Transient Addressing for Related Processes: Improved Firewalling by Using IPV6 and Multiple Addresses per Host

Peter M. Gleitz*
pmgleit@netscape.net

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

Abstract

Traditionally, hosts have tended to assign relatively few network addresses to an interface for extended periods. Encouraged by the new abundance of addressing possibilities provided by IPv6, we propose a new method, called Transient Addressing for Related Processes (TARP), whereby hosts temporarily employ and subsequently discard IPv6 addresses in servicing a client host's network requests. The method provides certain security advantages and neatly finesses some well-known firewall problems caused by dynamic port negotiation used in a variety of application protocols. A prototype implementation exists as a small set of *kame*/BSD kernel enhancements and allows socket programmers and applications nearly transparent access to TARP addressing's advantages.

1 Introduction

In the simplest of inter-networked host models, a client or server host has a single network interface with a single network address identifying the host. Even under such an elementary set-up, firewalls have traditionally faced difficulty when confronted with application protocols needing to open secondary channels. Examples abound, most notably `ftp`, but also `rsh`, RealAudio, H.323, `tftp` and the X Window System. To operate with such popular applications, firewalls have been forced either to follow the application layer protocol and configure themselves appropriately or to keep open, sometimes unnecessarily, a range of ports.

As an alternative to potentially complex, detailed, and often stateful firewall interaction, we propose a method using multiple network addresses per host to organize and simplify firewall decisions. Under our basic model, instead of trying to follow the unfolding application protocol details, the firewall makes an initial permissibility determination based on transport layer protocol and the endpoints' ports and addresses. Assuming approval of the proposed transaction, the firewall subsequently permits all traffic between the approved address pairs, irrespective of port.

The security concerns arising from the firewall's apparent loss of control over a session's evolving ports will be alleviated by dynamic control of the protected host's active addresses. Further, by segregating and controlling which addresses offer network services outside the firewall and which facilitate protected-host driven network requests, the architecture provides a natural address based division between potentially hostile requests from outside the bastion, and presumably benign outbound activities originating within the protected network.

To help distinguish among a protected-host's various client/server tasks, we will tie the address used by a client process to its process group identifier. This way, host addresses will come and go as part of the natural lifecycle of the processes that use them.

For example, when a TARP client starts `ftp` from behind a firewall to a similar server, also protected by a firewall, it configures a new address. The client's firewall recognizes the client's new address and records it along with the destination address. The firewall presumably grants the FTP request based on port and address criteria and subsequently passes all outbound and inbound packets between the two addresses. Further port negotiations (using the same addresses) conducted by `ftp` will have no effect on what the client side firewall passes; it will simply pass along all the server packets addressed to the client's ftp address.

The server side behaves similarly (though there are some

* Work done while interning at AT&T Labs Research

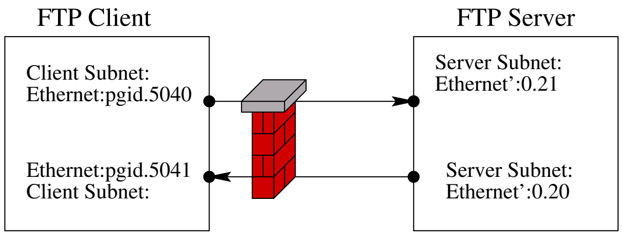

Figure 1: TARP Client FTP Through a Firewall: When the client connects to the FTP server, the client's firewall recognizes that FTP uses auxiliary ports and opens all ports between the server and the client's transient address, Client Subnet:Ethernet:pgid.The FTP thus proceeds without the firewall knowing that the client negotiated port 5041 for the data channel.

issues if different servers have different access policies); however, servers must use static addresses, largely so that clients can find them. The server's firewall sees the client address opening a connection to the server's fixed address and ftp port and decides whether to allow the connection. If the server's firewall permits the connection, it passes all subsequent packets between the client and server addresses, independent of port, including incoming calls.

Another way to view this is to adopt a virtual machine metaphor. That is, each client process group conceptually runs in a virtual machine, with an independent IP address space. This view can guide several design decisions, as shown in Figure 2.

