Implementing a Distributed Firewall[*]

Sotiris Ioannidis
Univ. of Pennsylvania
sotiris@dsl.cis.upenn.edu

Angelos D. Keromytis
Univ. of Pennsylvania
adk@adk.gr

Steve M. Bellovin
AT&T Labs — Research
smb@research.att.com

Jonathan M. Smith
Univ. of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

Conventional firewalls rely on topology restrictions and controlled network entry points to enforce traffic filtering. Furthermore, a firewall cannot filter traffic it does not see, so, effectively, everyone on the protected side is trusted. While this model has worked well for small to medium size networks, networking trends such as increased connectivity, higher line speeds, extranets, and telecommuting threaten to make it obsolete.

To address the shortcomings of traditional firewalls, the concept of a "distributed firewall" has been proposed. In this scheme, security policy is still centrally defined, but enforcement is left up to the individual endpoints. IPsec may be used to distribute credentials that express parts of the overall network policy. Alternately, these credentials may be obtained through out-of-band means.

In this paper, we present the design and implementation of a distributed firewall using the KeyNote trust management system to specify, distribute, and resolve policy, and OpenBSD, an open source UNIX operating system.

General Terms

Security

Keywords

Firewalls, distributed, access control, IPsec, network security, Trust Management, KeyNote, OpenBSD, credentials, IP, IKE.

1. INTRODUCTION

A *firewall* is a collection of components, interposed between two networks, that filters traffic between them according to some security policy [8]. Conventional firewalls rely on network topology restrictions to perform this filtering. Furthermore, one key assumption under this model is that everyone on the protected network(s) is trusted (since internal traffic is not seen by the firewall, it cannot be filtered); if that is not the case, then additional, internal firewalls have to be deployed in the internal network.

While this model worked well for small to medium size networks, several trends in networking threaten to make it obsolete:

- Due to the increasing line speeds and the more computation-intensive protocols that a firewall must support (especially IPsec [1] [13]), firewalls tend to become congestion points. This gap between processing and networking speeds is likely to increase, at least for the foreseeable future; while computers (and hence firewalls) are getting faster, the combination of more complex protocols and the tremendous increase in the amount of data that must be passed through the firewall has been and likely will continue to outpace Moore's Law [10].
- There exist protocols, and new protocols are designed, that are difficult to process at the firewall, because the latter lacks certain knowledge that is readily available at the endpoints. FTP and RealAudio are two such protocols. Although there exist application-level proxies that handle such protocols, such solutions are viewed as architecturally "unclean" and in some cases too invasive.
- The assumption that all insiders are trusted [2] has not been valid for a long time. Specific individuals or remote networks may be allowed access to all or parts of the protected infrastructure (extranets, telecommuting, *etc.*). Consequently, the traditional notion of a security perimeter can no longer hold unmodified; for example, it is desirable that telecommuters' systems comply with the corporate security policy.
- Worse yet, it has become trivial for anyone to establish a new, unauthorized entry point to the network

[*]This work was supported by DARPA under Contract F39502-99-1-0512-MOD P0001.

[1]IPsec is a protocol suite, recently standardized by the IETF, that provides network-layer security services such as packet confidentiality, authentication, data integrity, replay protection, and automated key management.

[2]This is an artifact of firewall deployment: internal traffic that is not seen by the firewall cannot be filtered; as a result, internal users can mount attacks on other users and networks without the firewall being able to intervene. If firewalls were placed everywhere, this would not be necessary.

without the administrator's knowledge and consent. Various forms of tunnels, wireless, and dial-up access methods allow individuals to establish backdoor access that bypasses all the security mechanisms provided by traditional firewalls. While firewalls are in general not intended to guard against misbehavior by insiders, there is a tension between internal needs for more connectivity and the difficulty of satisfying such needs with a centralized firewall.

- Large (and even not-so-large) networks today tend to have a large number of entry points (for performance, failover, and other reasons). Furthermore, many sites employ internal firewalls to provide some form of compartmentalization. This makes administration particularly difficult, both from a practical point of view and with regard to policy consistency, since no unified and comprehensive management mechanism exists.

- End-to-end encryption can also be a threat to firewalls [3], as it prevents them from looking at the packet fields necessary to do filtering. Allowing end-to-end encryption through a firewall implies considerable trust to the users on behalf of the administrators.

- Finally, there is an increasing need for finer-grained (and even application-specific) access control which standard firewalls cannot readily accommodate without greatly increasing their complexity and processing requirements.

Despite their shortcomings, firewalls are still useful in providing some measure of security. The key reason that firewalls are still useful is that they provide an obvious, mostly hassle-free, mechanism for *enforcing* network security policy. For legacy applications and networks, they are the only mechanism for security. While newer protocols typically have some provisions for security[3], older protocols (and their implementations) are more difficult, often impossible, to secure. Furthermore, firewalls provide a convenient first-level barrier that allows quick responses to newly-discovered bugs.

To address the shortcomings of firewalls while retaining their advantages, [3] proposed the concept of a *distributed firewall.* In distributed firewalls, security policy is defined centrally but enforced at each invididual network endpoint (hosts, routers, *etc.*). The system propagates the central policy to all endpoints. Policy distribution may take various forms. For example, it may be pushed directly to the end systems that have to enforce it, or it may be provided to the users in the form of credentials that they use when trying to communicate with the hosts, or it may be a combination of both. The extent of mutual trust between endpoints is specified by the policy.

