Maintaining High Performance Communication Under Least Privilege Using Dynamic Perimeter Control[⋆]

Paul Z. Kolano

NASA Advanced Supercomputing Division, NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035 U.S.A.
`kolano@nas.nasa.gov`

Abstract. From a security standpoint, it is preferable to implement least privilege network security policies in which only the bare minimum of TCP/UDP ports on internal hosts are accessible from outside the perimeter. Unfortunately, organizations with such policies can no longer communicate using common multiport protocols that require randomly chosen ports for auxiliary connections. This paper introduces a new approach for maintaining such communication under least privilege while achieving maximum performance. By dynamically modifying perimeter ACLs, inbound auxiliary connections are only allowed through the perimeter at exactly the times required. These modifications are made transparently to external users and with minimal changes to internal configuration. A prototype implementation of the **D**ynam**i**c **P**erimeter **E**nfo**r**cement system, called Diaper, has been implemented and tested with several applications.

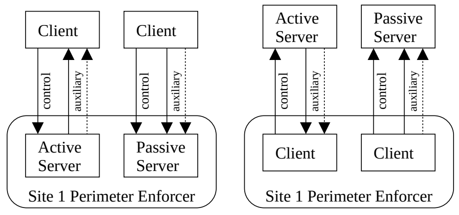
Key words: Firewalls, grids, high performance networking, multiport protocols, network access control, security

1 Introduction

A fundamental dictate of computer security is the *Principle of Least Privilege*, which states that "every program and every user of the system should operate using the least set of privileges necessary to complete the job" [28]. In networks, privilege traditionally corresponds to the set of TCP/UDP ports that are allowed to traverse a perimeter established by some form of *perimeter enforcer* such as a firewall or router/switch with access control lists (ACLs). A typical least privilege network policy might contain the rules (1) allow all outbound traffic to non-blacklisted hosts, (2) allow inbound traffic in direct response to established outbound traffic, (3) allow inbound traffic to a small set of well-known server control ports, and (4) deny all other traffic. In this policy, users on external hosts are limited to the least possible set of privileges necessary to provide some predetermined set of capabilities to them. Namely, they are only allowed to initiate connections to the control ports of designated network services, which are already bound, thus cannot be used for any other purpose. Internal users can generate arbitrary outbound traffic to non-blacklisted hosts and receive inbound responses to that traffic, but internal services they start are not directly accessible from beyond the perimeter.

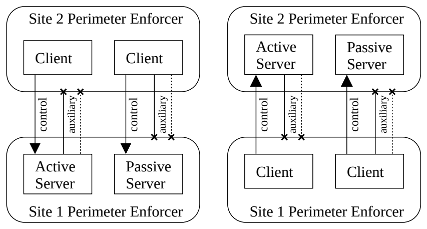
⋆ This work is supported by the NASA Advanced Supercomputing Division under Task Order NNA05AC20T (Contract GS-09F-00282) with Advanced Management Technology Inc.

While such a policy works well for single port protocols such as SSH and HTTP, it breaks down when utilizing multiport protocols. Figure 1 shows the basic multiport protocol models. Each model consists of a client and a server, one of which is inside and one of which is outside the perimeter created by the perimeter enforcer. To request a specific service provided by the server, the client connects to a statically determined *control port* on the server, after which it establishes a set of *auxiliary connections* over dynamically determined ports. Each server is designated as either *active* or *passive*. An active server is one that initiates the auxiliary connections to the client. A passive server is one that listens for the auxiliary connections from the client. While traditionally associated with the FTP protocol, which uses separate control and data channels, these same models are just as applicable to modern protocols for grids, high performance file transfer, voice over IP, multimedia over IP, and other applications.



(a) Internal server (b) External server

Fig. 1. Basic multiport service models



(c) Site 1 server (d) Site 2 server

Fig. 2. Communication breakdown

By default, the least privilege policy above supports the internal active server and external passive server models. Since all inbound auxiliary connections are denied, it does not support the internal passive server or external active server models. Figure 2 shows the breakdown in multiport protocols when two sites both utilize a least privilege policy. As can be seen, it is impossible for the two sites to communicate since the models that are supported by one site are exactly the models not supported by the other.

The easiest solution for many organizations is to permanently open up the perimeter and allow inbound traffic to a subset of unprivileged ports, which can then be used for auxiliary connections. Since these ports are accessible from anywhere, however, it is easier for attackers to hijack or interfere with auxiliary connections. In addition, applications only listen on these ports at specific times related to control port traffic, thus these ports are usually not bound until that time and can be used for other purposes. These may range from running software still under development to running unauthorized services or authorized services with unauthorized versions and/or configurations to Trojans acting without a user's knowledge waiting for malicious connections. The ports associated with these uses are now directly accessible from outside the perimeter and are subject to attack and/or unauthorized utilization.

Several approaches try to resolve this problem without resorting to permanent perimeter openings such as protocol-aware firewalls, routers/switches with network login, specialized proxies, new low-level protocols, etc. These approaches suffer from various drawbacks including the need for changes to client/server software and/or practices, requiring special trust relationships with other entities, substantial degradation of network performance, and inability to handle arbitrary protocols and applications.

This paper presents a new approach for **D**ynam**i**c **P**erimeter **E**nfo**r**cement called Diaper, which provides a general-purpose mechanism for maintaining least privilege network security policies while still supporting full utilization of multiport protocols. Diaper protects a site from unauthorized network flows by dynamically applying ACLs on the perimeter enforcer that temporarily allow specific paths through the perimeter at exactly the times required. The appropriate paths and times are derived by observing system calls from within the services that require them. Since Diaper operates at the system call level, it can guarantee that temporary openings cannot be utilized for other purposes since ports are already bound at the time they are allowed through the perimeter. Perimeter openings are only authorized for a single external host, thus making it more difficult to hijack or interfere with auxiliary connections. Diaper requires no changes to software or practices outside of the perimeter, only minimal changes inside, and can be deployed in configurations varying in size from a single host running a software firewall to an organization running multiple hardware perimeter enforcers. Finally, since Diaper utilizes the ACLs of network devices themselves, it does not degrade network performance allowing protocols to operate at the highest speeds possible.

