a challenge, to which A responds with a MAC covering the nonce, the IP addresses of A and B, and the request sequence, keyed with a symmetric key associated with the request. Upon receipt of the response, B will re-compute the MAC using the request that it received, the nonce that it sent, A's IP address as taken from the packet headers, and B's own IP address. If the MAC is valid, then B will perform the requested operation; otherwise, no action will be taken.

If HMAC-SHA1 is used as the MAC algorithm, then response messages will be 160 bits long. Due to birthday attacks [MvOV96], nonces used in this situation should be at least half the bit length of the MAC, or 80 bits in this case. As argued above, a hard-to-guess sequence of 8 to 10 bytes will serve as a request sequence.

This protocol is suitable for port knocking or SPA systems employing either preconfigured or user-issued commands. Servers can identify pre-configured commands by associating unique request sequences with each command. User-issued commands could be constructed by appending port numbers and other information to request sequences; if the command must be kept secret, it can be encrypted using Kreq . No integrity-checking information is needed in such request messages, because they are covered by the MAC in the response message, and none of the information in the command will be acted upon until after successful authentication. Pre-configured commands are best for port knocking systems, due to the relatively high overhead of sending data, but there is little penalty for attaching additional data to SPA requests.

A's IP address has been added to the MAC in Step 3 of the original algorithm to prevent possible Mafia fraud-based attacks [DGB87] (see Figure 4.1) in which an attacker C initiates the protocol and receives a challenge, intercepts (and blocks) a challenge issued to A in another protocol session, and forwards its own challenge to A in order to get A to generate a valid response for C (see Algorithm 4.3 for an example). By covering both IDs with the MAC, Algorithm 4.1 prevents C from subverting the protocol to authenticate to B as itself, but does not prevent C from subverting the protocol by masquerading as A. Such an attack still requires A to initiate an authentication exchange itself before it will generate a response, and would have the effect of causing B to perform the action specified by C's request. If A's and C's requests are the same, then there is no security breach: B does exactly what A asked, under A's credentials, and nothing more. If the request caused a port to be opened, then C could attempt to connect to it while continuing to masquerade as A, but this is then equivalent to C ignoring the authentication exchange and attempting to hijack a successful authentication, as in Section 3.2.3. If A's and C's requests are different, then authentication will fail, since A also covered its own request in the MAC and B will expect C's request when verifying it.

Instead of adding A's address to the MAC in the response, a MAC covering IDA and NB could have been added to the challenge message; this would have equivalent security properties but increase the amount of data to be transmitted. This is the approach suggested by van Oorschot and Stubblebine [vOS06] for preventing Mafia fraud-based attacks.

The original version of Algorithm 4.1, presented in [dAJ05], did not cover the request in the MAC and therefore depended on the keys associated with the two requests being different in order to resist Mafia frauds.

4.2 Authentication in the Presence of NATs

One flaw in Algorithm 4.1 is that it requires the client, A, to know its identity as seen by the server, B. Unfortunately, if the client is behind a NAT, then it may not know its public address and may not even know that the NAT exists. (Since the server is intending to receive

connections, it is assumed to have have a valid public address.) In the above protocol, A will use its private address PIDA to compute the response, but if A's address is re-written on the packets that it sends, then B will use A's public address IDA to verify it and authentication will fail.

Algorithm 4.2.1: NAT-aware unilateral authentication

1: $A \rightarrow B$: req, PID_A
2: $B \rightarrow A$: $N_B, ID_A, MAC_{K_{req}}(PID_A, ID_A)$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_B, ID_A, ID_B)$

where A is the client (prover)
B is the server (verifier)
req is a request for authentication
PID_X is the private IP address of host X
ID_X is the public IP address of host X
N_B is a nonce chosen by B
K_{req} is a secret key shared by A and B
MAC is a cryptographic message authentication function
, (a comma) represents concatenation

In an earlier work [dAJ05], I presented an algorithm called "NAT-aware unilateral authentication" (Algorithm 4.2) which I claimed would authenticate a client A that doesn't know its public IP address to a server B while resisting Mafia frauds. Unfortunately, I have since discovered an attack against this algorithm, shown in Algorithm.

