Towards a Deep-Packet-Filter Toolkit for Securing Legacy Resources

James Deverick and Phil Kearns – The College of William and Mary

ABSTRACT

Users of a network system often require access to legacy resources. Providing this access is a difficult task for system administrators because the access protocols for those resources are typically insecure. A common approach is to develop a custom wrapper or proxy that securely processes user requests before forwarding them to the legacy server. The problem with this approach is that administrators must develop a custom solution for every resource. We believe that there are common requirements for managing these resources that can be addressed from a more centralized model. The userspace queuing extensions of the Netfilter firewall modules provide a generic environment in which protocol-aware deep packet filters can be constructed to enhance the security of resource access protocols. We employ this environment to strengthen two commonly used legacy protocols, and compare their requirements. We show that it is possible to secure legacy resources with minimal degradation in performance. We also discuss considerations for development of a deep packet filter toolkit to aid system administrators in securely managing legacy network resources.

Introduction

It has been our experience that, for a variety of reasons, production networks typically house several legacy resources. It may be the case that an application critical to the users of a network is no longer under active development, and no adequate substitute has been found. If a newer solution is available, other factors such as deployment cost or compatibility concerns may prevent its installation. The result is that system administrators must provide access to resources that employ inherently insecure protocols, a concept in conflict with their responsibility to maintain the security of the network and its resources.

By developing a way to manage these resources securely, administrators can continue to provide the needed services without endangering other components of the system or risking the integrity of the resources themselves. Currently, the only way to achieve this is to develop a custom environment in which to ‘‘wrap’’ each resource. No toolkit or framework exists that allows administrators to secure existing protocols with minimal effort. We believe that despite the arbitrary complexity of a protocol, certain common requirements exist for administrators that allow the development of such a toolkit.

As the basis for this toolkit, we employ the userspace queuing extensions of the Netfilter firewall [17] distributed with the Linux operating system. The processing capabilities provided by the userspace extensions allow us to collect any information deemed appropriate for a resource request before determining if the traffic should be allowed to contact the server. We can delay, alter, or reroute packets, interactively challenge authenticity, collect system status information, or anything else dictated by the protocol and administrative requirements of the resource at hand.

To illustrate both the power and flexibility of this approach, we consider two common protocols and the weaknesses they contain. In both cases, newer versions or extensions exist that address those problems, but we assume that some other consideration such as cost or compatibility prevents them from being deployed. This provides an example of what we believe to be a common issue in typical network system environments. Specifically, we discuss the lpr line printer protocol as defined by RFC 1179 [9] and the Network File System (NFS) protocol, version 3 [2]. Both of these systems provide essential resources to users, yet neither provides any strong authentication of those users.

Using the Netfilter userspace queuing extensions, we develop strong user authentication for each of these protocols. In the case of lpr, we employ public-key cryptography to ensure that users can only submit print jobs under their own identities. This prevents users from constructing raw print job control files in order to print under a false identity, possibly bypassing quota enforcement. Our technique also directly authenticates users, allowing administrators to open the print server to outside domains, such as wireless networks, providing users the ability to print securely from their laptops. For the NFS example, we use symmetric-key challenge-response authentication to secure filesystem mount requests. In order to mount a secured filesystem, a client must be in possession of the current key and know the format of the challenge issued by the firewall. This prevents unauthorized client machines from mounting remote filesystems simply by pretending to be authorized hosts.

Both of our extensions significantly improve the security of the original protocols, while introducing minimal overhead. They demonstrate how two vastly different protocols can be secured by the same basic technique,

and suggest a set of tools that could be used by administrators to secure legacy applications and protocols with significantly less effort than is currently required.

Design Goals

Faced with the problems of securing and managing legacy resources in a network, we define the following requirements for a useful solution:

- Neither client nor server components of any application should require modification to take advantage of the features we provide. Many existing approaches, such as application proxies, are not always feasible since they require application code to be rewritten or expanded.
- The solution must be generic enough to support any application with minimal overhead. Each application will impose different requirements on the solution. It should allow administrators to develop complex management systems without developing entire filtering applications from scratch.
- The system must be able to adapt to changing conditions in the environment, adjusting its filtering behavior accordingly. For example, depending upon application requirements, traffic from authenticated clients might not need to be filtered for a given period of time.

General Related Work

Because we deal with protocol-specific traffic filtering, the most relevant existing work relates to *deep packet filtering* [7]. Several authors have acknowledged that the basic address/protocol-based filtering provided by traditional firewalls is not sufficient for many applications. In cases where filtering requirements are tied directly to a given protocol, stock firewalls are not appropriate. Instead, a filter that examines protocol-specific information contained in the packet payloads provides the required service. Because deep packet filtering incurs a higher cost than traditional header-based filtering, much of the research attention has been paid to improving performance of these filters, even via hardware solutions [4, 6, 18]. Very recently, it has been suggested [5] that forthcoming firewalls must include some level of deep packet filtering capability, due to the increasing number of application-level attacks. We agree, and the toolkit we develop is an essential step toward integrating these protocol-aware filtering capabilities in commonly existing software-based firewalls.

Implementations

Here we describe two prototypical applications chosen to demonstrate the capabilities of our technique. Each is exemplary of different components of the system interacting with different protocols. Both of them were implemented on a six machine testbed in a secure laboratory [12]. Figure 1 illustrates the testbed layout. The firewall separates two private subnets. On one subnet resides three test clients that interact with a server residing on the other subnet. They are connected by a path that travels through the firewall. This is designed to emulate a typical setup in which a perimeter firewall protects a server or set of servers from client traffic, whether it originates from an internal network or an outside internet. We also place a client-class machine on the server subnet to provide control data for performance analysis. Since it is on the same subnet, the control client need not traverse the firewall to interact with the server. The machine is composed of the same hardware that constitutes the other three clients. By running identical client-server applications on the control client and the firewalled client, we can derive the overhead imposed by our toolkit. Note that a second path exists through which traffic could possibly travel from clients to servers, but this connection is not a part of the testbed. It is used for administrative purposes in the lab environment.