This paper is organized as follows. Section 2 presents related work. Section 3 describes perimeter observation using system call interposition. Section 4 discusses how the perimeter is opened and closed. Section 5 describes implementation and performance. Finally, Section 6 presents conclusions and future work.

2 Related Work

There are a variety of efforts related to the problem addressed by this paper. Stateful firewalls such as Cisco's IOS Firewall [5] can interpret the control channels of specific protocols to determine when an auxiliary port needs to be opened between a given pair of hosts. Only unencrypted protocols can be supported, however, and such support is generally limited to a small set of standardized protocols. To increase support for new protocols, the NAI Labs Wrappers [7] allow system administrators and even users to add custom proxies into the firewall under the supervision of a system call wrapper that prevents subversion of perimeter policy due to bugs or malicious code. This approach does not help with encrypted control channels, however, and is unsuitable for high performance environments since it is deployed on software firewalls.

In general, even hardware stateful firewalls are unsuitable since they are significantly behind the performance curve of routers and switches. For example, the Juniper Netscreen-5400 is one of the few firewalls on the market with 10 gigabit interfaces [16], but only supports 5 Gb/s per interface. The Force10 P10 intrusion prevention appliance [8] operates at 10 Gb/s line-rate, but has no stateful capabilities. The first 10 Gb/s line-rate router, however, was available from Juniper six years earlier [17]. This performance lag is likely to continue as vendors move to 100 Gb/s and beyond.

Routers and switches often support another approach through network login mechanisms such as Cisco's Lock-and-Key [6], where initially all network traffic is denied until users authenticate themselves, after which a static set of ports is opened up from the originating host to internal resources. This approach supports maximum line-rates, but requires users to perform additional authentication and typically opens more ports than needed. Another built-in option is a virtual private network (VPN) [36], where an external host or network can be granted access to the internal network using an authenticated, encrypted connection. VPNs require either special trust relationships set up between organizations or additional steps performed by the user. In addition, the encryption used to guarantee privacy and integrity also degrades performance. For example, the Juniper Netscreen-5400, which is one of the fastest existing VPNs, is only capable of 2.5 Gb/s over each 10 gigabit interface [16].

SOCKS [21] is a protocol that allows clients to traverse firewalls through the use of a proxy server that relays packets from one side of the firewall to the other. SOCKS requires special software to communicate with SOCKS servers and external users must have knowledge of which sites require SOCKS and which do not, which SOCKS server is responsible for each host of a given site, and which set of authentication credentials must be used to access each SOCKS server. In addition, since the SOCKS server must relay every packet, but is not supported in high speed network devices, SOCKS is unsuitable for high performance environments.

Hole punching [9] is a network address translation (NAT) traversal technique where peers behind different NAT firewalls exchange contact information through a well-known rendezvous server and then use a specific series of outbound messages to open inbound paths through the firewall. Hole punching allows high performance communication, but requires special client software, knowledge of which sites utilize the technique and with which rendezvous servers, and a trust relationship with each server.

Many related projects are motivated by the use of grids across organization firewalls. The Globus grid middleware requires a large number of ports to be left open [38], which creates many difficulties behind restrictive firewalls [1]. Hillier proposes the use of a log reader to wait for a successfully authenticated Globus “ping”, after which it parses the source host and adds an appropriate rule to the firewall to allow access to a statically-defined range of ports from that host [12]. Dyna-Fire [11] is a dynamic firewall service that allows a host to access specific ports after receiving an appropriate *port knocking* sequence (i.e. a pattern of connection attempts to closed ports). Both of these approaches are essentially network login mechanisms with the same disadvantages.

Condor is another grid middleware that requires many port openings [18]. Dynamic Port Forwarding (DPF) [31] allows services on private internal hosts to lease external IP address/port pairs from a NAT firewall, which are sent to external clients for use in direct connections. Firewall requests are not authenticated, however, and the internal host is opened up to all external hosts. Cooperative On-Demand Opening (CODO) [30] adds more restrictive openings and basic authentication to DPF, but only supports basic multiprocess applications and only when they are recompiled with the CODO library. Generic Connection Brokering (GCB) [31] uses an external proxy to relay packets between external clients and internal servers, with drawbacks similar to SOCKS.

Voice over IP (VoIP) and multimedia over IP (MoIP) also use multiport communication models, thus are difficult to deploy behind firewalls [29]. Many proposed solutions involve adding knowledge of related protocols such as the Session Initiation Protocol

(SIP) into the firewall itself [22]. In the Distributed Dynamic Firewall Architecture [27], control channels are observed by protocol-specific proxies, which direct filters to allow and deny traffic as needed. Fung et al. enhanced SOCKS with additional UDP handling to support the Real-Time Streaming Protocol (RTSP) [10]. These solutions are similar to stateful firewalls with similar drawbacks.

An alternative approach is to let applications themselves control firewall behavior as needed. The Firewall Control Protocol (FCP) [20] was proposed as a standard for firewall query and control by applications that was originally motivated by the difficulties of deploying SIP servers behind firewalls. This approach relies on the standardization of FCP and its acceptance and integration by firewall vendors, however, which is a long-term effort. Universal Plug and Play (UPnP) [23] defines a similar capability in its Internet Gateway Device (IGD) specification, which allows clients to control UPnP-enabled gateways to permit inbound network access when needed. The IGD specification does not define any access control mechanisms, however, thus is only suitable for home networks with minimal security requirements.

The multiport problem is an artifact of basic TCP/UDP design, which only allows a single stream of data per port. Several new protocols such as the Stream Control Transmission Protocol (SCTP) [32] have been proposed to enable multiplexing of many streams into one. These protocols are not widely supported, however, thus cannot be used with existing implementations of software and cannot take advantage of existing higher level protocol support in network devices.