Although IPv4 might conceivably support such a scheme, IPv6 and its large 128-bit addresses [HD98] provide a simpler opportunity to deploy TARP. Typically, the high order 64 bits of an IPv6 address provide the information needed to deliver a packet to the host's LAN, and the low order 64 bits, assigned by the site administrator, are commonly the interface's 48-bit Ethernet address padded with a constant to create a unique 64-bit interface identifier. TARP's addressing scheme exploits these pad bits to create a family of 2^{16} IPv6 addresses with the same basic uniqueness properties as Ethernet-based host identifiers. These addresses become associated with the sockets of a process group and form the foundation for TARP.

A proof-of-concept implementation exists as a small set of *kame*/BSD [KAM] kernel enhancements, and allows socket programmers and applications transparent access to TARP's advantages. The basic server/client programs supplied with the *kame* IPv6 package (`tftp`, `ftp`, `rlogin`, `rsh`, `telnet`, `sendmail`, `ssh`, `mozilla`, `inetd`) all work with the modified kernel, and none of them required modification to run.

2 Multiple Addresses per Host and a Large Family of IPv6 Addresses

The notion of assigning multiple IP addresses to a (non-routing) host is not new. In 1994, RFC 1681 [Bel94] described a number of potential uses and benefits of assigning multiple IP addresses to a host, and Stevens' standard socket programming text [Ste90] is peppered with some of the subtleties of socket programming on multihomed servers. The IPv6 addressing model specifically supports assignment of "multiple IPv6 addresses of any type (unicast, anycast, and multicast) or scope" [HD98] to a single interface. Although primarily intended to facilitate renumbering, this IPv6 feature can also be used to simplify firewall interaction via TARP.

To form addresses based on unique interface identifiers, RFC 2373 [HD98] recommends using, whenever possible, the 48-bit IEEE 802/Ethernet MAC addresses, and the RFC offers the IEEE EUI-64 [Ins97] as one way to create a globally unique 64-bit identifier from the Ethernet address. The suggested method consists of inserting the two-byte padding pattern $FFFE_{16}$ after the third byte of the the Ethernet address.

To create TARP addresses, we actively vary the pad bytes to select from a large family of IPv6 addresses unique to an interface. We also use a slightly different ordering than the examples in RFC 2373. The first six bytes of our addresses are simply the interface's Ethernet address with the last two bytes varying as needed. Re-ordering the bits that vary to the least significant bits has the benefit of allowing the last hop router to use a single table entry with a 112-bit prefix to send all such packets to a single host.