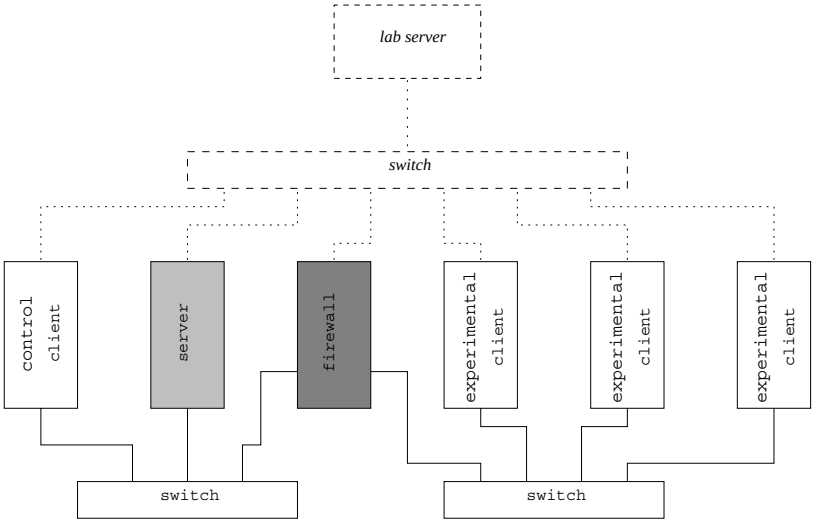

Figure 1: Prototype testbed.

iptables has a modular extension interface that allows developers to introduce additional features. One module included in the distribution introduces a QUEUE target that stores packets in a kernel data structure until they are retrieved by a userspace daemon. This daemon can then analyze the packet and its contents, and determine an appropriate course of action using the resources provided to userspace programs. Figure 2 illustrates the components of a firewall system that uses this approach. Since it allows us to implement more detailed packet analysis than the built-in features of iptables, we employ this module in the implementation of our deep packet filter daemons.

NFS Prototype Implementation

The Network File System [2] allows filesystems on a sever to be exported over a network to multiple clients. Clients can mount the filesystem, which then appears as part of the local directory structure; network operations are transparent to the user as he interacts with the remote files.

Since the filesystems being exported to remote clients may contain sensitive information, we must ensure that only authorized clients are able to mount them. The stock implementations of NFS provide simple host-based authentication by comparing the IP address of a client from which a mount request originates against a table of pre-authorized client addresses. To further increase security, some environments employ client MAC address filtering at the network switch level.

Even in this case, however, a malicious user can gain unauthorized access to the filesystem. He needs only to set manually the MAC address for his network interface, configure his client with an authorized IP address, and unplug a legitimate system. He can then connect to the switch, replacing the connection of the system whose MAC and IP addresses are being spoofed. By taking these steps, the user presents a client to the system that, in most configurations of the currently prevalent protocols, is indistinguishable from a valid, authorized client. These considerations dictate the need for stronger authentication of clients wishing to mount remote filesystems.

Related Work

Despite the presence of new, more secure, file sharing paradigms [13, 3], traditional NFS is still widely used. Accordingly, some attempts have been made to strengthen authentication in the NFS protocol itself. Ashley, et al. [1] introduced role based access control to the protocol. O'Shanahan [15] replaced the UID-based authentication in NFS with a public-key cryptosystem. A less intrusive modification was proposed by Goh, et al. [8]. In their system, the client intercepts all filesystem operations and encrypts them on the fly so that they are secure when stored on the server. The underlying NFS structure isn't changed,

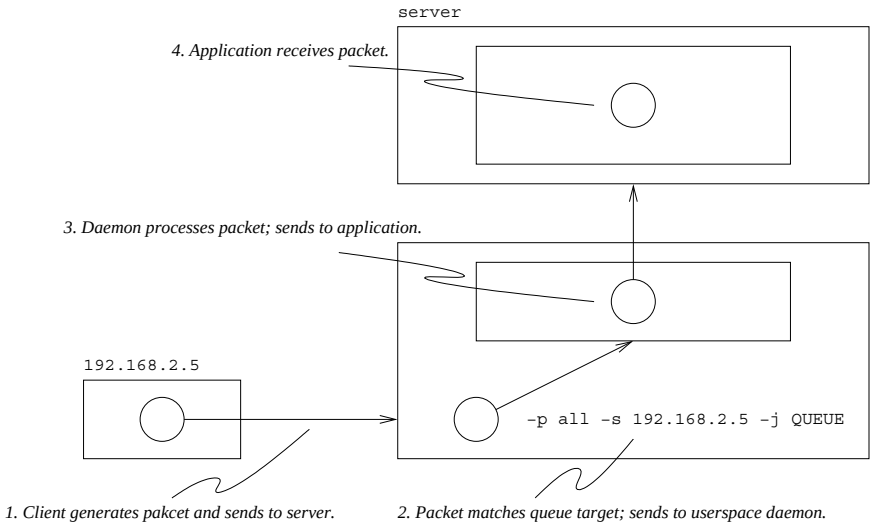

Figure 2: Packet filtering with userspace queues.

```
IPTABLES -A FORWARD -p TCP -s $CLNTNET --dport sunrpc --tcp-flags syn syn -j ACCEPT
IPTABLES -A FORWARD -p TCP -s $CLNTNET --dport nfs --syn -j ACCEPT
IPTABLES -A FORWARD -p UDP -s $CLNTNET --dport nfs -j ACCEPT
IPTABLES -A FORWARD -p UDP -s $SERVER --sport nfs -j ACCEPT

IPTABLES -A FORWARD -p TCP -s $SERVER --tcp-flags ack ack -j QUEUE
IPTABLES -A FORWARD -p TCP -s $CLNTNET --dport sunrpc --tcp-flags ack ack -j QUEUE
IPTABLES -A FORWARD -p TCP -s $CLNTNET --dport sunrpc --tcp-flags urg urg -j QUEUE
IPTABLES -A FORWARD -p TCP -s $CLNTNET --dport sunrpc --tcp-flags fin fin -j ACCEPT
IPTABLES -A FORWARD -p UDP -s $CLNTNET --dport sunrpc -j QUEUE
```

Figure 3: Initial ruleset for authenticated NFS.

but the data being stored within that structure are modified transparently to the user. Our approach is different in that no components of the system require modification of any kind. Server and client daemon behavior is unchanged, as are the data and metadata stored in the remote filesystem. The only component we introduce is a client-side authentication daemon that responds to challenges issued from the firewall during mount attempts. Assuming that clients are not able to bypass the mount process, we gain a more secure NFS environment. If, however, a client successfully guesses a remote filehandle, it can modify the associated file by manually constructing NFS packets and interacting directly with the server daemon. Our approach, like any system that does not authenticate every client-server interaction, cannot prevent this form of compromise.

