Abstract -- The TIS Labs Advanced Security Proxies’ (ASP) project is investigating software architectures for high-performance firewalls to enable the secure use of next generation networks. The project objective is to demonstrate an architecture and implementation in which protocol-specific proxies control when data transmission is allowed across the firewall, but which allows the proxy a range of options in determining how that data transits the firewall. By employing proxies that selectively use a range of lower-level protocol stack features, this novel architecture provides higher performance and greater flexibility in determining exactly what information the proxies examine. These decisions are made at the granularity of each proxied connection. We describe the firewall design and implementation and report preliminary experimental results using Fast Ethernet.

I. INTRODUCTION

The Advanced Security Proxies (ASP) project at TIS Labs is experimenting with novel firewall architectures and implementations suitable for use in high-performance network environments. Firewall implementations have traditionally been classified as either packet filtering firewalls, in which individual packets are examined and permitted or denied according to policy, or proxy firewalls. In a proxy firewall, the client or process initiating the connection contacts an intermediary, known as a proxy, running on the firewall. After establishing a connection with the proxy, and some additional authentication is performed between the client and the proxy, a second connection is established from the proxy to the server or other remote process listening for a network connection request. Thereafter, the client and server both carry on their normal activities, including sending and receiving application-level protocol messages. However, the messages are relayed through the proxy firewall, which performs all lower-level TCP/IP protocol handling, and delivers the data to the application proxy. The application proxy checks the message, to prevent illegal activity or possible attack, and then forwards the message, once again requiring the firewall to perform all the TCP/IP processing involved in transmitting data.

While proxies allow for the implementation of high assurance firewalls, they have often required that all data flow up and down the protocol stacks, even though much of the data was simply being copied from one side of the firewall to the other side. As network speeds have increased, protocol stacks on general purpose operating systems have not been sufficiently fast to keep pace. As a result, proxy firewalls had become speed bumps on the network. One way of mitigating these performance limitations has been to direct some classes of network traffic through packet filtering software on the firewall hosts. Under proxy control, the packet filters can be programmed to permit packets belonging to approved connections to pass through the firewall. At the completion of a proxied connection, the packet filter rules are dynamically removed, plugging the hole that is temporarily opened to carry the traffic.

In order to support higher speed network technologies with proxy firewalls, we are studying a variety of methods by which a proxy can control the lower-layers of the network hardware or software in order to optimize the flow of data across the firewall. Our research objective is to demonstrate an approach that can keep pace with increasing network speeds without resorting to expensive, overly specialized custom hardware solutions. Our current implementation goal is focused on reaching OC-12 ATM speeds of 622 Mbits/second, which represent a significantly higher performance network than is usually protected using a proxy firewall. Our initial implementation work has utilized 100 Mbit/s FastEthernet, with OC-12 ATM experimentation on the roadmap.

The rest of this paper continues with an overview of the architecture of the ASP firewall. This is followed by sections describing the implementation of the firewall using the Scout operating system, and experimental measurements reporting on the performance of the current prototype. The paper concludes with a review of related work, an outline of our future work and conclusions.
II. ARCHITECTURE

The ASP firewall architecture is designed to provide multiple fast datapaths across the firewall between the protected and unprotected networks. These network services, accessed through well-defined interfaces, are utilized by ASP firewall proxies to accelerate the traffic transiting the network boundary. Because each proxy retains full control over when traffic is switched to a fast datapath, security functions can be applied to the specific application protocol messages that must be inspected and checked against the security policy enforced by the proxy. For example, an FTP proxy might check that each FTP command including a file pathname uses a bounded length to prevent a buffer overrun attack.

In contrast, network traffic which is not subject to examination by the proxy may be directed over a fast datapath, reducing the delay experienced by each individual packet, and reducing the overall load on the firewall host system. For TCP connections, this will often allow greater throughput, while the firewall as a whole will expend more of its computational resources on performing useful proxy activity rather than just copying data bytes from one network protocol stack endpoint to another endpoint.

The architecture envisions multiple fast datapaths, corresponding to distinct implementation layers and degree of processing of network traffic. In an ATM environment, cells on switched virtual circuits (SVCs) might be the unit of traffic to be forwarded by the network plumbing. In more general environments, the IP datagram, with or without defragmentation support, can be forwarded. A TCP connection forwarder would deliver connection establishment and termination notifications to a proxy while immediately forwarding IP packets on an established connection. Finally, the approach is not limited to TCP connection-oriented traffic. Other classes of traffic including UDP based multimedia protocols or media level signaling protocols (such as ATM UNI or ILMI) can also be handled within this framework.