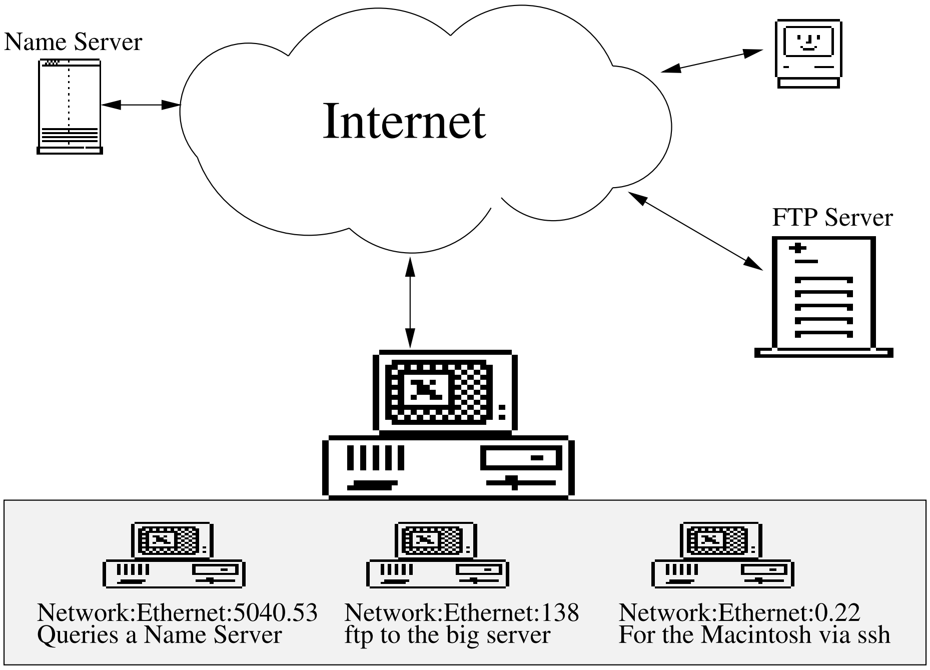

Figure 2: An Example Under the Virtual Machine Model:Viewing each process group as a virtual machine, the figure shows a single TARP host involved in three distinct network sessions run by separate process groups 1) an FTP session from the TARP client to a server across the WAN ; 2) the TARP client serving an `ssh` session to a Macintosh client; 3) A DNS request originating from a process within process group 5040.

3 Some Notation

We now introduce some notation. Suppose a host interface has an Ethernet address of 00:10:4b:63:80:4b, a 64-bit network prefix length and a subnet prefix of 1111:2:3:4. Then the TARP address family available for this interface ranges from 1111:2:3:4:10:4b63:804b:0 through 1111:2:3:4:10:4b63:804b:ffff. We describe the general concept of an address like 1111:2:3:4:10:4b63:804b:0, as *Subnet Prefix:Ethernet Address:0* and when the least significant bits of an address correspond to the process group identifier using the address, we write *Subnet Prefix:Ethernet Address:pgid*. When the discussion requires some notion of a port, a dot after the final address bytes delimits the port, as in *Subnet Prefix:Ethernet Address:0.21* for port 21, or *Subnet Prefix:Ethernet Address:0.** meaning any port on the address. As a further notational imprecision, quantities like *Subnet Prefix* and *Ethernet Address* can be enhanced to refer to some particular network, interface, or host context, as in *Subnet Prefix Network 1*, or *Ethernet Address Host 2*, *etc*.

4 Using a Large Family of Host Addresses

Host usage of this this large address family depends on whether the host process acts as client or server. In general, client applications select actively from the pool of available addresses and dynamically configure the appropriate interface. Server processes will use a designated static address.

Our decision to connect the address used by a client process to its process group was driven only by a desire to have a group of related processes use the same address. Process groups have no particular special properties, other than being somehow "related" (on Unix systems, they represent the processes descended from a single command line), and the basic concept of transient addressing extends to other similar, natural process relationships. As the basis for the UNIX process group indexing scheme, processes could, in principle, have their own IP addresses, although the clutter and possible performance penalties of establishing a per-process address seem excessive. In a multi-user environment, UIDs, credentials, or session identifiers could sensibly anchor similar addressing schemes, as could a new (as yet unspecified and only dimly imagined) set of kernel concepts and data structure for organizing related processes.

More fundmanentally, the requirement is that the addresses assigned to "related programs"—ones that somehow "expect" to be at the same place—be stable, but that "different" programs have different addresses. Thus, a Web browser structured as multiple processes should have a single IP address, while individual `ftp` commands should have different addresses.

Address organization by process group has the advantage that the kernel already supports the notion that a collection of processes is somehow related. The data structures are stable in the UNIX kernel, and bolting the addressing scheme onto them posed little threat to the processing flow for establishment, maintenance, and appropriate removal of the fundamental data structures managing process groups. We are unaware of any basic conflicts created by attaching our concepts to process groups, and found them lively and flexible enough to stress the kernel enhancements dealing with address management.

4.1 How a Client Uses the Large Family of Addresses

When a client implicitly binds a local address to a socket, the kernel decides which interface to use. Our modified kernel determines what process group the binding client process belongs to, and uses the corresponding *Subnet Prefix:Ethernet address:pgid* address for the socket. For a process group's first such socket, this involves assigning a new IPv6 address to the appropriate interface and joining the corresponding Solicited-Node multicast address required by the IPv6 specification [HD98]. Note that by only assigning a TARP address in the case of an implicit bind, the kernel still allows explicit binding to any valid host address a socket program can identify. With an address established for the process group, any other implicitly bound sockets arising from the same process group will use the same address. The address persists on the interface until the process group leader terminates, when the address and its Solicited-Node multicast address are removed.