Connection Life Cycle

In order to challenge traffic destined for NFS mount daemons, we must first learn the port at which those daemons are running. Since the portmapper query for this information will always be destined for the sunrpc port of the server, we can easily intercept this traffic, remember that a client has requested this port information, then allow the request to proceed.

Dynamic Rules

Our obligation to intercept any portmapper request defines the initial iptables ruleset, illustrated in Figure 3.

When the server responds, we also intercept that information before forwarding it to the client, and extract from the reply the ports on which client's mount request will be made. We then inject a new rule into the firewall that QUEUEs the client's forthcoming mount request to the authentication daemon. The system can handle requests from multiple clients simultaneously, provided that the Netlink queue buffers do not overflow. If this occurs, packets are dropped as though the firewall discarded them, or network congestion occurred.

Authentication Mechanism

We include an encrypted challenge-response scheme to authenticate remote clients. When a client issues a mount request, it is intercepted by the userspace daemon on the firewall and queued in a data structure to be either accepted or dropped depending on the result of the authentication process. The filtering daemon maintains a state variable for each packet that records that packet's sequential arrival order at the firewall. Since the value of this variable is effectively unique for each packet over the lifetime of the system, we can use it to key a data structure that records all unanswered challenges. Our prototype uses a linked list to store this information, allowing multiple simultaneous client connections. Figure 4 illustrates the structure of a single list node.

```
struct list_element
{
    ipq_packet_msg_t *packet;
    long packet_id;
    long nonce;
    struct list_element *prev;
    struct list_element *next;
};
```

Figure 4: NFS packet queuing structure.

The userspace daemon issues a challenge to the client consisting of a symmetrically encrypted id-nonce pair, and waits in the background for additional traffic to arrive. This traffic may be another mount request that triggers the same process, lengthening the queue, or it may be a client's response, determining the fate of the mount request. The daemon can apply

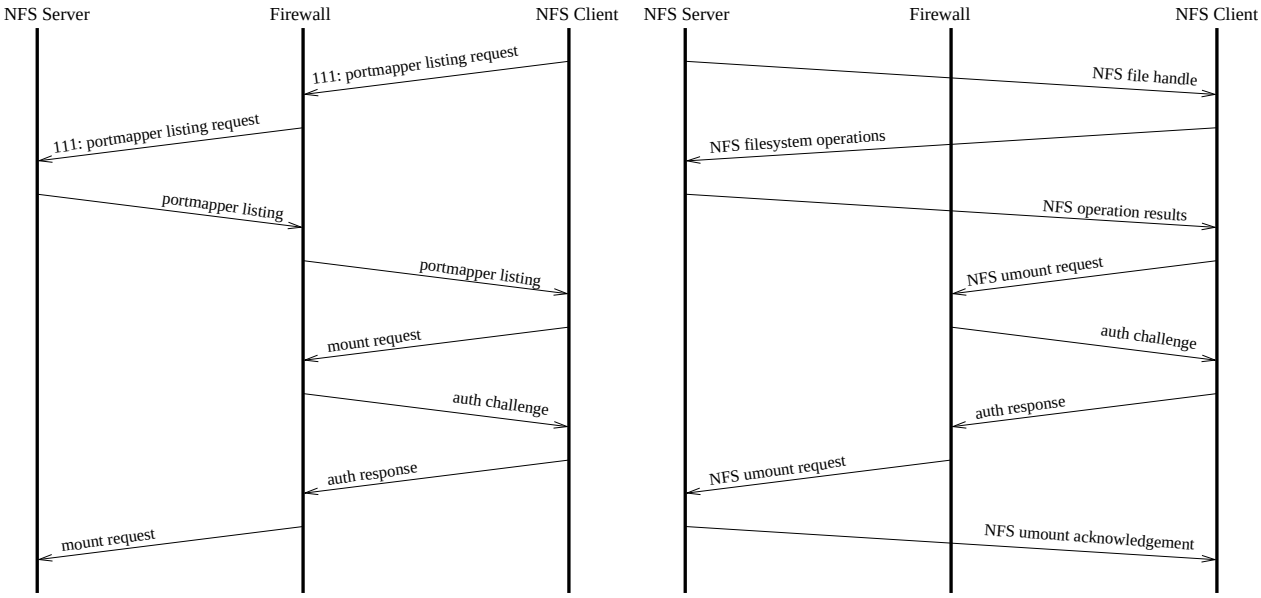

Figure 5: Authenticated NFS service.

to a packet the same primitive filtering targets as the kernel firewall. A library interface provided by the queueing extension module described earlier provides these targets, allowing the daemon to ACCEPT or DROP a packet. If the client's response is satisfactory, an ACCEPT verdict is issued on the original packet, and the mount request is processed by the server. If not, then the packet is dropped, and the server never sees the request. With encryption keys pre-loaded on systems and restricted to root access, a malicious user must successfully root-compromise a legitimate client to subvert the protocol. Spoofing network addresses of any type is no longer sufficient to gain access to the remote filesystem.

Assuming that authentication succeeds, the firewall simply forwards all other NFS traffic between these two systems; no further intervention is imposed until the session ends. Since unmount requests are also processed by mountd, an authentication challenge will be issued at this stage as well. This is a desirable behavior, as it prevents an unauthorized user from unmounting a filesystem from beneath a legitimate user. Figure 5 illustrates the message passing in the authenticated system.