To implement a distributed firewall, three components are necessary:

- A language for expressing policies and resolving requests. In their simplest form, policies in a distributed firewall are functionally equivalent to packet filtering rules. However, it is desirable to use an extensible system (so other types of applications and security checks can be specified and enforced in the future). The language and resolution mechanism should also support credentials, for delegation of rights and authentication purposes [4].

- A mechanism for safely distributing security policies. This may be the IPsec key management protocol when possible, or some other protocol. The integrity of the policies transfered must be guaranteed, either through the communication protocol or as part of the policy object description (*e.g.,* they may be digitally signed).

- A mechanism that applies the security policy to incoming packets or connections, providing the enforcement part.

Our prototype implementation uses the KeyNote trust-management system, which provides a single, extensible language for expressing policies and credentials. Credentials in KeyNote are signed, thus simple file-transfer protocols may be used for policy distribution. We also make use of the IPsec stack in the OpenBSD system to authenticate users, protect traffic, and distribute credentials. The distribution of credentials and user authentication occurs are part of the Internet Key Exchange (IKE) [12] negotiation. Alternatively, policies may be distributed from a central location when a policy update is performed, or they may be fetched as-needed (from a web server, X.500 directory, or through some other protocol).

Since KeyNote allows delegation, decentralized administration becomes feasible (establishing a hierarchy or web of administration, for the different departments or even individual systems). Users are also able to delegate authority to access machines or services they themselves have access to. Although this may initially seem counter-intuitive (after all, firewalls embody the concept of centralized control), in our experience users can almost always [4] bypass a firewall's filtering mechanisms, usually by the most insecure and destructive way possible (*e.g.,* giving away their password, setting up a proxy or login server on some other port, *etc.*). Thus, it is better to allow for some flexibility in the system, as long as the users follow the overall policy. Also note that it is possible to "turn off" delegation.

Thus, the overall security policy relevant to a particular user and a particular end host is the composition of the security policy "pushed" to the end host, any credentials given to the user, and any credentials stored in a central location and retrieved on-demand. Finally, we implement the mechanism that enforces the security policy in a TCP-connection granularity. In our implementation, the mechanism is split in two parts, one residing in the kernel and the other in a user-level process.

1.1 Paper Organization

The remainder of the paper is organized as follows: Section 2 discusses the distributed firewall concept and related issues. Section 3 gives an overview of the KeyNote trust-management system, which we use for policy specification and processing. Section 4 discusses the current implementation of the distributed firewall, and section 5 describes future development. Section 6 presents some related work.

[3]This is by no means a universal trait, and even today there are protocols designed with no security review.

[4]With the possible exception of military-grade systems or networks.

Section 7 summarizes how distributed firewalls address the problems of traditional firewalls, and concludes this paper.

2. THE DISTRIBUTED FIREWALL

A distributed firewall, of the type described in [3], uses a central policy, but pushes enforcement towards the edges. That is, the policy defines what connectivity, inbound and outbound, is permitted; this policy is distributed to all endpoints, which enforce it.

In the full-blown version, endpoints are characterized by their IPsec identity, typically in the form of a certificate. Rather than relying on the topological notions of "inside" and "outside", as is done by a traditional firewall, a distributed firewall assigns certain rights to whichever machines own the private keys corresponding to certain public keys. Thus, the right to connect to the `http` port on a company's internal Web server might be granted to those machines having a certificate name of the form `*.goodfolks.org`, rather than those machines that happen to be connected to an internal wire. A laptop directly connected to the Internet has the same level of protection as does a desktop in the organization's facility. Conversely, a laptop connected to the corporate net by a visitor would not have the proper credentials, and hence would be denied access, even though it is topologically "inside."

To implement a distributed firewall, we need a security policy language that can describe which connections are acceptable, an authentication mechanism, and a policy distribution scheme. As a policy specification language, we use the KeyNote trust-management system, further described in Section 3.

As an authentication mechanism, we decided to use IPsec for traffic protection and user/host authentication. While we can, in principle, use application-specific security mechanisms (*e.g.,* SSL-enabled web-browsing), this would require extensive modifications of all such applications to make them aware of the filtering mechanism. Furthermore, we would then depend on the good behavior of the very applications we are trying to protect. Finally, it would be impossible to secure legacy applications with inadequate provisioning for security.

When it comes to policy distribution, we have a number of choices:

- We can distribute the KeyNote (or other) credentials to the various end users. The users can then deliver their credentials to the end hosts through the IKE protocol. The users do not have to be online for the policy update; rather, they can periodically retrieve the credentials from a repository (such as a web server). Since the credentials are signed and can be transmitted over an insecure connection, users could retrieve their new credentials even when the old ones have expired. This approach also prevents, or at least mitigates, the effects of some possible denial of service attacks.

- The credentials can be pushed directly to the end hosts, where they would be immediately available to the policy verifier. Since every host would need a large number, if not all, of the credentials for every user, the storage and transmission bandwidth requirements are higher than in the previous case.

- The credentials can be placed in a repository where they can be fetched as needed by the hosts. This requires constant availability of the repository, and may impose some delays in the resolution of request (such as a TCP connection establishment).