For a number of reasons, the ASP firewall prototype described below is a software-only implementation of the architecture. In addition, we have focused exclusively on TCP-oriented connection traffic and IP-based fast datapaths as the former are the most widely deployed proxy firewalls, and the latter are the most generically supported WAN networks. However, the architecture itself is applicable to a wider range of network technologies and hardware implementations. After describing our implementation and experiments, utilizing the Scout extensible operating system, we also identify requirements for hardware support for the architecture in future work.

III. IMPLEMENTATION

Two separate fast datapath facilities have been implemented in the prototype ASP firewall. Known as IPCUT and NETTAP, both implementations are designed to accelerate the handling of IP packets traversing the firewall. The difference between the two is that IPCUT is designed to carry traffic belonging to a connection that has not been established prior to the proxy issuing commands to IPCUT, while NETTAP is designed to handle traffic belonging to a previously extant connection. In particular NETTAP can switch traffic from the currently proxied connection onto the fast datapath. To evaluate the performance of the firewall acceleration facilities, FTP and HTTP proxies have been implemented that utilize IPCUT and NETTAP respectively. Both proxies and fast datapath mechanisms have been implemented in the Scout extensible operating system.

A. Scout Operating System

The Scout O.S. [1], developed at the University of Arizona, is a modular O.S. directed toward communication-oriented applications. These included thin-client Java stations, file- and web-servers, QoS routers and firewalls. The key abstraction of Scout is the path, which replaces the process as the execution-time entity created to service a user request. Unlike a process, a path is explicitly tied to one or more I/O devices. Its purpose is to handle incoming messages across the set of composable modules that comprise the path. The Scout infrastructure provides general facilities for manipulating messages; scheduling threads associated with each path; creating, managing and destroying paths; and demultiplexing messages from an input device to a path capable of processing it.

Paths are composed of sequences of modules (confusingly called routers by the Scout developers) that process messages as they traverse the path. Many of the modules are designed to process one layer of a network protocol stack, and there are modules for TCP, IP, ARP and Ethernet devices (ETH), as well as specific network devices. A Scout kernel is configured by specifying all the modules that will be present in the system, and the router (module) graph indicating which modules have connecting edges. Each module has a number of interfaces, drawn from a small number of types defined by the Scout infrastructure. Interfaces may only be connected to each other if the are of the same type, reducing, though not eliminating, configuration errors. The workhorse type in the system is NET, corresponding to an abstract network interface over which a message between two layers in a protocol stack may be passed. The usual convention for naming protocol stack interfaces is to refer to them by orientation such as UP or DOWN. Thus the TCP module's DOWN:NET interface.
is connected to the IP module's UP:NET interface in an ordinary protocol stack.

Fig. 1 illustrates a firewall configuration with two protocol stacks. At boot time, each router is initialized, with addresses or routing entries. The Proxy calls pathCreate creating a path from the Proxy to TCP. This passive path waits for a TCP connection request. A software demultiplexing mechanism delivers each incoming packet to the path responsible for processing it. Demultiplexing is accomplished using a distributed approach in which each module attempts to match the fields in its own network layer with one or more possible paths. In the case of TCP, if no established connection exists, the demultiplexing search would first match the incoming packet's IP address with the local host, then the IP packet type (TCP) and TCP port number with the passive path previously created. In response to the connection request, pathCreate is called to construct an active path. Each module along the path initializes per-stage state used by that module in processing packets. TCP control block variables specifying sequence and acknowledgement numbers are stored here, along with congestion control windows and retransmission timers. After the TCP connection closes, a pathDestroy operation frees the stages associated with each module, and also removes the demultiplexing entries for that path.

B. Optimized Firewall Datapaths

Following earlier proxy firewall optimization work, we implemented a pair of identical TCP/IP protocol stacks with one or two modules at the top, inspecting and shuttling application messages from one side of the network to the other. Without any datapath optimization modules, the system performs all the protocol stack processing associated with any other proxy firewall. However, since Scout is a single address space system, data is passed via reference wherever possible. The message handling library takes care to avoid extra byte copying as headers are removed during input processing and latter added during output processing. The modules described below provide fast IP forwarding datapaths.