Host Failures

In the event that the server crashes, the client loses connectivity with the remote file system, but only while the server is down. Since authentication only happens during mounts, a reboot of the server simply delays NFS calls until the reboot is complete. The client does not have to remount the filesystem unless something catastrophic enough to change the filehandle on the server side has occurred.

Should the client crash, all mount information is lost during the reboot. the client will need to remount the remote filesystem, requiring that the authentication process be repeated. The firewall and server have no way of knowing that the client crashed, and assuming that the mount request is legitimate would open the system to attack.

Performance

To measure the performance overhead associated with the system, we run benchmarks from two client machines and determine achieved bandwidth in the filesystem. One client mounts the NFS share through the firewall, while the control client resides on the same subnet and has a locally switched connection to the server. Figure 1 illustrates the distinction.

Measurements were taken using the IOZone Filesystem benchmark suite [10]. First, a 32 MB file is written to the NFS share. The suite records the amount of time required to write the file and calculates the achieved bandwidth. A similar approach is used to measure reading performance. We present benchmark results for sequential and random writing and reading of records ranging from 2KB to 1024KB in a randomly generated 32 MB file. Data are presented within 95% confidence intervals generated from 30 independent measurements of the system's performance.

The benchmark suite allows us to compensate for many caching effects. We employ some of these features to present clearer results. Between every test, the filesystem is unmounted; this clears the filesystem buffer cache, and forces the next test to interact directly with the filesystem instead of a faster cached copy. We also employ a setting that flushes the processor caches between each test to ensure that no data are cached locally during read tests.

It is important to note that the benchmarks measure achieved bandwidth in the filesystem only after it is mounted. Time required to mount and unmount the filesystem is not included the results. This means that the authentication mechanisms used must be timed separately. What we measure here is how much the presence of the firewall between the client and server slows down normal operations.

First, we consider the slowdown imposed by the firewall on sequential read and write operations. Figures 6 and 7 illustrate the performance difference between the firewalled and control clients. In both cases, the slowdown is likely to be less than 1% for any record size. We consider this a negligible amount of overhead.

Next, we examine the random read and write cases. Figure 8 illustrates another negligible slowdown. In this case, the worst slowdown is just under 3%, with most cases falling well below 1%. The only case in which we see a significant slowdown imposed by the firewall is randomly reading records from the file, as illustrated in Figure 9. Here, the extreme case imposes a slowdown of about 11%, with most cases incurring a slowdown near 6%. Given that only one type of operation incurs a significant slowdown, and that slowdown, on average, is only about 6%, we argue that the overhead imposed by the system is small enough to justify the additional security afforded by our approach.

The time required to mount the remote filesystem, including the encrypted authentication exchange, was 268±11 µS on the firewalled client. The control client, which has a direct connection to the server and does not perform an authentication exchange, mounted the filesystem in 16±0.2 µS. Relative to the performance of the control client, the firewall introduces significant overhead in mount the remote filesystem. Note, however, that this overhead is encountered only when the filesystem is mounted, and the real time required to mount the directory is still extremely small. Amortized over the lifetime of the directory mount, the overhead is insignificant.

LPR Prototype Implementation

RFC 1179 [9] defines the widely used lpr printing protocol. It contains no support for secure authentication of users. In environments where printing is monitored

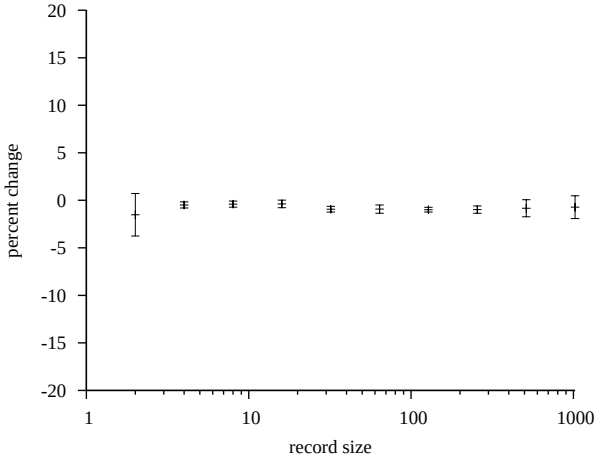

Figure 6: Firewall performance relative to control client. Sequential read of 32MB file. 95% confidence intervals.

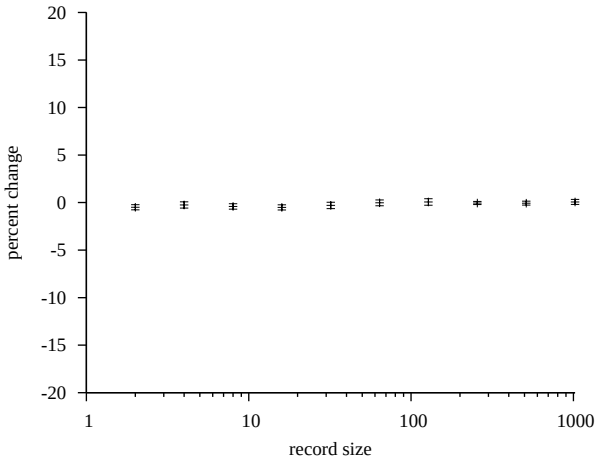

Figure 7: Firewall performance relative to control client. Sequential write of 32MB file. 95% confidence intervals.

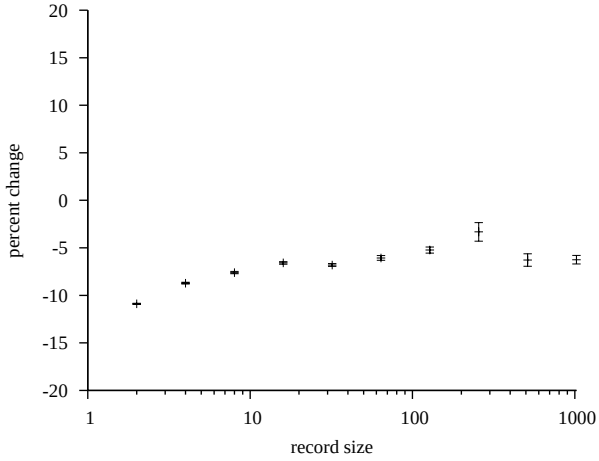

Figure 8: Firewall performance relative to control client. Random read of 32MB file. 95% confidence intervals.