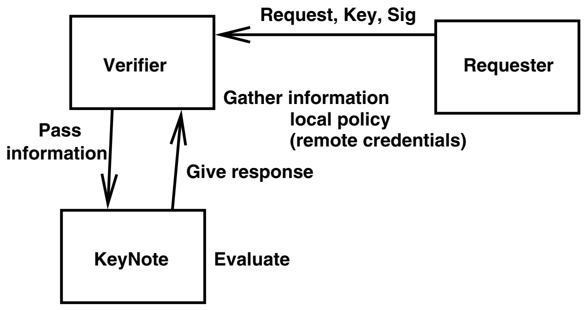

Figure 1: Application Interactions with KeyNote. The Requester is typically a user that authenticates through some application-dependent protocol, and optionally provides credentials. The Verifier needs to determine whether the Requester is allowed to perform the requested action. It is responsible for providing to KeyNote all the necessary information, the local policy, and any credentials. It is also responsible for acting upon KeyNote's response.

While the first case is probably the most attractive from an engineering point of view, not all IKE implementations support distribution of KeyNote credentials. Furthermore, some IPsec implementations do not support connection--grained security. Finally, since IPsec is not (yet) in wide use, it is desirable to allow for a policy-based filtering that does not depend on IPsec. Thus, it is necessary to provide a *policy resolution* mechanism that takes into consideration the connection parameters, the local policies, and any available credentials (retrieved through IPsec or other means), and determines whether the connection should be allowed. We describe our implementation of such a mechanism for the OpenBSD system in Section 4.

3. KEYNOTE

Trust Management is a relatively new approach to solving the authorization and security policy problem, and was introduced in [6]. Making use of public key cryptography for authentication, trust management dispenses with unique names as an indirect means for performing access control. Instead, it uses a direct binding between a public key and a set of authorizations, as represented by a safe programming language. This results in an inherently decentralized authorization system with sufficient expressibility to guarantee flexibility in the face of novel authorization scenarios.

One instance of a trust-management system is KeyNote. KeyNote provides a simple notation for specifying both local security policies and credentials that can be sent over an untrusted network. Policies and credentials contain predicates that describe the trusted actions permitted by the holders of specific public keys (otherwise known as principals). Signed credentials, which serve the role of "certificates," have the same syntax as policy assertions, but are also signed by the entity delegating the trust. For more details on the KeyNote language itself, see [5].

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: "rsa-hex:1023abcd"
Comment: Allow Licensee to connect to local port 23 (telnet) from
         internal addresses only, or to port 22 (ssh) from anywhere.
         Since this is a policy, no signature field is required.
Conditions: (local_port == "23" && protocol == "tcp" &&
             remote_address > "158.130.006.000" &&
             remote_address < "158.130.007.255) -> "true";
            local_port == "22" && protocol == "tcp" -> "true";

KeyNote-Version: 2
Authorizer: "rsa-hex:1023abcd"
Licensees: "dsa-hex:986512a1" || "x509-base64:19abcd02=="
Comment: Authorizer delegates SSH connection access to either
         of the Licensees, if coming from a specific address.
Conditions: (remote_address == "139.091.001.001" &&
             local_port == "22") -> "true";
Signature: "rsa-md5-hex:f00f5673"
```

Figure 2: Example KeyNote Policy and Credential. The local policy allows a particular user (as identified by their public key) connect access to the telnet port by internal addresses, or to the SSH port from any address. That user then delegates to two other users (keys) the right to connect to SSH from one specific address. Note that the first key can effectively delegate at most the same rights it possesses. KeyNote does not allow *rights amplification*; any delegation acts as *refinement.*

Applications communicate with a "KeyNote evaluator" that interprets KeyNote assertions and returns results to applications, as shown in Figure 1. However, different hosts and environments may provide a variety of interfaces to the KeyNote evaluator (library, UNIX daemon, kernel service, *etc.*).

A KeyNote evaluator accepts as input a set of local policy and credential assertions, and a set of attributes, called an "action environment," that describes a proposed trusted action associated with a set of public keys (the requesting principals). The KeyNote evaluator determines whether proposed actions are consistent with local policy by applying the assertion predicates to the action environment. The KeyNote evaluator can return values other than simply true and false, depending on the application and the action-environment definition. An important concept in KeyNote (and, more generally, in trust management) is *"monotonicity"*. This simply means that given a set of credentials associated with a request, if there is any subset that would cause the request to be approved then the complete set will also cause the request to be approved. This greatly simplifies both request resolution (even in the presence of conflicts) and credential management. Monotonicity is enforced by the KeyNote language (it is not possible to write non-monotonic policies).

It is worth noting here that although KeyNote uses cryptographic keys as principal identifiers, other types of identifiers may also be used. For example, usernames may be used to identify principals inside a host. In this environment, delegation must be controlled by the operating system (or some implicitly trusted application), similar to the mechanisms used for transfering credentials in Unix or in capability-based systems. Also, in the absence of cryptographic authentication, the identifier of the principal requesting an action must be securely established. In the example of a single host, the operating system can provide this information.

In our prototype, end hosts (as identified by their IP address) are also considered principals when IPsec is not used to secure communications. This allows local policies or credentials issued by administrative[5] keys to specify policies similar to current packet filtering rules. Naturally, such policies or credentials implicitly trust the validity of an IP address as an identifier. In that respect, they are equivalent to standard packet filtering. The only known solution to this is the use of cryptographic protocols to secure communications.

Since KeyNote allows multiple policy constraints, potentially for different applications, to be contained in the same assertion, it is trivial to support application-specific credentials. Credentials that specify, *e.g.,* Java applet permissions, could be delivered under any of the distribution schemes described in Section 2, and made available to the end application through some OS-specific mechanism (*e.g.,* `getsockopt(2)` calls).

In the context of the distributed firewall, KeyNote allows us to use the same, simple language for both policy and credentials. The latter, being signed, may be distributed over an insecure communication channel. In KeyNote, credentials may be considered as an extension, or refinement, of local policy; the union of all policy and credential assertions is the overall network security policy. Alternately, credentials may be viewed as parts of a hypothetical access matrix. End hosts may specify their own security policies, or they may depend exclusively on credentials from the administrator, or do anything in between these two ends of the spectrum. Perhaps of more interest, it is possible to "merge" policies from different administrative entities and process them unambiguously, or to layer them in increasing

[5]Note that from the point of view of KeyNote, all keys are "born" equivalent. The distinction between "administrative" and other keys lies only in the amount of trust placed on them by the local policies of end hosts.