3 Perimeter Observation

Diaper is based on the notion of a *pinhole*, which is a dynamic rule that allows TCP/UDP traffic to pass from a specific external host to a specific port on a specific internal host. The basic approach is to open a pinhole exactly when the external host needs to establish an inbound auxiliary connection and to close that pinhole exactly when the external host no longer needs the connection. By using this approach, sites with least privilege policies can still communicate, but users cannot hijack pinholes for their own purposes because the internal host will already be using the associated ports.

A single pinhole corresponds to a TCP/UDP ACL on a perimeter enforcer, which is defined by the 5-tuple of protocol, source IP address, source port, destination IP address, and destination port. Since pinholes only require basic ACLs, which can be done at line-rate on many network devices, the pinhole approach supports performance at the maximum capacity of the network itself. To successfully implement a pinhole approach, however, it is necessary to accurately determine three key pieces of information: which internal ports will be used for auxiliary connections, when these connections are needed, and which external host will initiate them.

3.1 Observer Location

This information may be observed at various locations in the network. The most appealing location is on the perimeter enforcer, which has access to all network traffic passing between client and server. In this case, all setup is self-contained on the device mediating network traffic. Unfortunately, at this location, every protocol must be handled differently, encrypted protocols cannot be supported at all, and hardware enforcers can only support the limited set of unencrypted protocols implemented by the vendor.

The least desirable observer locations are on the external host and in the external client/server. To open pinholes appropriately, users or software on external hosts must be given the authority to do so, which gives them the capability to change the internal site's network security policy, thus does not conform least privilege. In addition, the security mechanisms are no longer transparent since they require the installation of new software or modifications to the invocation of existing software as well as managing these changes across different sites that may require different configurations.

An observer on the internal host, but not in the internal client/server, requires no external changes and has access to information beyond that of a perimeter observer such as logs generated, kernel structures modified, etc. A log reader approach suffers from imprecision due to the lag between when an activity is performed and when it is logged. A kernel observer could provide the required information, but kernel development is difficult and error-prone and would affect every process on the internal host.

The final alternative is an observer in the internal client/server itself, which has access to detailed information about every aspect of program operation including variables, functions, system calls, etc. Modifying client/server source code is undesirable as different implementations use different programming languages, data structures, naming conventions, error conditions, etc. that must all be handled differently. In addition, these modifications must be kept up-to-date with the latest patches and revisions. Although services may have very different implementations, they are all built on top of the same set of standard system calls. Furthermore, with the advent of *system call interposition* [15], system calls can be changed dynamically on a per application basis without changes to existing code. A set of such system call modifications is known as an *interposition agent*. The application of this technique for the observation of perimeter information is described in the next section.

3.2 Diaper Interposition Agent

All TCP and UDP sessions share the same basic flow of Standard C Library system calls that occur between an initiator and a listener. In a TCP session, both parties create a socket using socket(). The listener binds its socket to a local address and port number using bind() and then indicates its willingness to receive connections on the socket using listen(). The initiator can then connect to the listening address using connect(). When the listener is ready to process inbound connections, it accepts one of the waiting connections using accept(). Finally, the two parties communicate using read() and write(), and at the completion of communication, they close their sockets using close(). UDP sessions are similar, but connectionless, thus after bind(), both parties can immediately send and receive datagrams using sendto() and recvfrom(). Equivalent system calls exist in the Windows Sockets API [24], but will not be discussed further.

In the models of Figure 1, clients and servers can be both initiators as well as listeners and have identical network system call behavior after the establishment of the control channel. Thus, the same Diaper interposition agent can be used to intercept the system calls of internal passive servers as well as internal clients used to connect to external active servers. The agent processes the control channel appropriately based on whether the wrapped application first connects outbound like a client or waits for inbound connections like a server. The combined functionality allows the agent to additionally handle mixed-mode applications, such as Iperf [13], that act as both a client and a server with auxiliary connections in both directions.

The main challenge in observing the required information by intercepting system calls is in grouping together separate unrelated system calls to obtain a complete picture of the information. At the point just after a bind(), it is known in both TCP and UDP sessions that a socket will be used to listen for external connections on a specific port number. This is also when it is safe to open a pinhole to that port since after bind(), a port can no longer be used by any other process on that host. The pinhole can be opened from this time up until just before accept() or recvfrom(), neither of which can succeed unless inbound traffic was already allowed to the port. Diaper opens a pinhole immediately after every bind() on an external interface, as shown in Figure 5, except the first bind() in the internal passive server case as that bind() is used to establish the control channel. The intercepted bind() does not return to the caller until the pinhole has been successfully opened or an error occurs.

To open a pinhole for an auxiliary connection requires knowledge of which external host will initiate a connection, which is not known until an accept() or recvfrom() completes on the same port. Thus, this information can only be obtained by associating the auxiliary connection with a previously established control channel connection for which the external host is already known. For clients, it is assumed that the control channel is established during the first outbound TCP/UDP connection from an unconnected state, thus the external address for subsequent auxiliary connections is obtained from the address in the last successful control channel connect() or sendto(). For servers, it is assumed that the first externally bound TCP/UDP port is the control port, thus the external address is obtained from each accept() or recvfrom() on this port.

The association of auxiliary ports with control ports must be handled differently depending on the concurrency model of the client/server. The three major concurrency models [25] are *multiprocess*, where each connection is managed by a separate process, *multithreaded*, where each connection is managed by a separate thread, and *multiplexed*, where all connections are managed by the same process/thread using non-blocking I/O. Association in the multiprocess model is straightforward as each control connection is managed by a different process that spawns its own auxiliary connection processes. In this case, the external address in auxiliary processes is the address of the host connected to the parent control socket. The multithreaded model is similar with each control connection managed by a different thread that spawns its own auxiliary connection threads. In this case, however, multiple threads may share the same memory space, thus care must be taken to store the control thread IP address in a thread-safe location. The other complication is the existence of multiple thread implementations. Diaper currently supports only POSIX threads. The basic flow of system calls in multiprocess and multithreaded servers is shown in Figure 3. The only difference is the use of pthread_create() instead of fork() and pthread_exit() instead of exit() in the multithreaded model.