4.2 How a Server Uses the Large Family of Addresses

Server interactions passing through the firewall must occur on the reserved *Subnet Prefix:Ethernet Address:0* address. This requirement serves several useful purposes. First, connecting to a server with a dynamic address is like hitting a moving target, so fixing the address allows clients to find servers. Next, fixing the server address (both listening and responding address) further assists the expected server/firewall interaction by omitting any need for server applications to inform interested firewalls of their addressing dynamics. Finally, it provides control for firewalled hosts running servers using wildcard listening sockets.

The control requirements are the most subtle, and an example of what could be dubbed "coat tail riding" will clarify the importance of restricting external server activity to a fixed address. Consider an installation with a policy freely allowing outgoing `ftp`, but not incoming `telnet` from outside its firewall. A host inside the installation opens an FTP connection from *Protected Installation Prefix:Ethernet Address:pgid.5040* to a hacker-compromised site, *Compromised FTP Server.21*. This complies with the installation's security policy, and the firewall enables the rule allowing all traffic between the protected host and the compromised server, $\{$*Protected Installation Prefix:Ethernet Address:pgid.** $\longleftrightarrow$ *Compromised FTP Server.**$\}$. A telnet server listening to a wildcard socket on the `ftp` client machine could then receive and serve connection requests from the compromised FTP server to *Protected Installation Prefix:Ethernet Address:pgid.23* in violation of the installation's telnet policy. We prevent a server from riding back on the client's coat tails by forcing server activities to occur on the :0 address and preventing a server's wild card listening sockets from connecting to inappropriate TARP addresses. Section 5.4 describes a set of socket matching rules that prevent coat tail riding. Note we cannot simply forbid connections to *Protected Installation Prefix:Ethernet Address:pgid* as active mode FTP will require that the compromised server connect back to the client.

The address based firewall rule and designated server address principles lack fine discrimination concerning what services will be served outside the protected bastion. Once a client opens a connection through the firewall to any service, the firewall rule subsequently allows the same client through to any port. In another example, suppose the protected host's security policy allows `ftp` connections from the outside, but not `telnet`. Any authorized user could start an ftp from the outside to *Protected Installation Prefix:Ethernet Address:0.21*, causing the firewall to allow all traffic between the ftp client and the protected server. With all ports now open to the FTP client, it can now start a `telnet` session from the outside, unimpeded by the firewall. This problem of finer discrimination of service access has some reasonable remedies short of following application-level protocols, and we discuss them in the security section below.

5 TARP and Sockets

5.1 Overview

Our TARP implementation rests on a set of kernel enhancements at the socket level and below. It relies extensively on the socket data structure's process group identifier, and every AF_INET6 socket receives the process group identifier of the process group creating it. The main intent has been for socket programs to receive, without further programmer intervention, the appropriate TARP address when simply using the vanilla Socket API.[1]

As illustrated above, it matters greatly to the addressing scheme's security concept whether a process acts as a server or client. As a result the kernel tries to classify a process group as either a client process group or a server process group, and typically assigns a classification to a process group according to the first SOCK_STREAM or SOCK_DGRAM socket usage of a process in a process group. A process group typically exists in an indeterminate state until one of its processes first reads or writes a socket. The exception to the usual indeterminacy occurs when a server spawns a new process group. In this case, the new process group inherits its server state from the parent process group. If a process group's first datagram or stream socket usage is a read, the enhanced kernel classifies that process group as a server process group. Similarly, if the first such socket usage writes a socket, then the process group receives a client classification. The implementation offers system calls to manipulate a process group's client server state for socket applications ill-served by the default classification rules, but experience has so far shown them needless: except for `auth` (see Section 6.1), all tested applications worked without change.

[1] In this prototype, we have not fully checked if SS_ASYNC's usage of the sockets process group components may conflict.