```
KeyNote-Version: 2
Authorizer: "rsa-hex:1023abcd"
Licensees: "IP:158.130.6.141"
Conditions:  (@remote_port < 1024 &&
              @local_port == 22) -> "true";
Signature: "rsa-sha1-hex:bee11984"
```

Figure 3: An example credential where an (administrative) key delegates to an IP address. This would allow the specified address to connect to the local SSH port, if the connection is coming from a privileged port. Since the remote host has no way of supplying the credential to the distributed firewall through a security protocol like IPsec, the distributed firewall must search for such credentials or must be provided with them when policy is generated/updated.

levels of refinement. This merging can be expressed in the KeyNote language, in the form of intersection (conjunction) and union (disjunction) of the component sub-policies.

Although KeyNote uses a human-readable format and it is indeed possible to write credentials and policies that way, our ultimate goal is to use it as an interoperability-layer language that "ties together" the various applications that need access control services. An administrator would use a higher-level language (*e.g.,* [2]) or GUI to specify correspondingly higher-level policy and then have this compiled to a set of KeyNote credentials. This higher-level language would provide grouping mechanisms and network-specific abstractions (for networks, hosts, services, *etc.*) that are not present in KeyNote. Using KeyNote as the middle language offers a number of benefits:

- It can handle a variety of different applications (since it is application-independent but customizable), allowing for more comprehensive and mixed-level policies (*e.g.,* covering email, active code content, IPsec, *etc.*).
- Provides built-in delegation, thus allowing for decentralized administration.
- Allows for incremental or localized policy updates (as only the relevant credentials need to be modified, produced, or revoked).

Figure 2 shows two sample KeyNote assertions, a policy and a (signed) credential. Figure 3 shows an example of a key delegating to an IP address. For more details on KeyNote, see [5, 7].

4. IMPLEMENTATION

For our development platform we decided to use the OpenBSD operating system [11]. OpenBSD provides an attractive platform for developing security applications because of the well-integrated security features and libraries (an IPsec stack, SSL, KeyNote, *etc.*). However, similar implementations are possible under other operating systems.

Our system is comprised of three components: a set of kernel extensions, which implement the enforcement mechanisms, a user level daemon process, which implements the distributed firewall policies, and a device driver, which is used for two-way communication between the kernel and the policy daemon. Our prototype implementation totals approximately 1150 lines of C code; each component is roughly the same size.

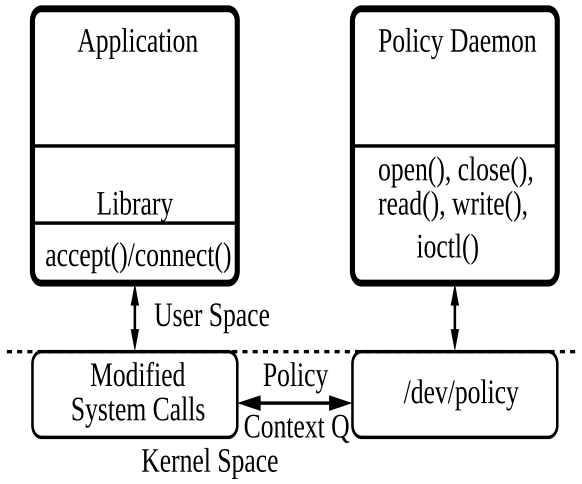

Figure 4: The Figure shows a graphical representation of the system, with all its components. The core of the enforcement mechanism lives in kernel space and is comprised of the two modified system calls that interest us, `connect(2)` and `accept(2)`. The policy specification and processing unit lives in user space inside the policy daemon process. The two units communicate via a loadable pseudo device driver interface. Messages travel from the system call layer to the user level daemon and back using the *policy context queue.*

Figure 4 shows a graphical representation of the system, with all its components. In the following three subsections we describe the various parts of the architecture, their functionality, and how they interact with each other.

4.1 Kernel Extensions

For our working prototype we focused our efforts on the control of the TCP connections. Similar principles can be applied to other protocols; for unreliable protocols, some form of reply caching is desirable to improve performance. We discuss a more general approach in Section 5.

In the UNIX operating system users create outgoing and allow incoming TCP connections using the `connect(2)` and `accept(2)` system calls respectively. Since any user has access to these system calls, some "filtering" mechanism is needed. This filtering should be based on a policy that is set by the administrator.

Filters can be implemented either in user space or inside the kernel. Each has its advantages and disadvantages.

A user level approach, as depicted in Figure 5, requires each application of interest to be linked with a library that provides the required security mechanisms, *e.g.,*, a modified `libc`. This has the advantage of operating system-independence, and thus does not require any changes to the kernel code. However, such a scheme does not guarantee that the applications *will* use the modified library, potentially leading to a major security problem.

A kernel level approach, as shown in the left side of Figure 4, requires modifications to the operating system kernel.

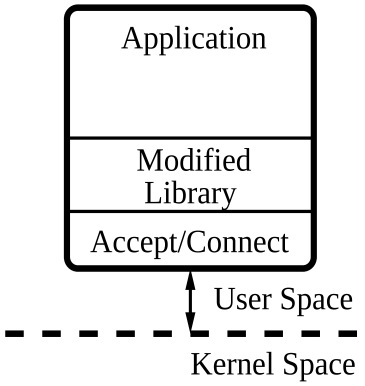

Figure 5: Wrappers for filtering the `connect(2)` and `accept(2)` system calls are added to a system library. While this approach offers considerable flexibility, it suffers from its inability to guarantee the enforcement of security policies, as applications might not link with the appropriate library.