Algorithm 4.2.2 Attack against NAT-aware unilateral authentication

1: $A \longrightarrow B$: req, PID_A
2: $C \longrightarrow B$: req, PID_A
3: $B \rightarrow /A$: $N_B, ID_A, MAC_{K_{req}}(PID_A, ID_A)$
4: $B \longrightarrow C$: $N'_B, ID_C, MAC_{K_{req}}(PID_A, ID_C)$
5: $C(B) \rightarrow A$: $N'_B, ID_C, MAC_{K_{req}}(PID_A, ID_C)$
6: $A \longrightarrow B$: $MAC_{K_{req}}(N'_B, ID_C, ID_B)$
7: $C \longrightarrow B$: $MAC_{K_{req}}(N'_B, ID_C, ID_B)$

In this attack, an attacker C waits for a legitimate client A to initiate the protocol and then opens its own protocol session by sending a copy of A's request. C then blocks the delivery of B's challenge to A and substitutes its own challenge. (The notation B ! /A means that B sends a message to A, but it is not delivered.) A accepts B's assertion that its public IP address is IDC and generates a response, which B rejects since it knows A's correct public IP address. However, this response is valid for C, which is then able to complete the protocol successfully without knowing Kreq .= This attack also works if C blocks the request and response sent by A or masquerades as B to A.

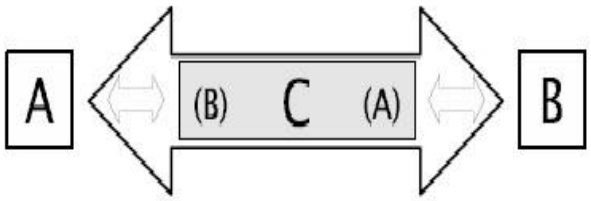

Figure 4.2: The Mafia fraud. A and B think that that they are authenticating to each other, but C is forwarding messages between them with the goal of convincing A that it is B and B that it is A.

Since this attack is exactly what Algorithm 4.1 was designed to prevent, there is no point in ever using Algorithm 4.2. However, the Mafia fraud (a man-in-themiddle attack related to the grandmaster postal-chess problem [MvOV96, BD90] and known as the MiG-in-the-middle attack in military contexts [And01], see Figure 4.1), which works because the verifier, B, has no knowledge of the physical location of the prover, A, is generally considered difficult to prevent and is frequently ignored by authentication systems [AS02]. None of the general solutions to this problem available are appropriate under these circumstances because A cannot practically be isolated during authentication [BBD+91], the only communication channel available can be accurately monitored by attackers [AS02] and doesn't have a constant communication delay [BD90], and A's physical description (here, A's IP address is sufficient – see Section 4.1.1) cannot be included in the exchange [Des88] because A doesn't know its IP address.

The lesson to learn here is that it is difficult to prevent Mafia frauds when the client does not know its identity (in this case, its public IP address). Even if it does, any other computer sharing the same public IP address (i.e., any computer behind the same NAT) can still carry out a Mafia fraud. The Mafia fraud is generally irrelevant in protocols that combine authentication with key agreement, because, even if an attacker succeeds in authenticating as some other entity, it will still not know the agreed-upon key [DvOW92]. Some of the solutions to the race attack problem presented in Section 4.4 are based on key agreement techniques, so they

may mitigate this problem. Another partial solution is for clients to request their public IP addresses from trusted third party identity oracles before starting the authentication protocol.

In addition to this attack, since req is not covered in the MAC in the response, C could substitute A's request with another one employing the same key to cause B to execute a command other than the one that A intended. This is easily corrected by adding req to the MAC, as in Algorithm 4.1.

Improvements to Application Filtering

Port knocking and SPA are primarily designed to communicate information about users to remote ingress packet filters, allowing them to filter on a per-user basis without relying on assumptions about IP addresses. However, neither are ideal for communicating with egress firewalls on local networks. On relatively trusted local networks, stealth is irrelevant: everyone on the inside knows where the firewall is and what it is capable of doing; there's no value in hiding it. Also, whereas filtering by applications is of limited use to ingress firewalls (allowing connections from poorly-implemented clients presents little risk to the server and malware is not likely to be able to authenticate as a valid user), it is of significant value to egress firewalls. Finally, whereas port knocking and SPA are designed to protect services that receive small numbers of connections, egress firewalls must handle much larger numbers; the overhead of port knocking may be a liability in this case.

Application filtering provides an efficient mechanism for host and network firewalls to make decisions based on the users and programs that are sending and receiving network traffic. This chapter discusses existing designs and implementations of application filters and a variety of common problems with such systems. It then presents partial solutions to some of these problems.

5.1 Existing Application Filtering Systems

Since user and application information is not included in IP packet headers, this information is generally not available to network firewalls. Some circuit and application gateways use special protocols to authorize connections that authenticate users (such as SOCKS [LGL+96]), but no existing network firewalls seem to make any checks on the applications making connections.