5.2 Client Sockets

Client applications wishing to use a TARP address simply create a socket and first write to it (using `write`, `writev`, `sendmsg`, or `sendto`) or call `connect` so that the socket receives an implicit `bind`. During the process of implicitly binding the socket to a local address, the kernel assigns and configures the appropriate TARP address.

5.3 Server Sockets

Like client sockets, server sockets can receive the source address of their outgoing packets via an implicit bind, but sometimes, this produces undesirable results. Under the general model that servers will create sockets somehow listening on some limited set of addresses for contact from clients, one of our goals is to insure that servers will listen to and respond from the same reserved server address that was contacted by the client. Achieving this goal depends on the transport layer protocol, the Unix flavor, and the server's socket programming style.

TCP servers pose little problem as the connection framework provides adequate structure to determine the correct address for a listening socket to bind to (presumably a reserved server address contacted by a client). We do not force the `accept` to occur on a server address, but rather view that decision as a administrative choice for the firewall. By allowing connections to valid addresses outside the TARP address family but sharing the same interface, we can also support multi-homed TCP servers. We expect (but do not force) that installations using these extra addresses will have firewalls routinely blocking all traffic involving these non-process group based addresses so that they will be served only from within the protected enclave.

UDP client/server applications are slightly more problematic, mostly because the BSD derived kernels do nothing to force a multihomed UDP server to respond from the same address as the destination address of the prompting packet[Ste90, p. 220]. Coupled with the reality that many common UDP servers let the kernel choose the server's responding address, this effectively causes problems for clients attempting to use UDP sockets to communicate with multi-homed UDP servers running a kernel supporting TARP addressing. The BSD based kernels choose a reply address from the outgoing interface which does not necessarily mean replying with either the same address as the client's destination address or the designated :0 server address. If a client contacts a UDP server on *Subnet Prefix:Ethernet Address:0*, the kernel must insure that the response returns from the same server address or the firewall rule set will likely block the response.

For UDP servers, the classification of a process group as a server process forces response on the server address, achieving the desired result. The whole classification scheme is mostly forced by BSD's carefree attitude about ensuring that the UDP server responds from the same address contacted by the client.

5.4 Matching Protocol Control Blocks for Inbound Data to Sockets

To read incoming data, a process typically creates a socket and places it in a state (often blocking, but perhaps polling) awaiting network data. The kernel monitors incoming IP data, classifies it according to transport layer protocol and ports, and finally matches it to a list of available sockets. To work well with TARP addressing, the standard BSD rules for matching network data to sockets need refinement, and the new assignment rules are described below.

These assignment rules enforce several operating principles. First, they must prevent coat tail riding by providing the control necessary to prevent wildcard listening sockets from providing network services on a process group based address. Second, the rules must also permit connections to servers on the *Ethernet:0* address. Third, they need to connect the appropriate incoming packets to sockets listening specifically to a TARP address. Finally (and optionally), because servers may wish to provide separate services inside the protected bastion, the rules need to allow a server to match sockets listening to a valid address outside the TARP address family. The rule set below achieves these goals.

1. A socket with a socket pair ({*source address.source port, destination address.destination port*}) exactly matching a datagram's source and destination ports and addresses receives the datagram payload. This rule typically applies to TCP sockets from established connections, for example.
2. Bound Listening Sockets: If no match is found in Rule 1, the sockets listening to a particular address are examined sequentially for the first match against the following ordered rule set:
    a. A socket listening to {**.*, Subnet Prefix:Ethernet Address:0.port*} receives the data sent by any client to the server on that port. This rule applies to server connection establishment, and UDP servers on the server address.
    b. A socket belonging to process group *pgid*, and listening to {**.*, Subnet Prefix:Ethernet Address:pgid.port*} gets data sent to that port and address. This rule allows TARP address clients to use listening sockets to accept connections from servers, like an `ftp` client might do.
    c. A datagram addressed to a socket listening to the specified port on a valid host address outside the TARP address family will receive data sent to that port and address. This is intended to handle requirements to provide services not available outside the protected bastion to internal hosts.
3. Wildcard Listening Sockets: If no match for any of the sockets is found in Rule 2, then a socket listening to {**.*,*.port*} can match if and only if:
    a. The datagram from the client is addressed to the server address.
    b. The datagram is addressed to a valid TARP address and the socket belongs to the process group designated by the address.
    c. The datagram destination address is outside the TARP Family but a valid host address.