IPCUT, illustrated in Fig. 1, provides an optimized datapath by connecting via its DOWN interface to two different IP modules. In addition, a command interface is provided and used by the FTP proxy to enable the flow of packets over the fast path. IPCUT connects to IP at the same point where a TCP module would connect. An IPCUT command includes parameters describing a TCP connection, consisting of the source and destination IP addresses and TCP port numbers. Optional arguments may include the selection of different initial sequence numbers on each side of the firewall connection. In response to the command, IPCUT creates a path in two steps. First, pathCreate is called to build a path from IPCUT down one protocol stack, through IP and ETH modules. IPCUT uses a hook in the TCP module to add this path to the TCP demultiplexing map. From the Scout point-of-view, IPCUT represents an optimized form of TCP. More importantly, the demultiplexing tree implementation is deterministic, and the TCP module is configured at compile time to demultiplex all IP packets with protocol field equal to 6, the standard value for TCP. Once the first half of the path is established, a pathExtend call is made to repeat the process on the other protocol stack connected to the opposite network interface. PathExtend is functionally equivalent to pathCreate, except the starting point is the existing path, and new stages are added in turn for each module of the new protocol stack.

Once the optimized IPCUT path is established between the two networks, the FTP proxy proceeds to forward the application level message which triggers the file transfer. FTP uses a separate connection for TCP data, which means the original TCP control connection does not need to be optimized. It remains quiescent while the file data moves over the IPCUT path. After the operation completes, the IPCUT module observes the TCP connection teardown by recording the packets with FIN flags, and the respective acknowledgements of these FIN packets by each side. At that point, IPCUT tears down the optimized datapath using pathDestroy, and the proxy is notified of the change.

NETTAP, illustrated in Fig. 2, provides application-level proxies with several additional approaches to establishing and managing fast datapaths. The NETTAP module is inserted directly between the TCP and IP modules, in a position where it can intercept and redirect the IP traffic under proxy control. When using NETTAP, a typical application will first establish two TCP connections and will initially fully proxy data between these connections. Our HTTP proxy currently makes use of the NETTAP module. HTTP traffic is collected and examined until a complete HTTP GET request has been received. The simple proxy implements a policy that
filters out URLs, protecting some private URLs on the server, while allowing the rest to be accessed across the firewall. Other policies are easily implemented. Once the proxy determines that the GET request is allowed, it forwards it to the web server, and issues a NETTAP JOIN command, resulting in the HTTP response being carried over an optimized datapath. Unlike the IPCUT module, the NETTAP module switches the same TCP connection used to carry the HTTP traffic onto the fast datapath.

NETTAP affords the proxying application with a range of cost-benefit tradeoffs by operating in one of three selectable modes. The first, JOIN, optimizes host TCP and application processing out of the path completely, in an analogous manner to IPCUT. Once the application has switched a connection to this mode, it cannot receive data on, send data to, or control either connection on the path. The second, SNOOP, establishes a fast datapath but also collects and queues payload data, which is delivered to the application as available CPU cycles permit. In this mode, the application may monitor traffic and, if needed, terminate both connections, but cannot guarantee that some hostile traffic has not already been forwarded before the moment it is detected. The third mode, INSPECT, addresses this issue by queuing and delaying the forwarding of traffic until the application has signaled that the collected data may be sent. Of these modes, JOIN has the least impact on system resources, and INSPECT the greatest.

Both IPCUT and NETTAP perform many of the same low-level operations on IP packets. IP addresses and TCP port numbers may be changed to facilitate network address translation and port hiding. In addition, the sequence and acknowledgement fields of the TCP header must be translated between the two halves of the network connection. TCP numbers the bytes transmitted from sender to receiver in order to provide reliable service. Acknowledgements in the TCP header of packets traveling in the opposite direction indicate the greatest sequence number received, and confirm delivery of data to the host. Sequence numbers do not start from any fixed initial value. Instead, each host picks an initial sequence number. In order for a fast datapath to be created, the firewall must modify the sequence and acknowledgement fields by adding or subtracting a constant value corresponding to the difference in the initial sequence numbers. Finally, the IP and TCP checksum fields must be updated to reflect any changes made to the headers. In our implementation, we use an efficient differential checksum algorithm. Instead of examining all the data in the TCP segment to re-compute a fresh checksum, the original checksum is used as a starting point, and is modified based on the old and new values of the header fields that change.