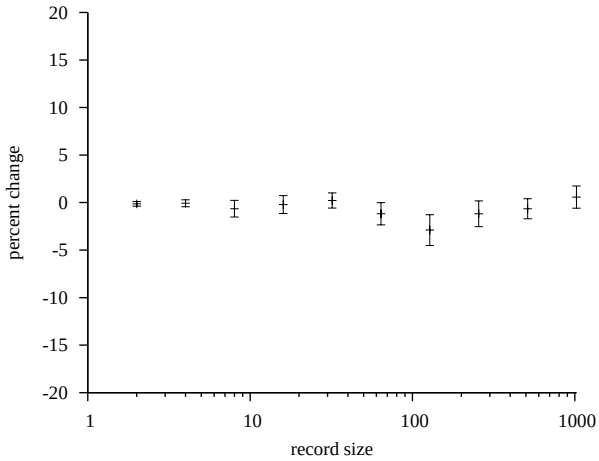

Figure 9: Firewall performance relative to control client. Random write of 32MB file. 95% confidence intervals.

monitored or under quota enforcement, this issue is significant. It means that a user can easily submit printing jobs under another user's ID, thereby bypassing the cost associated with his job request. This same issue complicates the matter of allowing authorized users to print without restriction based on network topology. Ideally, an associate of a department should be able to print to that department's printers whether using his desktop workstation or his laptop from home. Since, however, the protocol doesn't support strong authentication of users, allowing such open access would certainly result in numerous unauthorized printing requests.

Related Work

Many approaches exist to solve this problem. LPRng [16] introduced a new implementation of the protocol that supports the RFC 1179 interface while providing additional functionality such as Kerberos authentication [14] and public-key cryptography support. Most secure printing protocols that conform to the lpr specifications are extensions of LPRng. While this approach does solve the authentication problems we outline, LPRng is only compatible with UNIX-like operating environments. Windows clients, while capable of printing to RFC 1179 lpr servers, do not understand the extensions in LPRng. Accordingly, they cannot take advantage of the authentication attributes provided by the implementation. We offer an enhancement to the standard RFC 1179 protocol that requires no changes on either the client or server side, yet still introduces a strong authentication scheme.

A common approach to strengthening the security of a legacy protocol is to tunnel all of its traffic through an encrypted channel, usually with SSH. This does provide stronger security, but the degree of improvement is highly application specific. In this case, it only partially authenticates the user. An SSH tunnel authenticates the connection, but not the use of

the protocol at hand. Suppose a legacy line printer system is protected by an SSH tunnel, through which end users must forward jobs.

The result is that only authorized users will be able to submit jobs, but there is no enforcement that those jobs be submitted in the correct name. A user may have an account on the system, allowing her to establish an SSH tunnel to the print server, but nothing stops her from submitting the job under false identification (as described below) once that tunnel is in place. A similar property holds true for any encryption or tunneling scheme, including virtual private networks or IPSec tunnels. If the underlying protocol doesn't directly support user-specific strong authentication, then secure channels do not provide robust authentication in typical production environments.

RFC 1179 requires that clients submit a print job control file to the server in order to process a new job. Contained in this control file is the username of the entity submitting the job. The problem is that this field in the control file is presented in cleartext. Nothing prevents a malicious user from altering the contents of this file before the job is submitted, thereby tricking the server into printing the job under someone else's userid. The scheme that we offer forces the user to demonstrate with very high probability that he or she is, in fact, the user designated in the control file for the job being submitted.

Simply put, we intercept the TCP syn packet associated with the connection to lpd, the print server process, and delay it at the firewall until we receive a digitally signed message from the client informing us of the owner of the incoming job. Once this information is stored, the TCP connection is allowed to proceed, and when the control file begins to arrive at the firewall, the daemon extracts the job user information from the file and compares it to the signed username previously received. If they match, then we know that the user for this job is authentic, so we allow all other traffic on this connection through. If not, then we immediately tear down the connection. The server will abort the job, and the client's only option is to restart the job submission process.

User Authentication in Advance

The initial ruleset for our approach is as illustrated in Figure 10.

This small ruleset simply propagates all lpd-bound packets to our userspace daemon for processing, including the TCP packets used to construct the connection over which the job transfer will take place.

A client sends a print job to the print server in the standard fashion. Between the client and the print server is the firewall, listening for connection attempts destined for the print server's spooling ports. When a TCP connection request is made, the firewall intercepts

```
IPTABLES -A FORWARD -p TCP -d $SERVER --dport printer -j QUEUE
IPTABLES -A FORWARD -p TCP -s $SERVER -j ACCEPT
```

Figure 10: Initial ruleset for authenticated remote printing.

```
struct client_element
{
    // together these should indicate a TCP connection
    __u32                   sourceip;         //source address of client
    __u16                   sourceport;       //origin port on client
    ipq_packet_msg_t        *syn;             //syn packet for connection
                                              //will challenge this
    // timeouts are needed to allow reuse of ports
    struct timeval          lasttime;         //last activity time
    // authentication attributes
    long                    nonce, id;        //to prevent replay attacks
    unsigned char           user[32];         //username from challenge
                                              //control file must match
    int                     checked;
    // RFC 1179 defines the command protocol
    unsigned char           current_command;  //last level 1 cmd rec'd
    // control file is subcommand 02 after "receive job"
    unsigned char           current_subcommand; //last level 2 cmd rec'd
    long                    ctl_length, ctl_consumed;     //ctl file length
    long                    data_length, data_consumed;   //data file length
    int                     fin;
    // list operators
    struct client_element   *prev, *next;
};
```

Figure 11: lpr packet queuing structure.

the syn packet and stores it in a data structure until it can confirm the identity of the client. Figure 11 illustrates the data structure used to store client information.