    Rule 3 is a catch-all and the last socket matching under Rule 3(a), 3(b), or 3(c) receives the datagram.

An example shows how these rules effectively prohibit coat tail riding. Reconsider the example of Section 4.2, where a protected client runs both telnet and FTP servers on a wildcard address, but allows only FTP to outside the firewall. After the protected client starts an `ftp` session to the compromised FTP server, the compromised site attempts to open a connection to *Protected Installation Prefix:Ethernet Address:pgid.23*, and the firewall passes the malicious SYN packet according to the rule legitimately established by the `ftp` allowing {*Protected Installation Prefix:Ethernet Address:pgid.** $\longleftrightarrow$ *Compromised FTP Server.**}. As the `telnet` server is listening with a wildcard socket, the the first two rules (each requiring a specified destination address) will fail to match the SYN packet. The only remaining candidate is Rule 3(b), but this fails to match because the `telnet` server's listening socket will belong to a different process group than the `ftp`. Thus the packet intending to ride the coat tails of the open firewall filter is never delivered to the telnet server socket and the connection is never opened.

Rules2(c) and 3(c) are optional and permit a host to offer services on an address outside the TARP family, presumably to machines within the protected enclave. They should be filtered outside the protected enclave and also admit the possibility of firewall misconfiguration. For installations not requiring their functionality, a compile option can remove them and protect against misconfiguration dangers.

5.5 Aids for Socket Programmers

Although the enhanced kernel modifications work sensibly with a variety of unmodified IPv6 server/client software, socket programmers using TARP addresses may need to be mindful of certain subtleties. Socket programmers writing applications that fork new processes and subsequently assign the forked processes to new process groups will will need to remember that the resulting sockets may not share the same IP address (if that matters to them or their application). Additionally, the inheritance of a parent's process group server state could conceivably create problems if a server forks a process that needs to act as a client.

As a convenience and measure of control, the modified kernel provides a system call allowing a program to request either server or client status. So far, the client/server applications tested have required no such manipulations to use the correct addresses, suggesting that the kernel rules generally provide the desired results. During development, the modified kernel also contained a `setsockopt` call allowing programmer control of whether a socket uses the server addresss, but this control appeared superfluous in light of the server inheritance rule and was removed pending demonstration of its utility.

Host applications may also explicitly bind to any address they could otherwise bind to with an unmodified kernel, although doing so will likely produce useless results. For example, an application could bind to the address used by a separate process group, but the resulting socket would (absent process group manipulation) have an address with the process group component not equal to the socket's process group. This would always fail the protocol control block matching algorithm joining sockets and protocol control blocks, and so no data could flow to the socket. (A simple code change could prohibit the ability to misbind, to avoid violating the rule of least surprise.)

6 Experience with Client/Server Interactions

6.1 TCP Applications

Our experience is primarily limited to some of the IPv6 server/client inetd programs available from *kame* : `inetd`, `tftp`, `ftp`, `rlogin`, `rsh`, and `telnet`, and `auth`, although we also worked successfully with IPv6 ports of `ssh` and `sendmail` as well as with a web browser `mozilla` and a web proxy `wwwoffled`. Except for `auth`[Joh93], the TCP applications all appear to run reasonably well, unmodified, with the enhanced TARP address kernel. The UDP client/server pairs also ported well, though with some mild restrictions concerning the server's responding address.

The problems encountered by the unmodified applications depend variously upon the session layer protocol, the application layer protocol (`auth`), and whether the application uses address based authentication (*e.g.* `rsh` and `rlogin`). Cognizant of the terrible security properties of address based authentication [Bel89], we question the wisdom of bothering to fix `rsh` and `rlogin`, especially with `ssh` working, but their functionality can be restored with modest DNS adaptations discussed in Section 7.3. We have not performed exhaustive testing. (We do note that `ssh` [Ylo96] can use cryptogrpahically-protected name based authentication; similarly, `rlogin` et al. are secure when protected by IPsec [KA98]. If use of such protocols is considered desirable, we may need to reopen this issue.)