Host firewalls, on the other hand, do have access to user and application information. Although the built-in firewalling systems on most operating systems don't support application filtering, most third party packages do. Judging by its behaviour, ZoneAlarm [Zon07] (a popular commercial firewalling package for Microsoft Windows) checks hashes of programs and shared libraries that attempt to make connections and performs some tracking of programs that make connections on behalf of others. By default, when it detects a connection attempt from an unknown program which is not otherwise explicitly allowed or denied by the current configuration, it prompts the user for a decision. Unfortunately, information about the algorithms used internally by this and other proprietary packages is not publicly available. Prior to August 2005, the netfilter firewalling system on Linux supported a form of application filtering through the owner match module. This support was removed in kernel 2.6.14 due to conflicts with kernel locking [HM05]; the owner match still supports user and group matching. Originally, this module attempted to match packets to specified program names or process IDs by iterating over the list of currently running processes, searching for one matching a given command name or process ID, and then iterating over all files held by that process to check if any of them matched the socket used by the packet being matched; there was no interactive component. TuxGuardian [dS06] provides application-filtering support to Linux using a different approach: rather than working through net filter, it uses the Linux Security Modules system [WCM+02] to hook into attempts to open sockets or listen for connections and call out to a user space program to ask for permission to allow the attempt. The user-space program checks both the program's name and the MD5 hash of the executable file against its configuration, optionally prompts the local user if the program is not recognized, and allows or denies the request.

5.2 Problems with Application Filtering

Application filtering is a useful and powerful technique, but it cannot be trusted absolutely. It has many flaws, both in concept and in implementation, that restrict what application filters can feasibly do.

5.3 Dealing with Unrecognized Applications

The most obvious problem is what to do when unrecognized applications attempt to make connections. There are three possible responses in this situation:

1. always allow the request,
2. always deny the request, or
3. ask the user if the program should be allowed to use the network.

The first option is dangerous: the firewalling software cannot be expected to have a comprehensive database of all untrustworthy software and will end up letting worms and spyware communicate with impunity. No firewall should accept by default; application filters are no different.

The second requires that the firewall have a comprehensive database of all trusted software. This may be appropriate for professionally-managed networks, where the set of allowed applications is well defined and relatively small, but is problematic in other situations where no such list is available. It is infeasible to require firewall vendors to ship lists of trusted software with their products and impractical to trust them if they do. It is equally infeasible to require end users to build such databases when installing the firewalling software. Applications themselves can obviously not be trusted to assert their own trustworthiness to the firewall.

The third option is also troublesome. People will make mistakes and grant access to programs that they meant to deny. Many users are not sufficiently familiar with their computers to know what applications they should trust and under what circumstances they should trust them. Many will simply click "accept" to any request for a connection, regardless of what is asking [Nor83]. Others will not know the names of all trusted programs, but will recognize that applications should only make connections in response to their requests and will only grant permission shortly after performing some action that could reasonably be expected to make a network connection. However, malware could abuse this trust by monitoring other software and attempting to make network accesses only when something else does. This is compounded by the fact that many network-using programs do not make connections themselves but pass requests to other processes via inter-process communications (IPC), resulting in the application names being presented to the user having little to do with what the user is running. For instance, in some versions of Microsoft Windows, the Windows Update program's connection requests show up to many firewalls as originating in a program called wupdmgr.exe, despite being initiated by Microsoft Internet Explorer (iexplore.exe). Also, many components of Microsoft Windows (such as the DHCP client, network browser, time synchronizer, and messenger) make connections through a program called svchost.exe [Mic06], and some legitimate software (such as time synchronizers) runs automatically, rather than in response to user inputs; users who are unaware of this may deny network access to important software. Finally, malicious programs could explicitly instruct users to allow them access; suitably convincing messages would persuade many users to grant access when they otherwise would not [CBR03].

5.4 Application Spoofing

It is not sufficient for application filters to identify trusted programs solely by executable file names. Anyone could give an arbitrary program the same name as a trusted program; without further checks, an application filter could be completely bypassed in this manner. Some existing malware attempts to use this approach by mimicking operating system components or other trusted software. For instance, the Welchia worm stores a copy of itself with the name svchost.exe under a different path than the legitimate program of this name on Microsoft Windows systems [Sym07d].