The multiplexed model is significantly different. In a multiplexed client/server, poll() and/or select() system calls are used to determine which sockets have data waiting, thus preventing the single thread of execution from blocking during accept() or read(). Since poll() and select() may return any number of sockets, a multiplexed client/server may handle any number of arbitrarily ordered control and auxiliary connections in a single round of implementation-dependent processing. Diaper manages this complexity by intercepting poll() and select() and artificially limiting the concurrency during each round to a single socket (with starvation prevention using a round robin approach). Since auxiliary connections will only be created when a need for them arises during control channel processing, the external address used to open an auxiliary pinhole can be obtained from the socket that is currently being processed. The basic flow of system calls in multiplexed servers is shown in Figure 4.

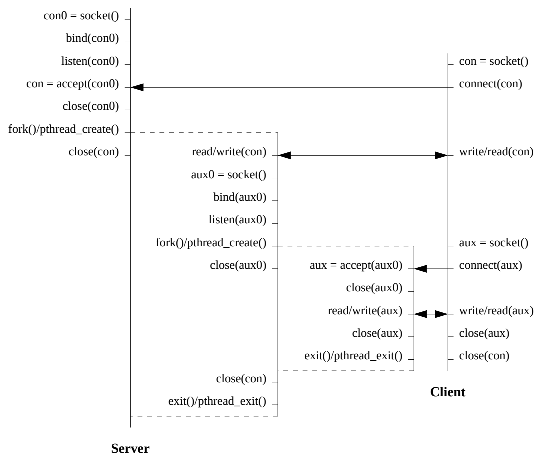


Fig. 3. Multiprocess/multithreaded concurrency

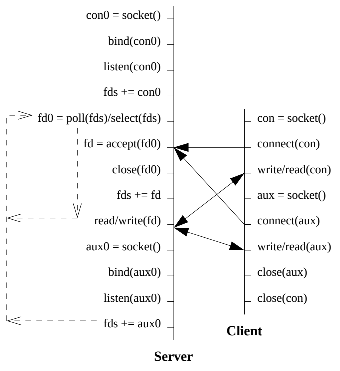


Fig. 4. Multiplexed concurrency

The final piece of information needed by the interposition agent is the point at which each auxiliary connection is no longer needed so that the associated pinhole can be closed. The basic approach is to examine each close() system call to determine if it is closing a socket associated with an open pinhole. Care must be taken to avoid closing pinholes prematurely due to duplication of sockets caused by accept() and fork(). A close() on a listening socket that has already been accepted (i.e. not the accepted socket) does not trigger a pinhole close nor does a close() on a listening socket that has not been accepted, but which has been preceded by a fork(). In the latter case, closing the pinhole becomes the responsibility of the child process. Pinholes are also closed in the same manner within exit() and _exit() system calls and within signal handlers.

4 Perimeter Control

Once all of the required information has been observed by the interposition agent and a pinhole request has been generated, that request must be carried out on the perimeter enforcer. The *perimeter controller* is defined to be the system that interacts with the perimeter enforcer to dynamically open and close pinholes upon request. In order to interact with the perimeter enforcer, the perimeter controller may need to be in special proximity to it. For a hardware device such as a switch or router, the controller may need to be on a host attached to the device's management port. For a software enforcer such as Iptables [14], the controller must usually be on the same host. Since network services will be hosted and invoked on potentially many different systems, pinhole requests generated by an interposition agent must be authenticated and sent to the perimeter controller host. After a request reaches the perimeter controller, it is authorized against a set of allowed perimeter changes. Finally, the request is executed by making the appropriate ACL changes on the perimeter enforcer. Figure 5 shows the components of the perimeter controller and the flow through a multiprocess internal passive server. Diaper was designed as a modular collection of servers with unique functions that can be combined or replaced individually with different implementations as desired.

4.1 Pinhole Authentication

When a pinhole request is generated by an interposition agent, it must be sent to the perimeter controller for execution. Before execution, the perimeter controller must verify that the request came from a legitimate source. The component on the perimeter controller that authenticates requests is called the *remote pinhole authentication server*. With the need to support the external active server model, remote authentication becomes subject to a number of complications. In this model, a client running inside the perimeter must request pinholes for auxiliary connections from the external active server. Clients are executed by normal users, but users must not be able to modify the perimeter policy directly. Instead, requests are carried out indirectly on their behalf by a component on the client host called the *local pinhole authentication server*.

Local Authentication The local authentication server runs with administrator privileges and reads pinhole requests from interposition agents. After a request is received, the local authentication server authenticates to the remote authentication server on the user's behalf, after which the request is passed on for further processing. Users cannot read the remote authentication credentials, thus cannot make direct requests. The local authentication server must also guarantee that only authorized clients in system directories that need pinholes are able to request them and that those clients are not under the control of mechanisms such as LD_PRELOAD or ptrace().

To achieve these goals, authorized clients are wrapped with a simple setuid program. The owner of the wrapper may be any valid user recognized by the local authentication server and is designated as the *local delegate* for that particular client. The wrapper first opens a temporary file descriptor used to communicate with the local authentication server. It then clears unsafe environment variables such as LD_PRELOAD and LD_LIBRARY_PATH and preloads the interposition agent. Finally, the wrapper permanently drops all delegate privileges and executes the original command with the original arguments. The FD_CLOEXEC flag is immediately set on the file descriptor by the agent to prevent abuse by any local shell escapes, etc. that a client may implement. User-level mechanisms for dynamically modifying application behavior are disabled by the kernel since the wrapper is a setuid program.