Of the TCP applications, `ssh`, `ftp`, and `telnet`, `sendmail`, and `ssh` appear to function normally with the modified kernel, while `rsh` and `rlogin` both suffer mildly from authentication problems already described.

The `auth` server presents insurmountable problems, mainly because its poor interaction with TARP addressing. Consider a server installation wishing to fire an `Ident` query to a TARP client's auth server about a connection coming from, say, *Subnet Prefix:Ethernet Address:pgid* . `Ident` queries consist only of the port numbers; the query addresses are implied by the source and destination address of the queries. The querying host has two choices for an address to query, neither of which work. Querying the *Subnet Prefix:Ethernet Address:pgid* will fail, as there will be no server on the process group address. Querying the server address, *Subnet Prefix:Ethernet Address:0* yields a null result, as the connection does not originate from that address. This failure seems a small loss, as the quality of the information returned from the `auth` server has always been greatly disparaged (c.f.[CB94, GS96], and even its defining RFC [Joh93]). As a workaround, client installations feeling an obligation to reply to `Ident` queries should have little trouble hacking a small `Ident` server into client applications, so that the application itself replies to `Ident` queries on its process group address. Of course, such an implementation would require a `setuid` helper program.

6.2 UDP Applications

By restricting UDP clients and servers to asking only for services and responding to requests (respectively) from an interface's designated server address, then all the *kame*/BSD UDP client/server applications work correctly, unmodified. We wrote simple socket clients to communicate with `inetd`'s built-in servers, and (except for `auth`) succeeded with both the TCP and UDP versions of: `time, echo, daytime`, and `chargen`.

We should distinguish that our UDP clients used *unconnected* UDP sockets when communicating with the built-in servers. When requesting service from a valid address outside the TARP Family, the clients executed a `sendto` followed by a `read`. Following the rule that processes first reading then writing sockets are servers, the server responded from its *Subnet Prefix:Ethernet Address:0* address, which the client successfully read from its unconnected socket.

Had we instead used a client socket connected to the valid non-TARP address where the UDP service request was sent, the UDP client server transaction would have failed when the reply returned from the server address. This problem with connected sockets is not isolated to the modified kernel, but also exists when interacting with multihomed Berkeley-derived UDP servers [Ste90, pp. 219–220]. Under BSD, the server is only guaranteed to reply from an address chosen from the same interface as the contacted server address. The server can reply with an address other than the one contacted by the client (but on the same interface) and a connected UDP socket will fail to read the server's reply.

6.3 ICMPv6

Some network services are bound to the existence of an IP address, rather than to a specific port number. For example, ICMP messages are received by a host, rather than by a specific process. How should messages sent to TARP addresses be treated?

Our primary guiding principle is that of the virtual machine, though for implementation reasons we cannot always follow this. Thus, a "ping" message sent to a TARP address should be replied to, because a real machine with that address would respond.

Some ICMP messages create state on the receiving host. For example, ICMP Redirect changes the local routing table. Given the security risks posed by this message [Bel89], it would be nice if these changes could be restricted to the particular process group to which the messages were addressed, but that would require substantial kernel changes. Furthermore, there are distinct advantages to making certain state information global, such as that distributed by Path MTU [MDM96]. Further work is needed in this area.

7 Network Interaction

7.1 Firewall Interaction

TARP was designed for firewalls, with the goal of providing a mechanism for firewalls to make sensible security decisions without following application layer protocols. The resulting firewall interactions are intended to be straightforward and simple. The main precept is to use address based filtering after an initial authorization. For a connection based protocol like TCP, this means port information need only be examined at connection establishment.

For outbound data, the firewall only needs to examine the outbound destination port, determine whether the originating address belongs to a family of TARP addresses, and know whether the protocol involved uses auxiliary data connections involving other ports. Assuming it approves the protected client's transaction and the underlying protocol requires dynamic port negotiation, the firewall simply permits any incoming traffic to that address, regardless of port. The firewall no longer needs to know any protocol details; it simply needs to know that the protocol involves secondary channels. For services using only fixed ports, the firewall can filter traditionally.