A slightly stronger approach is to identify trusted programs by the absolute paths of executable files. This would prevent the sort of file name spoofing used by Welchia but would not catch malware that modifies or overwrites trusted programs, such as some versions of the Spybot (which insert themselves into the command-line FTP client on Microsoft Windows systems) [Sym07c] or Erkez (which attempts to overwrite executable files belonging to Symantec products) [Sym07b] worms. This approach would also be ineffective if the firewall's view of the directory tree could be changed; on Unix-based systems this could be done by mounting a new filesystem on top of an existing one so that a new program occupies the path of the original.

A stronger approach again is to identify trusted programs by cryptographic hashes of executable files. This would detect any attempts to modify or replace trusted programs and would defeat the

above worms. However, even this method can be attacked. Malware need not live in executable programs; it can live in shared libraries. Legitimate shared libraries could be modified to launch malware as a side-effect of a normal library call, or malicious plugins could be written to launch malware instead of their advertised functions. For example, the Fuwudoor back door takes advantage of svchost.exe's ability to run code from arbitrary shared libraries on Microsoft Windows systems to launch itself [Sym07a]. File hash checking may also be vulnerable to race conditions: malware could overwrite a legitimate file, launch, and overwrite itself with the original, legitimate file before attempting network access. An application firewall that checked file hashes would see only the legitimate trusted program, and grant access to the malware. The directory re-mapping trick suggested above could also be used to accomplish this.

Attacks using plugins or shared libraries could be prevented by not only verifying the executable files making network requests, but also all shared libraries currently linked; however, malware could switch shared libraries just as easily as it could executables. Defeating the file-switching attack is more difficult; unless the operating system prevents modifications to currently-loaded executable files and shared libraries, then no checks of these files can be trusted. Malcode inserted into a legitimate running process via a buffer overflow or other exploit also cannot be detected using checks against files.

5.5 Interpreted Languages and Virtualization

Application filtering relies on being able to uniquely identify the application that is requesting a connection. With traditional compiled programs, this is feasible; each instance of such an application is loaded and has its resources managed by the operating system, so it is possible to map packets to programs. Unfortunately, when programs written in interpreted languages load or request resources, the operating system sees the interpreter, not the program itself. To an application firewall, a news reader running in a Java virtual machine is indistinguishable from a backdoor running in a Java virtual machine.

This could be fixed by requiring interpreters to pass information to operating systems about what program they are currently executing, but the diversity of interpreters makes this infeasible. Growing numbers of applications have built-in Turing-complete scripting languages capable of making network connections, and there is no feasible way for operating systems to check if a given program can execute arbitrary code within its execution context. Even if there was, this moves critical firewalling functionality to untrusted user-space programs; there is no way to prevent interpreters from lying to the operating systems about what they are running. Finally, the name of a script is largely meaningless to a firewall; an application filter must make some check against the script's codebase. But interpreted languages may not have code that exists on disk; it might exist solely in a memory buffer or even be retrieved on demand from an external source. In other words, application filtering is mostly useless against interpreted programs.

The same problem applies to programs running in virtualized environments. No program running in a virtualized environment, compiled or interpreted, can be accurately identified by the host operating system unless the virtualized environment is trusted to the same degree as the host OS and can supply information about internal processes to the host. Although the relatively small numbers of virtualization platforms available makes adding such support feasible, many existing virtualization packages (such as VMware [VMw07]) are designed to run operating systems that were not specifically designed for virtualization, and may not even be aware that they are running in virtualized environments. Thus, proper support for application filtering virtualized environments would be at cross-purposes with many existing tools and is not likely to be available any time soon.

Connections by Proxy

Programs that use network resources do not necessarily make network connections themselves. They can instead start different programs or pass requests to existing processes to perform actions on their behalf [CBR03]. An application firewall will see connections originating in the processes that attempt to open them, not the processes that requested them. This flaw can be abused both ways: legitimate programs that pass network connections to others, such as inetd, may be fooled into passing connections to malicious software, and malware

could use legitimate software, such as web browsers, FTP clients or the ping command, to do its dirty work.

Detecting the true originator of network connections requires tracking process parent-child relationships and IPC. Some commercial application firewalls, including ZoneAlarm, appear to use some form of this, but information on the algorithms involved is not publicly available.