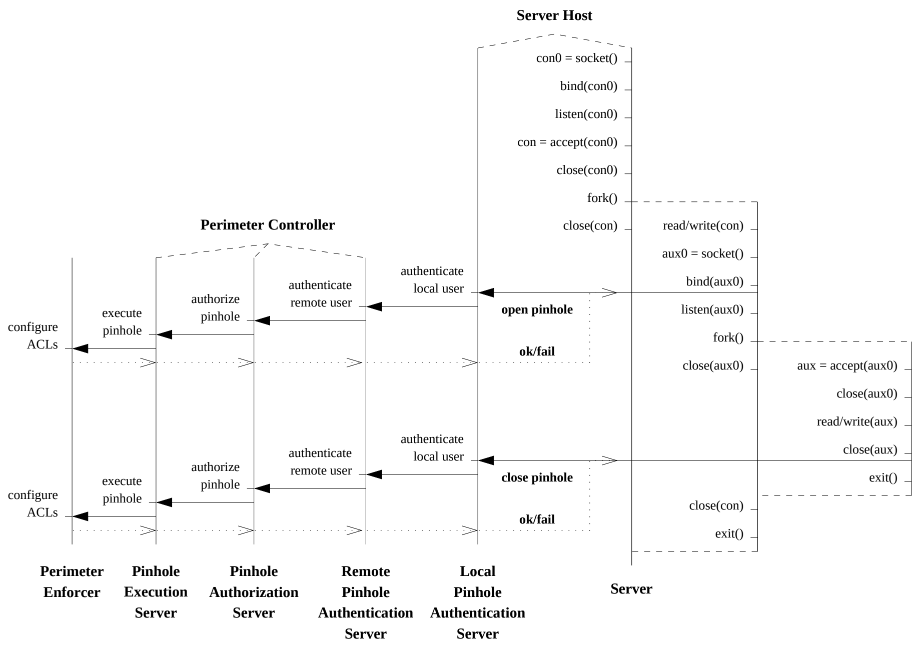


Fig. 5. Diaper event flow in multiprocess internal passive server

When a client's interposition agent needs to make a pinhole request, it searches for an open file descriptor not owned by itself. The ability to write to this descriptor provides assurance that the client has been executed by the setuid wrapper. A second descriptor is opened by the client itself based on a name provided by the local authentication server to obtain the invoking user's identity and prevent replay attacks.

Note that the internal passive server model works similarly, but since servers typically run with elevated privileges to service multiple users, a setuid wrapper is not required. Instead, the interposition agent running in the server intercepts the setuid(), seteuid(), and setreuid() system calls. At this point, before dropping privileges, the appropriate file descriptor is created, after which the same authentication scheme is used.

Remote Authentication Remote authentication ensures that only legitimate users can access the perimeter controller in order to issue pinhole requests. The actual mechanism used to enforce this policy between the local and remote authentication servers can take any form. In the Diaper implementation, a stock SSH server is used with an extremely restrictive login shell called Mash [19] that does not allow remote delegates to do anything besides issue pinhole requests via a command created for that purpose. Each local authentication server defines a mapping from each of its local delegates to a *remote delegate* known to the remote authentication server for which it possesses authentication credentials. These credentials are then used to transmit locally authenticated requests to the remote authentication server for further processing.

An undesirable, but unavoidable risk of allowing non-interactive requests to the perimeter controller is the need for the remote delegate credentials to be unprotected beyond standard file system discretionary access controls. For instance, an SSH private key cannot be encrypted or else the local authentication server cannot use it to authenticate to the remote authentication server. Thus, a root compromise of a local authentication server host allows the attacker to issue pinhole requests. To mediate this risk, authenticated pinhole requests are first checked against a site security policy before being carried out on the perimeter enforcer.

4.2 Pinhole Authorization

The component responsible for validating authenticated pinhole requests against a site security policy is called the *pinhole authorization server*. A site policy is defined by a set of rules of the form "(allow|deny) <remote delegate> <end user> (tcp|udp) <source IP address range> <source port range> <destination IP address range> <destination port range>". A pinhole is permitted if it allowed by at least one rule in the policy and is not denied by any rule in the policy. Fairly restrictive policies can be imposed by employing service or host specific remote delegates. For example, host specific remote delegates can be used to minimize the damage of local authentication server compromises by using a remote delegate with the same name as each host "host_i" and a rule "allow host_i * * * * host_i *". In this setup, hosts can only open pinholes to themselves, thus a breach of one host does not affect the security of all the others.

4.3 Pinhole Execution

Once the authorization server determines that a pinhole request is permitted, that request must actually be carried out on the perimeter enforcer. Since many different requests may be received around the same time, access to the perimeter enforcer must be strictly controlled to avoid interference between requests. The required mutual exclusion and perimeter enforcer interaction is provided by the *pinhole execution server*. This server only accepts requests from root-level processes, such as the authorization server. Requests are processed in batches by a single process as will be described in Section 5.1.

Each pinhole request is translated into an ACL update command on the perimeter enforcer, which varies by the enforcer's type and vendor. Some products have APIs or SNMP-based mechanisms for manipulating the running configuration. Those without such support require scripting of the command-line interface. Diaper currently supports one software enforcer (Iptables) and four classes of hardware enforcers (Cisco IOS devices, Force10 FTOS devices, Foundry IronWare OS devices, and Juniper JunOS devices). Additional enforcer types can be added in a modular fashion.

The pinhole execution server also detects and cleans up stale pinholes, which are those that are no longer needed by any process, but which are still active on the perimeter enforcer. This can occur when a process receives a SIGKILL before it is able to clean up its own pinholes or after a crash of the pinhole execution server itself. A related condition is when the ACL state of the perimeter enforcer does not match that of the pinhole execution server as could potentially occur after a reboot of the perimeter enforcer or a manual update by an administrator. Both of these cases are handled by a periodic status check of the perimeter enforcer's ACLs. Stale pinholes are detected using ACL accounting functionality that counts how many packets have matched a given ACL. A pinhole is considered stale and will be closed if its packet count does not change between two consecutive status calls. Using this approach, Diaper's pinhole state is resilient even across failures and restarts of multiple components.