After storing the TCP syn packet, the firewall issues a challenge to the client machine. A specialized daemon on the client listens for and responds to these challenges. The format of the challenge is simple; it consists of a packet id and a random nonce to avoid replay attacks. When the client-side daemon receives this challenge it constructs a response consisting of the same packet id, the incremented nonce, the userid of the owner of the daemon (presumably the person submitting the job), and the digital signature for the message as associated with the given userid. Figure 12 illustrates the response issued from the client upon receiving a challenge from the firewall.

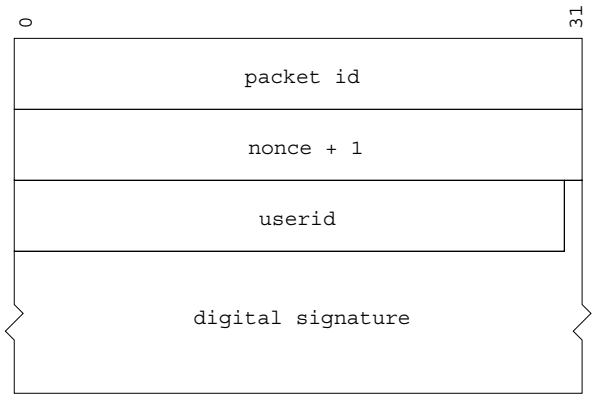

Figure 12: Structure of the printing challenge response.

When the firewall daemon receives the client's response, it first verifies that the signature on the response is valid. To accomplish this step, enterprise environments will require the employment of a scalable public key infrastructure. The details of of such a system are beyond the scope of our discussion. The prototype we present simply assumes that the public keys for legitimate users are located in a directory to which the firewall daemon has read access. The daemon searches that directory for a key with the name <userid>.pem, where <userid> is extracted from the cleartext portion of the client's response. If the signature is not valid for the message, then we do not allow the TCP connection request to reach the server at all.

To enhance the performance of the system and reduce unnecessary network traffic, we do not free the allocated data structures for the connection right away. Because the daemon must only assume a strict conformity to RFC 1179, we do not know what to expect from the client in response to the initial connection packet being dropped. LPRng version 3.8.19 will issue three connection attempts, each timing out at 10 seconds, before giving up on the connection. Alternate implementations of the protocol may use different approaches. Some may even depend solely on the reliability components of TCP to establish the connection. Once the data structures are de-allocated, syn packet retransmissions from the client will be seen by the firewall as new connection attempts, resulting in a repetition of the authentication process. We avoid this by leaving the data structures in place long enough to ensure that the syn packets are re-transmissions and not new connection requests; having this information allows us to silently discard those packets without consuming additional bandwidth and processor time. In the prototype implementation, this timeout variable is static. For a production release, we would allow a configurable runtime option to set this variable, since the administrator would have the context information needed to optimize it.

At this point, the firewall has created an instance of the data structure illustrated in Figure 11 and loaded it with with client IP address/port number combination and the verified username of the entity submitting the request. By indexing the data structure on host IP/port pairs, we can handle inbound packets from a number of different connections simultaneously.

Administrative Jobs

The LPR protocol allows the superuser significantly more control over job submission and administration than regular users. For example, an administrative user can issue or remove a print job with a manually specified username as the owner of that request. The benefit of this is that system administrators can delete pending print jobs through the LPR interface without having to assume the identity of the actual owner or manually modify the state of the printing system.

This means that for connections submitted by the root user of a client machine, the embedded user information may not match the credentials presented during authentication of the connection. Suppose that Mary needs to print a large document, and the department in which she works has a specially designated printer for large jobs. She sends the job to the print server without specifying the correct printer, and the job lands in the queue for the default printer, which can't correctly handle the large job. The system administrator notices the problematic job, and sends an lprm command for that job to the print server as the root user on Mary's machine. The server deletes the pending job, and Mary can be informed that her job was canceled and she should submit it to the correct printer.

If we absolutely require that requests be submitted in the name of the user whose credentials were presented during authentication, we break this component of the protocol. Accordingly, we must make a special case for jobs submitted from root users. If the challenge response packet contains root as the userid, and is properly signed, then we can make no assumptions about what user information the actual job request will contain. With a valid signature on the

challenge response, we know that the client must be an authorized administrative user. Therefore, we place no restrictions on what can be done over the lifetime of that connection.

lpr Job Interception

Once the authentication process is complete (and assuming the authenticated user is not root), the userspace daemon still intercepts the traffic, because it must confirm that job is actually submitted as the user who authenticated with the firewall. It observes the stream as it passes through, watching for the octet that indicates the user field of the control file. The control file consists of commands, subcommands, and data, exactly one combination of which indicates the user for the current job. Specifically, we let everything through until we reach command 2 (receive job), subcommand 2 (receive control file), at which point we begin parsing the control file.

Each line in the control file has a specific format: some character indicates what data will appear on the line, then the rest of the line is that data. For our purposes, we skip lines in the control file until we see one that beings with the character 0x0A, which indicates the ascii username of the owner of the job being submitted. If this user matches the user that previously authenticated, we forward all remaining traffic to the server. Because there may be additional packets already queued in the Netlink buffers, we leave the userspace data structures in place and set a flag indicating that everything else on that connection should immediately be forwarded.

To enhance performance, we also inject two rules into the kernel firewall tables. One forwards everything except the TCP fin packet for the connection directly to the server. By bypassing the userspace propagation for the remaining packets, we significantly reduce the amortized cost of the authentication process. The second rule matches the TCP fin packet from the client, indicating the end of the connection. This packet is forwarded to the userspace daemon, which de-allocates the data structures for this client, then forwards the final packet to the server.

If the user in the control file does not match the authenticated user, no further data will be sent to the print server, and the connection between the server and client will be aborted. Since the job is not actually processed until all of the control file is received, the document will never print.

Authentication Failure and Cleanup

When the firewall denies a print job that has failed the authentication process, the server already has begun receiving the print job. Once the job is denied, these connections must be destroyed so that the server can process additional jobs in its queue. If we do nothing, eventually, both the server and client will timeout and the job will be aborted. The problem with this approach is that in the meantime, jobs accumulate in the queue and are delayed while the server waits for additional data that will never arrive. A better approach is to actively destroy the connection, resulting in an immediate abortion of the job on the server side. We employ the techniques of Lowth [11] to accomplish this. Excerpts from his TCP/IP connection cutter software, available under GNU Public License, satisfy our requirements that both ends of the connection be shut down upon seeing an invalid user for a print job.