5.6 Attacks Against Firewalling Software

Instead of attempting to exploit weaknesses in how application firewalls verify that applications are trusted, malware could attack the application firewalls directly. Malware that attempts to shut down security software is known to exist in the wild [Sym07b]; user-space application filters may have no effective defense against this sort of attack. Malware running at sufficiently high privilege levels may be able to bypass firewalls and send or receive packets without firewall checks. Malware may even be able to interfere with firewalls' perceptions of file contents or paths by intercepting system calls; rootkit software frequently uses this technique to hide itself from IDSs. This is not a likely attack against host firewalls — any malware that has compromised a host to the point that it can do this is more likely to simply disable, cripple, or circumvent the firewall — but could be used against application filters on network firewalls. For these reasons, host firewalls should never be trusted to the same degree as network firewalls, and even network firewalls should not fully trust packet metadata that they cannot independently verify.

5.7 An Improved Architecture for Application Filtering

As pointed out above, application filtering has many flaws. In this section, I present an architecture for application filtering that addresses many of them. It is possible that some or all of these techniques are used by existing proprietary software, but to the best of my knowledge, none of these have appeared in published literature.

Application Filtering by Network Firewalls

Since user and application information is normally not available to network firewalls, application filtering is normally only done by host firewalls. However, there is no reason why it can't be done at network firewalls. Protocols like SOCKS, TAP [Ber92] and Ident [Joh93] can be used to pass information about users to network firewalls; these could be extended to supply application information as well. There are disadvantages to such a system, the most notable being the overhead required for passing user and application information and the possibility of forgery, but there are also a number of advantages. Since there are few avenues to execute arbitrary code on network firewalls (as compared to hosts employing host firewalls), it is more difficult for malware to attack and circumvent network firewalls. Also, network firewalls allow centralized policy management without needing to distribute policy to hosts, thus avoiding the problems that policy distribution entails (see Section 2.4.1).

Two basic methods can be used for application filtering; either can be extended to enable application filtering by network firewalls. One way is to look for packets that open connections and then look up the users and applications that sent or will receive them. This is the approach taken by the pre-2.6.14 Linux owner match, and is necessarily inefficient since the application must be identified by matching the properties of the packet to a socket in kernel data structures, an operation which requires at least O(n) operations (where n is the number of currently-executing processes) an requires locking. However, if this is done inside a stateful packet filter, which already has the ability to detect new connections, then it is trivial to combine application and packet filtering. The other way is to hook into appropriate system calls (such as connect() and accept()) to intercept applications' attempts to open connections. Since this must be done on the hosts running the applications and occurs within the applications' execution contexts, it is trivial to identify the applications involved using this method. However, packet filtering is not normally done inside system calls, so it may be difficult to perform both packet and application filtering using this method.

Either design may be extended to support application firewalling at network firewalls. Again, two basic approaches are possible: hosts wishing to open connections can pre-approve them with firewalls, or firewalls can detect attempts to open connections and ask hosts for application information. Network firewalls can

only detect new connections by packet analysis, so the obvious place to implement application filters is inside stateful packet filters or circuit gateways. Hosts using the pre-approval strategy could detect new connections using either method (although system call hooking is more efficient) and send information about the user and application to the firewall before sending the connection-opening packet. Inbound connections can be handled in the same way: a host that detects an application attempting to listen for connections can inform the firewall of this and let the firewall decide what to do when an attempt to connect arrives. Pre-approval requires a minimum of only one extra message (for outbound connections, pre-approval messages could be attacked to connection establishment packets, but this may interfere with some existing protocols), sent from the host to the firewall, making it relatively fast, but does have a few drawbacks. Since pre-approval messages can be generated at any time, malicious hosts could attempt to flood firewalls with pre-approval messages for connections that they have no intention of establishing. Pre-approval messages must expire after some period of time in order to prevent firewalls from accumulating too many valid pre-approval messages for programs that have exited or will otherwise never follow through. Short lifetimes will require hosts listening for connections to periodically send new pre-approval messages as long as they continue to listen; long lifetimes may make forgery easier. When using the call-back strategy, hosts could either look up processes via socket addresses on demand or build a list of processes that are attempting to open connections via system call hooking, then look up requesting processes there. Either method is slower than the pre-approval strategy, especially since this approach requires a minimum of two messages: a request from a firewall to the host and a response. However, approval lifetimes and DoS attacks against the firewall are no longer an issue.

In order to make forgery of pre-approval messages more difficult, pre-approval messages for outgoing packets should include a hash of the packet to be sent. Since the contents of connection-establishing packets may be predictable, it may be preferable to use a MAC that covers both the packet and a nonce or timestamp instead. The contents of packets won't be known when pre-approval messages are created for hosts that want to receive connections, but MACs covering the hosts' addresses could be used in this case. Similarly, information requests for outgoing messages under the call-back strategy should contain a hash or MAC covering the packet being sent; the originating host can then verify that it actually sent that packet.