5 Implementation and Performance

A prototype of Diaper has been fully implemented. The local pinhole authentication server, pinhole authorization server, and pinhole execution server are written in Perl. The remote pinhole authentication server is a stock SSH server. The interposition agent is written in Bypass [33], which is a minimal syntactic wrapper around C/C++ code that isolates the user from differences in system call interfaces and implementations between Unix operating systems. Bypass supports layering of multiple agents, thus using the Diaper agent does not preclude using other agents for other purposes.

Diaper has been fully tested on Linux, but can run on any Unix operating system with a library preload mechanism. The agent is compiled into a shared library, which is loaded into applications by setting the appropriate preload environment variable (e.g. LD_PRELOAD, _RLD_LIST, etc.) before the application is executed. With the exception of POSIX thread support, all of the intercepted system calls are a core part of the Standard C Library, which is linked into almost every application on Unix operating systems, thus Diaper is likely to work correctly with the vast majority of dynamically linked multiport applications without modification. The same approach can also be used on Windows systems by utilizing an equivalent mechanism such as Fault Tolerant Interposition Agents [3]. The Diaper interposition agent is around 1000 lines of Bypass C++ code. The setuid wrapper is about 50 lines of C code.

The test network consisted of an internal host, an external host, and a perimeter controller, each on a 2.4 GHz Pentium 4 Linux box connected by 100 Mb/s Ethernet through three types of perimeter enforcer: Iptables, a Cisco 6500, and a Force10 E600. The least privilege policy of Section 1 was applied to each perimeter enforcer. Raw performance numbers were also gathered for a Foundry MLX-4 and a Juniper MX960.

5.1 Scalability

The main bottleneck in the Diaper architecture is the perimeter enforcer. While the various Diaper servers can be scaled using standard server load balancing techniques, the perimeter enforcer is a unique resource that must be involved in every interaction. The two main scalability measures of a perimeter enforcer are the maximum number of ACLs that can be in effect at any given time without appreciable performance degradation and the maximum rate at which ACLs can be updated.

Hardware enforcers typically have no performance degradation regardless of the number of ACLs in effect due to special Ternary Content Addressable Memory (TCAM) that can perform simultaneous line-rate lookups across the entire ACL space. TCAM is expensive, however, thus, depending on the vendor, is usually limited to somewhere between thousands and tens of thousands of ACL entries per interface. Software enforcers have access to large amounts of cheap memory, thus can theoretically support very large numbers of ACLs. Since standard memory is not optimized for large scale parallel lookups, however, performance degrades significantly as more and more ACLs are applied. Thus, the practical limit for these enforcers is usually between thousands and tens of thousands of ACLs total.

The other limiting factor is the maximum rate at which ACLs can be updated. Figures 6 and 7 show the time required to apply ACLs sequentially and in batches using Iptables, a Cisco 6500, a Force10 E600, a Foundry MLX-4, and a Juniper MX960. As can be seen, significant performance gains can be achieved by batching ACL updates together. Figure 8 shows the update rates achieved with different batch sizes when no ACLs are initially in effect. The optimal batch size is approximately 200 for Iptables, 300 for the Cisco 6500, 1000 for the Force10 E600, 5000 for the Foundry MLX-4, and 4200 for the Juniper MX960, achieving effective rates of 10000, 1220, 445, 1070, and 446 updates per second, respectively. The pinhole execution server processes ACLs in batches of the optimal size when at least that many requests are queued, thereby maximizing the update rate. As shown in Figure 9, the maximum achievable rate can only be maintained up to a certain number of existing ACLs before factors such as exhaustion of the TCAM or kernel caches are encountered. The update performance of the Force10 E600 and the Foundry MLX-4 degrades far less dramatically than the others as the number of ACLs in effect increases. The Juniper MX960 has several orders of magnitude more ACL capacity than the others, but is hampered by a slow ACL compilation process. The Foundry MLX-4 has the best overall ACL performance with a high maximum update rate and almost no degradation as existing ACLs increase.

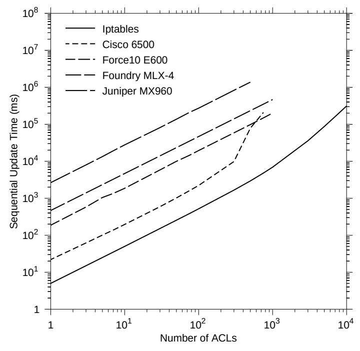


Fig. 6. Sequential update times

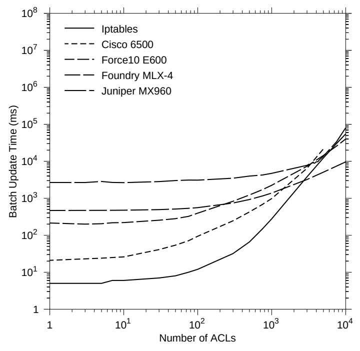


Fig. 7. Batch update times

To determine the suitability of Diaper to a particular organization, the ACL characteristics of that organization's perimeter enforcer must be compared to its expected network traffic patterns. Namely, the expected average number of concurrent multiport protocol connections must be less than the ACL capacity of the perimeter enforcer and the initiation/termination rate of those protocols must be less than the maximum update rate achievable with a number of ACLs in effect equal to the average number of connections. To handle bursty traffic where the average number of connections is less than the ACL capacity of the perimeter enforcer, but the maximum number of connections is sometimes greater, the pinhole execution server keeps track of ACL usage and buffers requests until the load returns to normal. If the ACL characteristics of the primary perimeter enforcer are not adequate for the expected traffic, there are still many possible deployment options due to the flexibility of the Diaper architecture.