This restricts us to open source operating systems like BSD and Linux. The main advantage of this approach is that the additional security mechanisms can be enforced transparently on the applications.

As we mentioned previously, the two system calls we need to filter are `connect(2)` and `accept(2)`. When a `connect(2)` is issued by a user application and the call traps into the kernel, we create what we call a *policy context* (see Figure 6), associated with that connection.

The policy context is a container for all the information related to that specific connection. We associate a sequence number to each such context and then we start filling it with all the information the *policy daemon* will need to decide whether to permit it or not. In the case of the `connect(2)`, this includes the ID of the user that initiated the connection, the destination address and port, *etc.* Any credentials acquired through IPsec may also be added to the context at this stage. There is no limit as to the kind or amount of information we can associate with a context. We can, for example, include the time of day or the number of other open connections of that user, if we want them to be considered by our decision–making strategy.

Once all the information is in place, we *commit* that context. The commit operation adds the context to the list of contexts the policy daemon needs to handle. After this, the application is blocked waiting for the policy daemon reply.

Accepting a connection works in a similar fashion. When `accept(2)` enters the kernel, it blocks until an incoming connection request arrives. Upon receipt, we allocate a new context which we fill in similarly to the `connect(2)` case. The only difference is that we now also include the source address and port. The context is then enqueued, and the process blocks waiting for a reply from the policy daemon.

In the next section we discuss how messages are passed between the kernel and the policy daemon.

4.2 Policy Device

To maximize the flexibility of our system and allow for easy experimentation, we decided to make the policy daemon a user level process. To support this architecture, we implemented a *pseudo device driver*, `/dev/policy`, that serves as a communication path between the user–space policy daemon, and the modified system calls in the kernel. Our device driver supports the usual operations (`open(2)`, `close(2)`, `read(2)`, `write(2)`, and `ioctl(2)`). Furthermore, we have implemented the device driver as a loadable module. This increases the functionality of our system even more, since we can add functionality dynamically, without needing to recompile the whole kernel.

If no policy daemon has opened `/dev/policy`, no connection filtering is done. Opening the device activates the distributed firewall and initializes data structures. All subsequent `connect(2)` and `accept(2)` calls will go through the procedure described in the previous section. Closing the device will free any allocated resources and disable the distributed firewall.

When reading from the device the policy daemon blocks until there are requests to be served. The policy daemon handles the policy resolution messages from the kernel, and writes back a reply. The `write(2)` is responsible for returning the policy daemons decision to the blocked connection call, and then waking it up. It should be noted that both the device and the associated messaging protocol are not tied to any particular type of application, and may in fact be used without any modifications by other kernel components that require similar security policy handling.

Finally, we have included an `ioctl(2)` call for "house–keeping". This allows the kernel and the policy daemon to re–synchronize in case of any errors in creating or parsing the request messages, by discarding the current policy context and dropping the associated connection.

4.3 Policy Daemon

The third and last component of our system is the policy daemon. It is a user level process responsible for making decisions, based on policies that are specified by some administrator and credentials retrieved remotely or provided by the kernel, on whether to allow or deny connections.

Policies, as shown in Figure 2, are initially read in from a file. It is possible to remove old policies and add new ones dynamically. In the current implementation, such policy changes only affect new connections. In Section 5 we will discuss how these changes can potentially be made to affect existing connections, if such functionality is required.

Communication between the policy daemon and the kernel is possible, as we mentioned earlier, using the `policy` device. The daemon receives each request (see Figure 7) from the kernel by reading the device. The request contains all the information relevant to that connection as described in Section 4.1. Processing of the request is done by the daemon using the KeyNote library, and a decision to accept or deny it is reached. Finally the daemon writes the reply back to the kernel and waits for the next request. While the information received in a particular message is application-dependent (in our case, relevant to the distributed firewall), the daemon itself has no awareness of the specific application. Thus, it can be used to provide policy resolution services for many different applications, literally without any modifications.

When using a remote repository server, the daemon can fetch a credential based on the ID of the user associated with a connection, or with the local or remote IP address (such