In order for any form of application filtering at network firewalls to work, the operating systems (although not the user-space processes) of all protected hosts must be trusted; malicious operating systems (such as those infected with rootkits) could lie to the firewall about what processes are running or any other requested information. However, if application filtering is combined with some other form of firewalling on a network firewall, then traffic from hosts with malicious operating systems is still subject to some form of filtering, whereas such traffic can bypass any form of host firewall.

Preventing Application Spoofing

Since attempts to verify that connections come from legitimate applications by checking files on demand suffer from race conditions and are prone to abuse, an alternate method must be found. One possibility is to verify programs and libraries when loaded, as done by integrity shells [Coh89], and cache the hashes of each file until needed. This scheme will generally require that all programs and libraries, not only those involved in making network connections, be verified; if this overhead is too high, then programs expected to make network connections could be flagged for load-time verification, and connection attempts from all others could be summarily refused. Applications will take longer to load when using this method, but there will be less overhead when opening connections at run-time. If the operating system can load and verify files atomically, this should not suffer from race conditions. Even if a race condition still exists, it should only be exploitable in a much smaller time window (no more than a few milliseconds) than the race condition with on-demand checking.

However, this approach is not without flaws. As with on-demand file checks, it is ineffective against interpreted programs or those running in virtual machines. Hostile operating systems could substitute hash values for legitimate programs to disguise malware. If a collision can

be found in the hash function used, then a malicious program could be substituted for a legitimate one even on a computer with a trusted operating system. A network firewall could foil this last attack by requesting a hash or MAC covering both the program and a nonce, but this requires that hashing be done on demand and that the firewall store copies of all relevant files. An alternative possibility is to eschew files altogether and hash processes' memory images instead. Obviously, writable memory pages must be excluded from such a hash. Unfortunately, this will not work on systems that perform any sort of relocation at load time (including most modern operating systems), since memory addresses inside the code in processes' memory images may differ between executions [Lev00]. Even if all memory addresses were constant between executions, a hostile operating system could still substitute the memory image of a legitimate program in the hashing algorithm or could allow another process to launch a page-replication attack along the lines of Wurster et al.'s [WvOS05]. Finally, this method can't effectively verify programs that execute code from writable pages in memory (although operating systems can prevent this by preventing code execution from writable memory pages, as in PaX [the05]).

Neither method presented here will detect processes that are executing code injected by malware (through buffer overflows or other exploits) rather than their own original code. Other countermeasures (such as non-executable writable memory and address space randomization [dR07, the05, BDS03]) must be taken against these threats.

5.8 Detecting Connections by Proxy

Detecting libraries linked to a process, malicious or otherwise, can be done by simply monitoring what it loads or analyzing its memory space (assuming that processes cannot execute code from writable memory). However, detecting actions taken on behalf of other processes is more difficult. In this section, I present an algorithm that will detect many situations where a process could be opening connections on behalf of others.

A process will be considered to be opening a connection on behalf of another if there is a possible causal relationship between an action of another process and the connection establishment. A causal relationship exists whenever a process receives information from another process before opening a connection. For example, a process could read from shared memory or a socket, pipe or other IPC mechanism to which another process had written. Alternately, a process could have been started, directly or indirectly, by another process. Files could also be used for IPC, but, since files are persistent, tracking reads and writes to files over long periods of time may not be practical; for this reason, a configurable limit of n seconds is placed on how long records of file writes are kept. Using these rules, a directed graph of process interactions can be constructed as follows:

• When a process starts another, a link is added from the new process to the old one.

• When a process writes to shared memory or to a pipe, socket, or other object that is shared between processes, its identity is added to a list associated with that object.

• When a process receives a signal from another process or reads from shared memory, a pipe, socket, or other object that is shared between processes, a link is added from the reader to all processes that have written to it.

• When a process writes to a file or other persistent object, its identity is added to a list associated with that file, which may only be removed after at least n seconds have passed.

• When a process reads from a file, a link is added from the reader to all processes that have written to it within the past n seconds.

Note that the process relationship graph must include processes that have terminated but which still have causal relationships to existing processes. Processes' identities must include all information that the application filter uses to identify processes, such as executable file names, shared library names, and file hashes. Using this graph, all processes having causal relationships to a process that is attempting to open a connection can be identified by performing a traversal of the weakly connected component of the graph rooted at the process that is attempting to open the connection; this set of processes will be known as a process group.