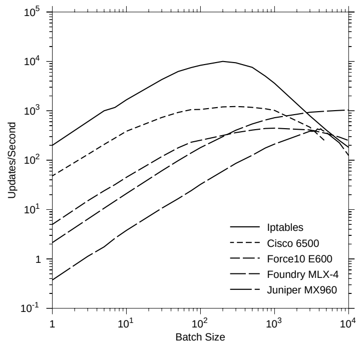


Fig. 8. Batch update rates

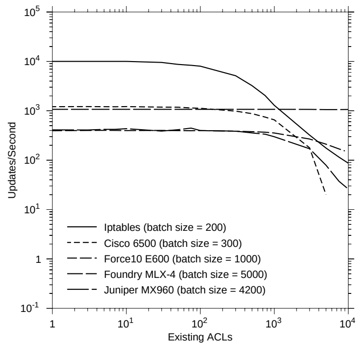


Fig. 9. Update rate degradation

First, Diaper can be deployed on a per application basis, thus the protocol load may easily be shared with other approaches. For example, a network login capability can be used to offload portions of the traffic such as long running MoIP sessions while Diaper can dynamically control the remaining traffic. Second, Diaper is designed to control both firewalls as well as core routers and switches with no modifications necessary beyond the perimeter. By deploying multiple perimeter controllers each in charge of a different network device, the natural segmentation of network traffic provided by internal switches and routers can be used to combine the ACL capacities of multiple devices. A similar approach can be used with a sequence of perimeter enforcers on the border. Finally, Diaper is lightweight enough to be deployed on every host in the network that runs its own software enforcer. In this case, the perimeter enforcer, perimeter controller, and internal host are all one and the same. The organization perimeter enforcer can statically allow some subset of traffic to pass through to the end hosts, which themselves can dynamically control which connections they will and will not allow.

As a sanity check for the scalability results, publicly available packet traces of FTP connections to Lawrence Berkeley National Laboratory [26] were analyzed to determine the requirements for a real organization. The traces represent over 22,000 FTP control connections and over 49,000 data connections consisting of more than 3,200,000 packets between 320 unique servers and 5832 unique clients over a 10 day period. Figure 10 shows the number of open FTP data connections over time, which represents the maximum number of perimeter enforcer ACLs required at any given time. Figure 11 shows the ACL updates per second required to open/close the corresponding pinholes. As can be seen, although the traces encompass one of the most prevalent multiport protocols and a fairly large number of hosts and connections, the ACL usage

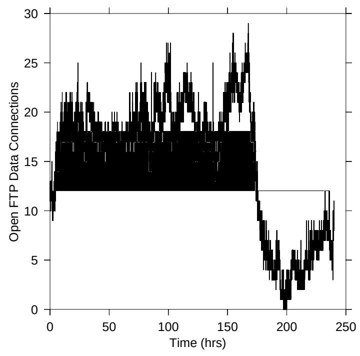


Fig. 10. Open data connections

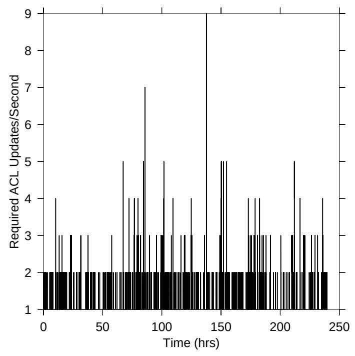


Fig. 11. Required update rate

and update rate requirements are very modest and could easily be handled by any of the perimeter enforcers tested. Additional study is needed, however, to assess whether such requirements are typical of all organizations and multiport protocols.

5.2 High Performance File Transfer

Diaper was tested with a variety of high performance file transfer protocols that represent a wide cross-section of existing multiport protocol behavior including TCP and UDP control streams, TCP and UDP data streams, encrypted control streams, multiple data streams, and internal passive/external active server models with multiprocess, multiplexed, and multithreaded concurrency. The applications used for testing include BbFTP [2], BetaFTPD [4], Tsunami [34], UDT [35], Vsftpd (SSL mode) [37], and Wzdftpd [39]. None of the corresponding protocols besides unencrypted FTP (i.e. BetaFTPD and Wzdftpd) are supported by existing firewalls due to either their use of encryption on the control channel or their nonstandard research-oriented nature.

Table 1 shows the overhead in milliseconds introduced by Diaper while transferring a 100 MB file through three types of perimeter enforcer using each of the applications. The overhead was measured against the same transfers through statically authorized ports without the agent wrapper. In these tests, overhead was proportional to the number of ACL updates divided by the ACL update speed of the given perimeter enforcer plus a slight overhead of around 15 ms per update. No benefit was gained from batching in the multiple stream BbFTP case as it binds its auxiliary ports sequentially. Overall, Diaper operated correctly with a variety of protocols with minimal overhead.

6 Conclusions and Future Work

This paper has described a new approach for enabling least privilege network security policies based on **D**yn**a**m**i**c **P**erimeter **E**nfo**r**cement called Diaper. Diaper observes the behavior of network services to identify the specific inbound perimeter access that is required at any given time and dynamically adjusts the ACLs of a perimeter enforcer to open and close the perimeter accordingly. It supports inbound access for both clients and servers and is completely transparent to external users. Internal services must be invoked slightly differently, but no source code modifications nor changes to user usage patterns are required. Through the use of the Diaper framework, each site can have the tightest perimeter policy possible and yet still communicate at the highest bandwidth with almost any multiport application.

Application	Server Model	Control	Data	Concurrency	Iptables	Cisco 6500	Force10 E600
BbFTP (1 stream)	Internal Passive	TCP	TCP	Multiprocess	30.1	61.6	397
BbFTP (2 stream)	Internal Passive	TCP	TCP	Multiprocess	68.4	137	755
BbFTP (4 stream)	Internal Passive	TCP	TCP	Multiprocess	149	293	1570
BbFTP (8 stream)	Internal Passive	TCP	TCP	Multiprocess	400	682	3410
BetaFTPD	Internal Passive	TCP	TCP	Multiplexed	29.8	64.1	373
Tsunami	External Active	TCP	UDP	Multiprocess	29.9	68.3	406
UDT	External Active	UDP	UDP	Multiprocess	29.5	62.9	370
Vsftpd (SSL mode)	Internal Passive	TCP	TCP	Multiprocess	29.4	66.9	406
Wzdftpd	Internal Passive	TCP	TCP	Multithreaded	30.8	68.9	426

Table 1. Diaper overhead (ms) during 100 MB file transfer

There are a variety of directions for future research. The ACL characteristics of additional perimeter enforcers will be evaluated and corresponding support added to the pinhole execution server. Scalability analysis will be performed on additional multiport protocols when corresponding packet traces become available. Support for NAT environments and a Windows interposition agent will also be investigated. Alternatives to library preloading will be studied to enable support of static binaries. For deployment in real-world security settings, mechanisms for redundancy and resiliency must be added such as automatic fail-over based on factors including the health of the Diaper servers and the perimeter controller's connectivity to the perimeter enforcer. Finally, additional pinhole authorizations can be added including time-based permissions and dynamic permissions on the pinhole execution server that can, for example, limit the number of pinholes that any one user can have open at once.