Performance Analysis

We measure the impact the filter has on two aspects of the printing system: connections and bandwidth. We present these data separately because there is a significant amount of initial overhead on a per-connection basis due to the asymmetric encryption used for advance user authentication. Once that is complete, the filter imposes minimal overhead on data transmission. Measurements were conducted from both firewalled and control clients, as in the NFS benchmarks.

To calculate job submission times, we modified the source code of the print server daemon to record the arrival time of the first and last packets on the connection, and present the difference in a log file. The same file was printed from both the firewalled client and the control client in these measurements. Submitting a 17 byte text file from the control client took $47{,}719 \pm 3{,}569$ µS. As expected, the firewalled client achieved a slightly slower rate of transmission, requiring $202{,}420 \pm 21{,}660$ µS to complete.

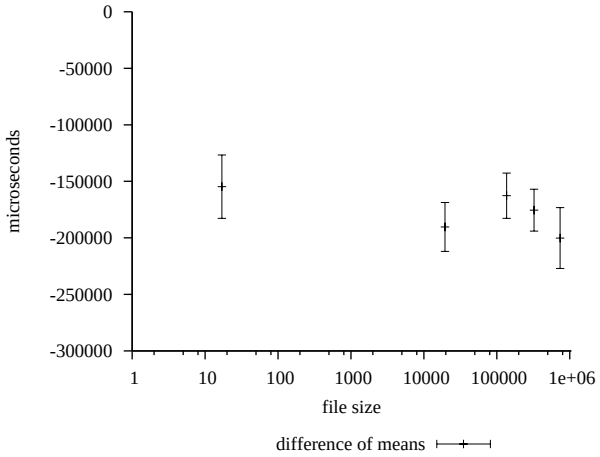

Figure 13: Firewall performance relative to control client. Receipt of control and data files at print server. 95% confidence intervals.

Unlike connection times, which experience a small, static delay with respect to file size, we see in Figure 13 the impact of channelling each packet through the firewall. As files grow in size, more packets are required to contain them; each of these packets must traverse the firewall, experiencing a small delay that is cumulative for submitted job.

Future Work

Having constructed prototypes and described potential uses of our system, we now outline specific tasks that we intend to pursue in the next stages of the project. We present our envisioned final product, complete with additional examples of applications that can take advantage of the services it provides. We are currently developing a firewall management toolkit for the Netfilter system that uses the userspace queuing extensions to allow more active participation from the firewall in the system at large. Figure 14 illustrates the proposed basic structure of the system. The toolkit will include several built-in functions to achieve tasks that we believe will be common in applications. Some functions will be statically implemented; others will be implemented via a pluggable module interface. This allows implementations to be swapped out as appropriate for specific applications.

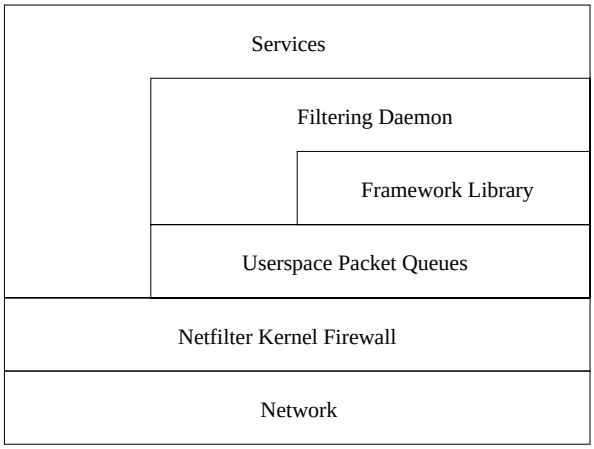

Figure 14: Basic toolkit structure.

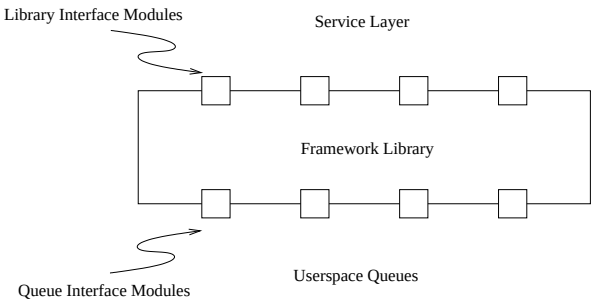

Figure 15: Toolkit modules.

Illustrated in Figure 15, we plan to include a module interface at both ends of the toolkit in the interest of keeping the solution generic. The manner in which the system interacts with filtering daemons is defined by the library interface modules. Similarly, the system's interaction with the Netfilter queuing system is defined by the queue-level modules, though the variability in the latter should be less prominent given the restrictive interface of the queuing system.

Note that services need not be placed behind protections provided by the firewall toolkit. They still have access to basic services provided by the underlying Netfilter system, should the administrators deem it appropriate. Our proposed toolkit is an extension of the firewall model, not a replacement for it. The administrator can construct rulesets that direct traffic for only some services through the system. Other services can use traditional protections, since only those rules that specify the QUEUE target will be propagated to the toolkit.

Planned Toolkit Modules

The prototypes we have implemented dictate a set of useful modules for the toolkit. They provide basic functionality that we believe will be common in several filtering applications. Here, we outline several of the planned toolkit modules. The complete list of included modules will be determined as we construct the system and determine common needs among potential applications. A systems administrator can implement and supply additional modules as deemed necessary for the desired application. This shields application development from cryptographic implementations and allows the administrator to focus on more abstract functions appropriate to the application at hand.

Cryptographic Support for Authentication

Since nearly every application we have proposed relies heavily on authentication and authorization, we must provide direct support for those features in our toolkit. Different forms of authentication require different types of cryptography, so the interface should be generic and configurable, providing support for arbitrary cryptographic modules.