As an example, consider the following interactions:

1. Process A starts process B
2. Process B starts processes C and D

3. Process E writes to a pipe which is read by process C
4. Process D writes to a multicast socket, which is read by processes E, B, and F

The graph generated for these interactions will then be that shown in Figure 5.8. The groups for all processes are as shown in Table 5.8.

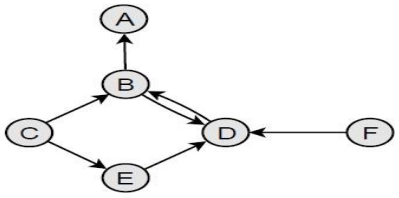

Figure 5.8: Graph of causal relationships between processes.

Process	Process group
A	{A}
B	{A, B, D}
C	{A, B, C, D, E}
D	{A, B, D}
E	{A, B, D, E}
F	{A, B, D, F}

Table 5.8: Process groups for all processes in

A firewall that attempts to detect connections by proxy in this manner would need to be configured with not only the set of programs allowed to make connections, but also all other programs allowed to be in the process groups of a program that is attempting to make a connection. Using the above example, a firewall could only allow connections from process D if it both recognizes processes A, B and D, and it was configured to allow connections from D while it has causal relationships from A and B. If, for instance, D was a web browser and A and B were recognized operating system components, then connections would be allowed. On the other hand, if A was not recognized by the firewall, then all connections from B would be rejected.

Unfortunately, this form of IPC tracking cannot detect all forms of interaction between processes. For example, the inetd program on Unix-based systems receives connections from remote processes and then executes arbitrary commands to handle them. There is no way to detect this interaction when the connection is opened, since the process interaction hasn't occurred yet. Programs may be able to use covert channels, such as those described by Lampson [Lam73] to pass commands in manners undetectable to this algorithm. This algorithm also can't detect any communication that is persistent across reboots, reads from files that were written to by processes that have timed out of the graph, or anything that involves processes on more than one computer. However, it can be used to track most common forms of inter-process communication and will be effective against most attempts to use another program as a network client.

Further Research in Application Firewalling

• The application firewalling techniques described in Chapter 5 have not been implemented and tested. While I believe them to be practical, I have no figures on the overhead imposed by such a system or the effect of this overhead on network communications. A complete implementation would require significant amounts of work, but would be necessary for a thorough analysis of the efficiency and effectiveness of such an application firewalling system.

• In Section 5.3.2, I identified several ways that my techniques for identifying legitimate applications could be fooled or bypassed using virtual machines, interpreters, or malicious operating systems. I do not believe that these can be solved without making unrealistic assumptions about the trustworthiness of software, but I have no proof of this.

• The process graph mechanism described in Section 5.3.3 should detect causal relationships between process established via communication over normal IPC channels, but does not take into account inter-process covert channels [Lam73]. It also does not take into account programs like inetd that accept connections and then execute arbitrary programs to handle them. It may be possible to devise another mechanism that can more accurately identify connections by proxy.

Even without these enhancements, application firewalls and covert authentication systems such as port knocking and SPA, as presented in this thesis, are useful tools that can significantly enhance existing firewall systems by providing them with additional information about who and what is attempting to communicate through them.

REFERENCES

[1] wikipedia.org. Slashdot effect. http://en.wikipedia.org/wiki/Slashdot effect,3 April 2014
[2] New Trojan disables firewall defenses http://totaldefense.iyogi.com/?p=77, 3 April 2014
[3] James Yonan. OpenVPN. http://openvpn.net/, 4.April 2014
[4] VMware, Inc. Vmware. http://www.vmware.com/, 2 April 2014
[5] About Network Security http://www.technologyevaluation.com/search/for /thesis-on-network-security.html,2 April,2014
[6] Harald Welte. Netfilter: firewalling, NAT, and packet mangling for Linux. http://www.netfilter.org/, 3 April 2014
[7] Online book Bruce Schneier. Applied Cryptography. John Wiley & Sons, Inc., 2nd edition, 2006.
[8] Wikipedia about Malware http://en.wikipedia.org/wiki/Malware , 5 April 2014
[9] About Port firewall http://support.microsoft.com/kb/308127 10 4 April 2014
[10]https://its.ucsc.edu/security/training/intro.html
[11] http://www.open.edu/openlearn/science-maths-technology/computing-and-ict/systems-computer/network-security/content-section---references
[12]https://searchnetworking.techtarget.com/feature/Network-Security-The-Complete-Reference-Chapter-10-Network-device-security
[13]http://people.scs.carleton.ca/~paulv/5900wBooks.html
[14] Boyle and Panko, Corporate Computer Security, 3/e (2013, Prentice Hall). See also: Panko, Corporate Computer and Network Security, 2/e (2009, Prentice Hall).