```
typedef struct  policy_mbuf      policy_mbuf;
struct  policy_mbuf     {
        policy_mbuf     *next;
        int              length;
        char             data[POLICY_DATA_SIZE];
};

typedef struct  policy_context  policy_context;
struct  policy_context  {
        policy_mbuf     *p_mbuf;
        u_int32_t        sequence;
        char            *reply;
        policy_context  *policy_context_next;
};

policy_context  *policy_create_context(void);
void            policy_destroy_context(policy_context *);
void            policy_commit_context(policy_context *);
void            policy_add_int(policy_context *, char *, int);
void            policy_add_string(policy_context *, char *, char *);
void            policy_add_ipv4addr(policy_context *, char *, in_addr_t *);
```

Figure 6: The `connect(2)` and `accept(2)` system calls create *contexts* which contain information relevant to that connection. These are appended to a queue from which the policy daemon will receive and process them. The policy daemon will then return to the kernel a decision on whether to accept or deny the connection.

```
u_int32_t seq;    /* Sequence Number */
u_int32_t uid;            /* User Id */
u_int32_t N;     /* Number of Fields */
u_int32_t l[N]; /* Lengths of Fields */
char      *field[N];       /* Fields */
```

Figure 7: The request to the policy daemon is comprised of the following fields: a sequence number uniquely identifying the request, the ID of the user the connection request belongs to, the number of information fields that will be included in the request, the lengths of those fields, and finally the fields themselves.

credentials may look like the one in Figure 3). A very simple approach to that is fetching the credentials via HTTP from a remote web server. The credentials are stored by user ID and IP address, and provided to anyone requesting them. If credential "privacy" is a requirement, one could secure this connection using IPsec or SSL. To avoid potential deadlocks, the policy daemon is not subject to the connection filtering mechanism.

4.4 Example Scenario

To better explain the interaction of the various components in the distributed firewall, we discuss the course of events during two incoming TCP connection requests, one of which is IPsec–protected. The local host where the connection is coming is part of a distributed firewall, and has a local policy as shown in Figure 8.

In the case of a connection coming in over IPsec, the remote user or host will have established an IPsec Security Association with the local host using IKE. As part of the IKE exchange, a KeyNote credential as shown in Figure 9 is provided to the local host. Once the TCP connection is

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: ADMINISTRATIVE_KEY
```

Figure 8: End-host local security policy. In our particular scenario, the policy simply states that some administrative key will specify our policy, in the form of one or more credentials. The lack of a Conditions field means that there are no restrictions imposed on the policies specified by the administrative key.

received, the kernel will construct the appropriate context as discussed in Section 4.1. This context will contain the local and remote IP addresses and ports for the connection, the fact that the connection is protected by IPsec, the time of day, *etc.* This information along with the credential acquired via IPsec will be passed to the policy daemon. The policy daemon will perform a KeyNote evaluation using the local policy and the credential, and will determine whether the connection is authorized or not. In our case, the positive response will be sent back to the kernel, which will then permit the TCP connection to proceed. Note that more credentials may be provided during the IKE negotiation (for example, a chain of credentials delegating authority).

If KeyNote does not authorize the connection, the policy daemon will try to acquire relevant credentials by contacting a remote server where these are stored. In our current implementation, we use a web server as the credential repository. In a large-scale network, a distributed/replicated database could be used instead. The policy daemon uses the public key of the remote user (when it is known, *i.e.,* when IPsec is in use) and the IP address of the remote host as the keys to lookup credentials with; more specifically, credentials where the user's public key or the remote host's address appears in

```
KeyNote-Version: 2
Authorizer: ADMINISTRATIVE_KEY
Licensees: USER_KEY
Conditions:
    (app_domain == "IPsec policy" &&
     encryption_algorithm == "3DES" &&
     local_address == "158.130.006.141")
          -> "true";
    (app_domain ==
          "Distributed Firewall" &&
     @local_port == 23 &&
     encrypted == "yes" &&
     authenticated == "yes") -> "true";
Signature: ...
```

Figure 9: A credential from the administrator to some user, authorizing that user to establish an IPsec Security Association (SA) with the local host and to connect to port 23 (telnet) over that SA. To do this, we use the fact that multiple expressions can be included in a single KeyNote credential. Since IPsec also enforces some form of access control on packets, we could simplify the overall architecture by skipping the security check for TCP connections coming over an IPsec tunnel. In that case, we could simply merge the two clauses (the IPsec policy clause could specify that the specific user may talk to TCP port 23 only over that SA).

the Licensees field are retrieved and cached locally (Figure 3 lists an example credential that refers to an IP address). These are then used in conjunction with the information provided by the kernel to re-examine the request. If it is again denied, the connection is ultimately denied.

5. FUTURE WORK

There are a number of possible extensions that we plan to work on in the process of building a more general and complete system.

As part of the STRONGMAN project at the University of Pennsylvania, we are examining the application of higher-level security policy languages to large-scale network management. KeyNote is used as a common language for expressing policies that can be distributed in different applications and systems. The distributed firewall is an important component in the STRONGMAN architecture. This is a subject of ongoing research.

As we described in Section 4.3, the policy daemon runs as a user level process that communicates with the kernel via a device driver. This design maximizes the flexibility of our system and allows for easy experimentation. Unfortunately, it adds the overhead of cross-domain calls between user space and kernel. An alternate design would be to run the policy daemon inside the kernel, much like `nfssvc(2)`. The policy daemon will then have direct access to the policy context queue, eliminating the system call overhead.