References

1. Baker, M., Ong, H., Smith, G.: A Report on Experiences Operating the Globus Toolkit Through a Firewall. Sep. 2001.
2. BbFTP. `http://doc.in2p3.fr/bbftp`.
3. Benso, A., Chiusano, S., Prinetto, P.: A COTS Wrapping Toolkit for Fault Tolerant Applications Under Windows NT. 6th IEEE Intl. On-Line Testing Wkshp., Jul. 2000.
4. BetaFTPD. `http://betaftpd.sourceforge.net`.
5. Cisco Systems, Inc.: Cisco IOS Firewall Design Guide. Jan. 2006.
6. Cisco Systems, Inc.: Lock-and-Key: Dynamic Access Lists. Jan. 2005.
7. Epstein, J., Thomas, L., Monteith, E.: Using Operating System Wrappers to Increase the Resiliency of Commercial Firewalls. 16th Annual Computer Security Appl. Conf., Dec. 2000.
8. Force10 Networks, Inc.: Force10 Networks Introduces the Industry's First Line-Rate 10 Gigabit Intrusion Prevention System to Secure High Perf. Networks. Press release, Apr. 2006.

9. Ford, B., Srisuresh, P., Kegel, D.: Peer-to-Peer Communication Across Network Address Translators. USENIX Annual Tech. Conf., Apr. 2005.
10. Fung, K.P., Chang, R.K.C.: A Transport-Level Proxy for Secure Multimedia Streams. IEEE Internet Computing, vol. 4, num. 6, 2000.
11. Green, M.L., Gallo, S.M., Miller, R.: Grid-Enabled Virtual Organization Based Dynamic Firewall. 5th IEEE/ACM Intl. Wkshp. on Grid Computing, Nov. 2004.
12. Hillier, J.: A "Dynamic" Firewall. UK GRID Firewall Wkshp., Dec. 2002.
13. Iperf. `http://dast.nlanr.net/Projects/Iperf`.
14. Iptables. `http://www.netfilter.org`.
15. Jones, M.B.: Interposition Agents: Transparently Interposing User Code at the System Interface. 14th ACM Symp. on Operating System Principles, Dec. 1993.
16. Juniper Networks, Inc.: Juniper Networks Enhances and Extends High-End Security Portfolio. Press release, Aug. 2005.
17. Juniper Networks, Inc.: Juniper Networks Ships Full-Performance M160 Router with OC-192c/STM-64 Interfaces. Press release, Mar. 2000.
18. Kewley, J.: Using Condor Effectively in the Presence of Personal Firewalls. Oct. 2004.
19. Kolano, P.Z.: Mesh: Secure, Lightweight Grid Middleware Using Existing SSH Infrastructure. 12th ACM Symp. on Access Control Models and Technologies, Jun. 2007.
20. Kuthan, J.: Internet Telephony Traversal Across Decomposed Firewalls and NATs. 2nd IP Telephony Wkshp., Apr. 2001.
21. Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., Jones, L.: SOCKS Protocol Version 5. IETF Request for Comments 1928, Mar. 1996.
22. Martin, C., Johnston, A.: SIP Through NAT Enabled Firewall Call Flows. IETF Internet Draft, Aug. 2001.
23. Microsoft Corporation: Understanding Universal Plug and Play. Jun. 2000.
24. Microsoft Corporation: Windows Sockets 2. Mar. 2005.
25. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An Efficient and Portable Web Server. USENIX Annual Tech. Conf., Jun. 1999.
26. Pang, R., Paxson, V.: A High-level Programming Environment for Packet Trace Anonymization and Transformation. 2003 ACM SIGCOMM Conf., Aug. 2003.
27. Roedig, U., Ackermann, R., Rensing, C., Steinmetz, R.: A Distributed Firewall for Multimedia Applications. Wkshp. "Sicherheit in Netzen und Medienstromen", Sep. 2000.
28. Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. Proc. of the IEEE, vol. 63, num. 9, 1975.
29. Shore, M.: H.323 and Firewalls: Problem Statement and Solution Framework. IETF Internet Draft, Feb. 2000.
30. Son. S., Allcock, B., Livny, M.: CODO: Firewall Traversal by Cooperative On-Demand Opening. 14th IEEE Intl. Symp. on High Performance Distributed Computing, Jul. 2005.
31. Son, S., Livny, M.: Recovering Internet Symmetry in Distributed Computing. 3rd Intl. Symp. on Cluster Computing and the Grid, May 2003.
32. Stewart, R., Xie, Q. et al.: Stream Control Transmission Protocol. IETF Request for Comments 2960, Oct. 2000.
33. Thain, D., Livny, M.: Multiple Bypass: Interposition Agents for Distributed Computing. J. Cluster Computing, vol. 4, num. 1, 2001.
34. Tsunami. `http://anml.iu.edu/projects.html`.
35. UDT. `http://sourceforge.net/projects/dataspace`.
36. Venkateswaran, R.: Virtual Private Networks. IEEE Potentials, vol. 20, num. 1, 2001.
37. Vsftpd. `http://vsftpd.beasts.org`.
38. Welch, V.: Globus Toolkit Firewall Requirements. Oct. 2006.
39. Wzdftpd. `http://www.wzdftpd.net`.