Common paradigms such as RSA digital signatures and fast Blowfish encryption will be included toolkit modules in the system we build. Instead of relying on high-level generic interfaces provided by the standard cryptographic libraries, the administrator will have access to a suite of cryptographic functions tailored to packet-level encryption and challenge-response authentication schemes.

Connection Destruction

In most cases where an authentication challenge fails, we envision the need to abort any pending connections to the server. In the NFS case, this is trivial, since authentication failure is detectable before a client actually connects to the server, and subsequent application traffic is sent over the connectionless UDP protocol. A failed authentication simply means that the firewall blocks all application traffic from the suspicious client, and none reaches the server.

The lpr example demonstrates a more difficult case, however, where authentication cannot be detected until *after* the server connection is in place. Here, we must forcefully terminate the open connection to prevent the

client from sending any additional data to the server. The prototype includes code to accomplish this based on [11], but it adds unnecessary complexity to the filtering daemon. Such a common function is exemplary of the modules our toolkit will include.

Ruleset Modifications

Both of the prototypes we preset modify the kernel firewall's ruleset over the lifetime of the system. The NFS example injects a rule to QUEUE incoming mount requests after it learns the port location of the file server's mount daemon. The LPR prototype injects a rule to accept all remaining traffic on a connection if authentication succeeds. The first case is functional in nature; the second is a performance enhancement.

Both implementations achieve this via invocations of the userspace iptables utility. Since that utility interacts with the kernel firewall through system calls that modify network sockets, we can achieve the same functionality directly. We will include a module that provides a library interface to the underlying firewall ruleset, allowing direct modification of the rules in response to changing system conditions. This will significantly reduce the overhead required to provide a dynamic ruleset.

Implementations and Evaluations

With a toolkit in place, we intend to revisit the two prototypes presented earlier, and re-implement them using the firewall toolkit. This will allow us to offer a comparison of brute force implementations with toolkit assisted constructions. We can compare the ease and efficiency of implementations, and analyze the differences in performance overhead obtained in each method.

Because we are dealing with a real-time environment, performance is of paramount concern. We must take steps to ensure that our toolkit introduces minimal overhead. In order to achieve this, careful measurements must be taken on each module that we introduce.

Conclusions

We have presented two prototypical applications that illustrate how the userspace queuing extensions of a commonly available firewall can be used to secure legacy protocols. The performance imposition of our system is minimal, and certainly justified in light of the increased security and functionality the system can introduce. We have discussed how this approach can be extended into a toolkit that system administrators can use to secure additional legacy protocols and introduce additional arbitrary functionality at the firewall. We believe that the toolkit we discuss will allow administrators to balance more easily the need to provide existing services to users with the need to maintain security features in a networked environment. Finally, we have outlined a larger, more general application to which our toolkit will be useful, demonstrating the flexibility of our approach.

Author Biographies

James Deverick is currently a Ph.D. candidate and systems administrator at The College of William and Mary's Department of Computer Science. His research focuses on building active firewalls that enhance the security of legacy systems. Reach him at jwdeve@cs.wm.edu .

Phil Kearns is an Associate Professor of Computer Science at the College of William and Mary. His research interests lie in the general area of computer systems.

Bibliography

[1] Ashley, Paul, Bradley Broom, and Mark Vandenwauver, "An implementation of a secure version of NFS including rbac," *Australian Computer Journal*, Vol. 31, Num. 2, 1999.

[2] Callaghan, Brent, Brian Pawlowski, and Peter Staubach, "NFS version 3 protocol specification," *RFC 1813*, Internet Engineering Task Force, 1995.

[3] Cattaneo, Giuseppe, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano, 'The design and implementation of a transparent cryptographic filesystem for UNIX," *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX, 2001.

[4] Cho, Young H. and William H. Mangione-Smith, "Specialized hardware for deep network packet filtering," *12th International Conference on Field-Programmable Logic and Applications*, ACM, 2002.

[5] Cho, Young H. and William H. Mangione-Smith, "Deep packet filtering with dedicated logic and read only memories," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2004.

[6] Dharmapurikar, Sarang, Praveen Krishnamurthy, Todd Sproull, and John Lockwood, "Deep packet inspection using parallel bloom filters," *Proceedings of the 11th Symposium on High Performance Interconnects*. IEEE, 2003.

[7] Dubrawsky, I., *Firewall evolution – deep packet inspection*, http://online.securityfocus.com/infocus/1716, 2003.

[8] Goh, Eu-Jin, Hovav Shacham, Nagendra Modadugu, and Dan Boneh, "SiRiUS: Securing Remote Untrusted Storage," *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, Internet Society (ISOC), pp. 131-145, February, 2003.

[9] McLaughlin III, Leo J., "Line printer daemon protocol," *RFC 1179*, Internet Engineering Task Force, 1990.

[10] IOzone, *IOzone filesystem benchmark*, http://www.iozone.org , 2005.

[11] Lowth, Chris, *TCP/IP connection cutter*, http://www.lowth.com/cutter , 2003.

[12] Mayo, Jean and Phil Kearns, "A secure untrusted advanced systems laboratory," *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, 1999.

[13] Miltchev, Stefan, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith, "Secure and flexible global file sharing," *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*. USENIX, 2003.

[14] Neuman, B. Clifford and Theodore T'So, "Kerberos: An authentication service for computer networks," *IEEE Communications Magazine*, Vol. 32, Num. 9, pp. 33-38, 1994.

[15] O'Shanahan, Declan Patrick, *CryptoFS: Fast cryptographic secure NFS*, Master's thesis, University of Dublin, 2000.

[16] Powell, P. and J. Mason, "Lprng – An enhanced printer spooler system," *Proceedings of the Ninth USENIX Systems Administration Conference*, pp. 17-22, USENIX, 1995.

[17] Russell, R., *The netfilter project*, http://www.netfilter.org , 2005.

[18] Vlachos, K., N. Nikolaou, T. Orphanoudakis, S. Perissakis, D. Pnevmatikatos, G. Kornaros, J. A. Sanchez, and G. Konstantoulakis, "Processing and scheduling components in an innovative network processor architecture," *Proceedings of the 16th International Conference on VLSI Design*, IEEE, 2003.