Our current system focuses on controlling TCP connections. We plan to expand our implementation by adding an IP filter-like mechanism for a more fine grained control (perhaps based on some existing filtering package, like IPF). This will allow per-packet, as opposed to per-connection, policing. Apart from protecting applications based on UDP, a packet-based implementation would also limit the types of "scanning" that an attacker might perform. In order to avoid the high overhead of such an approach we plan to use "policy caching." With policy caching, we invoke the policy daemon only the first time we encounter a new type of packet, *e.g.*, a packet belonging to a new connection. The decision of the policy daemon may be cached (subject to policy) in the filtering mechanism, and any other packets of the same type will be treated accordingly. In the event of a policy change, we can simply flush the cache. Given the simplicity of the KeyNote language, it is also possible to statically analyze the policies and credentials and derive in advance the necessary packet filtering rules, as a form of pre-caching. This however imposes greater demands on the credential-distribution mechanism.

We should note here that in our view most communications should be secured end-to-end (via IPsec or other similar mechanism); thus, most hosts would have a minimal set of filtering entries established at boot time. The rest of the rules would be established dynamically through the key exchange (or policy discovery) mechanisms in IPsec (or equivalent protocol). This approach can potentially scale much better than initializing or updating packet filters at the same time policy is updated. The cost of always-encrypted communication is less than one might think: PCI cards that cost less than US$300 retail (year 2000 prices) can easily achieve 100Mbps sustained throughput while encrypting/decrypting; there also exist a number of ethernet cards with built-in support for IPsec, potentially allowing for even higher throughput and much lower cost.

Another point to address is policy updates. As we mentioned in Section 4.3, we can update the policies of the daemon dynamically. However, policy updates do not affect already existing connections in the current implementation. We would like to add a revocation mechanism that goes through the list of all the connections and re-applies the policies. Connections that do not comply with the changes will be terminated by the kernel. One obvious method of doing so is by adding an `ioctl(2)` call that notifies the kernel of a policy update; the kernel then walks the list of TCP connections and sends a policy resolution request to the policy daemon, pretending the connection was just initiated. This approach requires minimal modifications in our existing system.

We have already mentioned that KeyNote may be used to express application-specific policies, and the relevant credentials may be distributed over the same channels (IPsec, web server, *etc.*). The interaction between application-specific and lower-level (such as those equivalent to packet filtering) policies is of particular interest, as it is possible to do very fine-grained access control by appropriately mixing these.

One final point to address is *credential discovery.* Users need a way to discover what credentials they (might) need to supply to the system along with their request for a connection. The policy daemon can then process these credentials along with the request. A simple way of adding this capability is by using the already existing `setsockopt(2)` system call. These credentials will then be added to any policy context requests associated with the socket.

6. RELATED WORK

A lot of work has been done over the previous years in the area of (traditional) firewalls[8, 16, 17].

[19] and [15] describe different approaches to host-based enforcement of security policy. These mechanisms depend on the IP addresses for access control, although they could potentially be extended to support some credential-based policy mechanism similar to what we describe in our paper.

The Napoleon system [18] defines a layered group-based access control scheme that is in some ways similar to the distributed firewall concept we have described, although it is mostly targeted to RMI environments like CORBA. Policies are compiled to Access Control Lists (ACLs) appropriate for each application (in our case, that would be each end host) and pushed out to them at policy creation or update time.

The STRONGMAN project at the University of Pennsylvania is aiming at simplifying security policy management by providing an application-independent policy specification language that can be compiled to application-specific KeyNote credentials. These credentials can then be distributed to applications, hosts, and end users and used in an integrated policy framework.

The Adage/Pledge system uses SSL and X.509-based authentication to provide applications with a library that allows centralized rights management.

[1] presents an in-depth discussion of the advantages and disadvantages of credential-based access control.

SnareWork [9] is a DCE-based system that can provide transparent security services (including access control) to end-applications, through use of wrapper modules that understand the application-specific protocols. Policies are compiled to ACLs and distributed to the various hosts in the secured network, although a pull-based method can also be used. Connections to protected ports are reported to a local security manager which decides whether to drop, allow, or forward them (using DCE RPC) to a remote host, based on the ACLs.

Perhaps the most relevant work is that of [2]. The approach there is use of a "network grouping" language that is customized for each managed firewall at that firewall. The language used is independent of the firewalls and routers used. In our approach, we introduce a three-layer system: a high-level policy language (equivalent in some sense to that used in Firmato), an intermediate level language (KeyNote) used by the mechanisms, and the actual mechanisms enforcing policy. This allows us to:

1. Express multi-application policies, rather than just packet filtering rules.

2. Express Mixed-layer policies (*e.g.,* policies of the type "email has to either be signed in the application layer or delivered over an IPsec SA that was authenticated with a credential matching the user in the From field of the email").

3. Permit delegation, which enables decentralized management (since KeyNote allows building arbitrary hierarchies of trust).

4. Allows incremental and asynchronous policy updates, since, when policy changes, only the relevant KeyNote credentials need to be updated and distributed (*e.g.,* only those relevant to a specific firewall).

7. CONCLUSION

We have discussed the concept of a distributed firewall. Under this scheme, network security policy specification remains under the control of the network administrator. Its enforcement, however, is left up to the hosts in the protected network. Security policy is specified using KeyNote policies and credentials, and is distributed (through IPsec, a web server, a directory-like mechanism, or some other protocol) to the users and hosts in the network. Since enforcement occurs at the endpoints, various shortcomings of traditional firewalls are overcome:

- Security is no longer dependent on restricting the network topology. This allows considerable flexibility in defining the "security perimeter," which can easily be extended to safely include remote hosts and networks (*e.g.,* telecommuters, extranets).