.

Syed Jamaluddin Ahmad, achieved Bachelor of Science in Computer Science and Engineering (BCSE) from Dhaka International University, Masters of Science in Computing Science Associates with research: Telecommunication Engineering from Athabasca University, Alberta, Canada and IT-Pro of Diploma from Global Business College, Munich, Germany. Presently Working as an Assistant Professor, Computer Science and Engineering, Shanto-Mariam University of Creative Technology, Dhaka, Bangladesh. Formerly, was head of the Department of Computer Science & Engineering, University of South Asia from 2012-2014, also Lecturer and Assistant Professor at Dhaka International University from 2005-2007 and 2011-2012 respectively and was a lecturer at Loyalist College, Canada, was Assistant Professor at American International University, Fareast International University, Royal University, Southeast University and Many more. He has **alrea**dy 15th international publications, 12th seminar papers, and conference articles. He is also a founder member of a famous IT institute named Arcadia IT (www.arcadia-it.com). Achieved Chancellor's Gold Crest in 2010 for M.Sc. in Canada and Outstanding result in the year of 2005. and obtained "President Gold Medal" for B.Sc.(Hon's). Best conductor award in Germany for IT relevant works. Membership of "The NewYork International Thesis Justification Institute, USA, British Council Language Club, National Debate Club, Dhaka, English Language Club and DIU . Developed projects: Mail Server, Web Server, Proxy Server, DNS(Primary, Secondary, Sub, Virtual DNS), FTP Server, Samba Server, Virtual Web Server, Web mail Server, DHCP Server, Dial in Server, Simulation on GAMBLING GAME Using C/C++, Inventory System Project, Single Server Queuing System Project, Multi Server Queuing System Project, Random walk Simulation Project, Pure Pursuit Project (Air Scheduling), Cricket Management Project, Daily Life Management Project, Many Little Projects Using Graphics on C/C++, Corporate Network With Firewall Configure OS:LINUX (REDHAT) Library Management Project Using Visual Basic, Cyber View Network System:Tools:Php OS: Windows Xp Back-end: My SQL Server, Online Shopping: Tools: Php, HTML, XML. OS:Windows Xp, Back-end: My SQL and Cyber Security" Activities-'Nirapad Cyber Jogat, Atai hok ajker shapoth'-To increase the awareness about the laws, 2006 (2013 amendment) of Information and Communication and attended Workshop on LINUX Authentication"-Lead by- Prof. Andrew Hall, Dean, Sorbon University, France, Organized By- Athabasca University, CANADA, April, 2009. His areas of interest include Data Mining, Big Data Management, Telecommunications, Network Security, WiFi, Wimax, 3g, 4g network, UNIX, LINUX Network Security,Programming Language(C/C++ or JAVA), Database (Oracle), Algorithm Design, Graphics Design & Image Processing and Algorithm Design.

Roksana Khandoker, achieved Bachelor of Science in Computer Science and Engineering (BCSE) from United International University, Masters of Science in Computer Science and Engineering from University of South Asia. Presently Working as a Senior Lecturer, Computer Science and Engineering, University of South Asia, Dhaka, Bangladesh. Formerly, was also a lecturer at different poly-technique institutes. She has 4th international journals and attended different international and national conferences. She is the Chairman of the famous IT institute named Arcadia IT and Chairman of Brighton International Alliance. Her areas of interest include Data Mining, Big Data Management, Telecommunications, Network Security, WiFi, Wimax, 3g and 4g network.

Farzana Nawrin achieved Bachelor of Science in Computer Science and Engineering (BCSE) from Dhaka City College under National University, Masters of Science in Computer Science from Jahangirnagar University. She achieved 1st class 1st both B.Sc. and M.Sc. degree. Presently working as a lecturer, Computer Science and Engineering department, Shanto-Mariam University of Creative Technology, Dhaka, Bangladesh. Formerly, was also a lecturer of Computer Science and Engineering department of Dhaka City College under National University. She has done two thesis about network security in her B.Sc. and M.Sc. level. She was also attended many national workshop. She has got special training from Bangladesh Institute of Design and Development(BIDD). She has special certification in CCNA and CompTIA A+. Her areas of interest include Network Security, Telecommunication, System analysis, Automata Design, Routing and Switching, Design and Analysis Compiler, Wifi,Wimax,3g and 4g Network.