IV. EXPERIMENTS

Initial measurements of firewall performance have been collected using a testbed consisting of 450 Mhz Pentium II PCs with 128 Mbytes of RAM and 100 Mbit/second full-duplex FastEthernet interfaces. A client and server machine are each connected via switches to a firewall machine with two separate network interfaces. Fig. 3 compares the performance of the ASP FTP proxy using IPCUT against a proxy and packet filter firewall. The peak CPU utilization of the ASP firewall was 30%.

Fig. 4 compares the performance of the ASP HTTP proxy using NETTAP JOIN mode against a proxy and packet filtering firewall. For large transfer sizes, the ASP firewall provides nearly the same throughput as the control case with no firewall between the client and web server. The peak CPU utilization of this configuration was 28%, in contrast to 61% with the NETTAP SNOOP mode. Smaller transfer sizes result in performance similar to traditional proxies, as the costs of connection setup and inspecting the HTTP GET messages dominate overall performance in this regime.
V. RELATED WORK

Several other researchers have investigated ways of improving firewall proxy performance by optimizing the underlying network protocol stacks. Spatscheck et. al. [1] describes a TCP connection optimization for Scout paths and a firewall proxy that utilizes it. Maltz and Bhagwat [2] have implemented similar TCP splicing mechanisms in a Unix kernel to support firewall and web caching applications. Fall [3] investigated more general mechanisms for optimizing data copying between both files and network sockets, and introduced a Unix interface to control optimization.

VI. FUTURE DIRECTION AND CONCLUSION

There are many additional features we would like to implement in our ASP firewall prototype. While the current Scout system required a simple change to the TCP module to allow IPCUT to insert paths into the demultiplexing map, a more sophisticated Scout infrastructure would allow IPCUT to lookup and access the demultiplexing map dynamically. This would remove a dependency between the IPCUT module on low-level internals of the TCP module. Also, while the path operations are conceptually very simple, the actual interface in Scout requires developers to directly manipulate the pointers linking the various stages of the path together. This code is difficult to debug, extremely dense, and highly prone to coding errors resulting in deformed paths. An enhanced version of the system should provide a higher-level interface to aid developers in the construction and management of paths.

An additional NETTAP command, SPLIT, is currently in development, which will resume the full proxying of a connection by switching the traffic away from the fast datapath and back to the two distinct TCP/IP protocol stacks below the proxy. SPLIT will provide the ability for proxies to periodically re-inspect a connection. A security policy might require re-examination of a connection after a fixed time period, or after a specified number of bytes or packets had traversed the fast path.

Once the ASP architecture is well established, it should be feasible to implement key pieces in hardware, allowing proxy firewalls to scale with increasing network bandwidth. Many hardware routing devices include the capability of updating the IP checksum, but do not have the capability to update the TCP checksum. This is probably due to the fact that all IP routers must decrement the time-to-live field, requiring changes to the IP checksum, but do not change any TCP fields. One clearly identified hardware requirement to support optimized TCP datapaths in firewalls is a method for differential or constant time TCP checksum recalculation. The demultiplexing of packets to different destinations involves the inspection of both IP and TCP fields. The hardware requirements for firewalls vs. routing are not dramatically different, except that TCP flags are also useful as part of the demultiplexing rules. However, the rate at which new TCP connections are established and destroyed may be much greater than the rate at which routing table changes propagate to forwarding engines in routers. Firewalls require an interface to the demultiplexing mechanisms that supports many more changes per second, and with small, bounded latency. Finally, firewalls need to track and record connection termination. A simple state machine for recording the sequence number of packets with the FIN flag, and generating a notification when the corresponding acknowledgement number is seen would also be a requirement for implementing all of the optimized network plumbing in hardware.

As more proxies are implemented that utilize optimized datapaths, the configuration of firewall security policies becomes more complicated. More investigation of the various ways in which security policies exploit dynamic firewall behavior to improve performance without compromising network boundary integrity is needed.

ACKNOWLEDGEMENTS

John Sebes conceived of the original ASP project goals and architecture, and contributed to its success. Hilarie Orman suggested the use of Scout as a firewall platform.

REFERENCES