- Since we no longer solely depend on a single firewall for protection, we eliminate a performance bottleneck. Alternately, the burden placed on the traditional firewall is lessened significantly, since it delegates a lot of the filtering to the end hosts.

- Filtering of certain protocols (*e.g.,* FTP) which was difficult when done on a traditional firewall, becomes significantly easier, since all the relevant information is present at the decision point, *i.e.,* the end host.

- The number of outside connections the protected network is no longer a cause for administration nightmares. Adding or removing links has no impact on the security of the network. "Backdoor" connections set up by users, either intentionally or inadvertently, also do not create windows of vulnerability.

- Insiders may no longer be treated as unconditionally trusted. Network compartmentalization becomes significantly easier.

- End-to-end encryption is made possible without sacrificing security, as was the case with traditional firewalls. In fact, end-to-end encryption greatly improves the security of the distributed firewall.

- Application-specific policies may be made available to end-applications over the same distribution channel.

- Filtering (and other policy) rules are distributed and established on an as-needed basis; that is, only the hosts that actually need to communicate need to determine what the relevant policy with regard to each other is. This significantly eases the task of policy updating, and does not require each host/firewall to maintain the complete set of policies, which may be very large for large networks (*e.g.,* the AT&T phone network). Furthermore, policies and their distribution scales much better with respect to the network size and user base than a more tighly-coupled and synchronized approach would.

On the other hand, a distributed firewall architecture requires high quality administration tools, and de facto places high confidence in them. We believe that this is an inevitable trend however, even if traditional firewalls are utilized; already, large networks with a modest number of perimeter firewalls are becoming difficult to manage manually.

Also, note that the introduction of a distributed firewall infrastructure in a network does not completely eliminate the need for a traditional firewall. The latter is still useful in certain tasks:

- It is easier to counter infrastructure attacks that operate at a level lower than the distributed firewall. Note that this is mostly an implementation issue; there is no reason why a distributed firewall cannot operate at arbitrarily low layers, other than potential performance degradation.
- Denial-of-service attack mitigation is more effective at the network ingress points (depending on the particular kind of attack).
- Intrusion detection systems are more effective when located at a traditional firewall, where complete traffic information is available.
- The traditional firewall may protect end hosts that do not (or cannot) support the distributed firewall mechanisms. Integration with the policy specification and distribution mechanisms is especially important here, to avoid duplicated filters and windows of vulnerability.
- Finally, a traditional firewall may simply act as a fail-safe security mechanism.

Since most of the security enforcement has been moved to the end hosts, the task of a traditional firewall operating in a distributed firewall infrastructure is significantly eased. The interactions between traditional (and, even more interesting, transparent) and distributed firewalls are a subject for future research.

A final point is that, from an administrative point of view, a fully-distributed firewall architecture is very similar to a network with a large number of internal firewalls. The mechanism we have already described may be used in both environments. The two main differences between the two approaches lie in the granularity of "internal" protection (which also depends on the protected subnet topology, *e.g.,* switched or broadcast) and the end-to-end security guarantees (better infrastructure support is needed to make IPsec work through a firewall; alternately, transparent firewalls may be used [14]).

We have demonstrated the feasibility of the distributed firewall by building a working prototype. Further experimentation is needed to determine the robustness, efficiency, and scalability of this architecture. We hope that our work will stimulate further research in this area.

8. ACKNOWLEDGEMENTS

We would like to thank the CCS 2000 anonymous reviewers and Avishai Wool for their comments and suggestions on improving this paper.

9. REFERENCES

[1] T. Aura. Distributed access rights management with delegation certificates. In *Secure Internet Programming* [20], pages 211–235.

[2] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 17–31, May 1999.

[3] S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, November 1999.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming* [20], pages 185–210.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust management system version 2. Internet RFC 2704, September 1999.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.

[7] M. Blaze, J. Ioannidis, and A. Keromytis. Trust Management and Network Layer Security Protocols. In *Proceedings of the 1999 Cambridge Security Protocols International Workshop*, 1999.

[8] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[9] J. Chinitz and S. Sonnenberg. A Transparent Security Framework For TCP/IP and Legacy Applications. Technical report, Intellisoft Corp., August 1996.

[10] M. Dahlin. *Serverless Network File Systems*. PhD thesis, University of California, Berkeley, Dec. 1995.

[11] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proc. of the 1999 USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.

[12] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, Nov. 1998.

[13] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, Nov. 1998.

[14] A. D. Keromytis and J. L. Wright. Transparent Network Security Policy Enforcement. In *Proceedings of the Annual USENIX Technical Conference*, pages 215–226, June 2000.

[15] W. LeFebvre. Restricting network access to system daemons under SunOS. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 93–103, 1992.

[16] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[17] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the USENIX Summer 1989 Conference*, pages 203–221, 1989.

[18] D. Thomsen, D. O'Brien, and J. Bogle. Role Based Access Control Framework for Network Enterprises. In *Proceedings of the 14th Annual Computer Security Applications Conference*, December 1998.

[19] W. Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 85–92, 1992.

[20] J. Vitek and C. Jensen. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.