For service requests originating from outside the protected bastion, the firewall typically would reject all requests for the services of an address other than a designated server address. For authorized services, the firewall permits packets to flow freely between a server address and the client outside. For example, an FTP server host administratively configured to provide no other services outside the firewall would reject all inbound connection attempts and UDP packets other than those to the FTP server port. Following a successful connection, outbound packets to the client from *Subnet Prefix:Ethernet FTP Server Host:0.** are freely allowed between the approved server and client.

The firewall can use several approaches to revoke authorization. TCP connections are easiest: If the packet triggering authorization comes from a TCP connection, the firewall simply disallows (pending possible future authorization) packets between the two addresses when all related connections (as determined by address) terminate. Additionally, and more appropriate for UDP and ICMP, a simple timer mechanism can revoke authorization some number of minutes after the last use of an address. Finally, a protected host can explicitly release an address, upon process group termination.

Having a host communicate its release of an address provides the firewall the best general understanding of when to terminate authorization, but it requires the protected host to know how to reach its firewall(s). The best control exists in situations where all relevant firewalls are either embedded in the host (like a Distributed Firewall [Bel99]) or or share the same link layer and are able to see the link local ICMPv6 broadcast as a result of the host leaving the Solicited-Node multicast address. Here, the firewall either knows or has an inkling of the address being removed at process group termination. The embedded firewall can simply de-authorize the canceled address from within the kernel, while the link local firewall will need to match the Multicast Listener Done message to its authorized addresses, and cancel upon determining their unreachability.

For installations where routers lie between the protected host and the relevant firewall, hosts wishing to communicate address revocation will need more complex interaction mechanism with their firewalls.

Using a traditional proxy firewall to protect hosts with TARP addresses works as before, but will require some flexibility in configuration to accommodate the large number of addresses that will be coming and going. For example, certain outbound filter rules will need to apply for *Subnet Prefix:Ethernet:** where before they perhaps pertained only to a single address per host.

Note well that clients are usually well protected, even if the firewall is slow revoking authorization. This is because when a process group terminates, its corresponding address is scrubbed from the client's interface, so even if the firewall is tardy blocking the inbound packets, they fall on deaf ears when reaching the host. Any nefarious or spurious inbound packets to unconfigured addresses should ultimately die an unconsumed death. *This is the fundamental reason why TARP-based hosts can simplify firewalls.*

Because we use only the initial address and port of a connection, we have prevented new application protocols from obsoleting otherwise adequate firewall software. Instead of requiring complex (and possibly buggy and insecure) new application protocol-aware software to be written and installed in the firewall, to keep up with new application protocol developments, the firewall administrator need only know what port the service uses to start its transactions and filter accordingly. Thus, only minor firewall configuration changes will accommodate a large class of new, unforseeable applications. (This work does not discuss application-specific audit trails or intrusion detection system events that an application-aware firewall may collect today.)

7.2 Router Interaction

At one level, routers are not affected by TARP addresses. They will see an IP address for which they have no corresponding link-layer address cached; accordingly, they will resort to the Neighbor Discovery Protocol [NNS98], in normal fashion. But TARP-aware routers and protocols can do better.

Widespread use of TARP would require considerably more storage on routers for link-layer addresses. But this set of IP addresses all share a common prefix and a common link-layer address. Accordingly, routers could employ a single mapping of *prefix* to link-layer address. Unfortunately, that is not supported by NDP. If necessary in a given environment, this could be faked by having a host pretend to be a stub router; however, this would require the host to participate in routing protocols, which is generally considered to be a bad idea. A better solution would be to extend NDP to handle host address prefix lengths.

7.3 DNS Interoperation

As evident in the discussion concerning `rlogin` and `rsh`, TARP Addressing encounters problems interacting with DNS. Of particular concern is the dubious practice of so-called "double-reverse lookups" (cf. [ZCC00] and [GS96]) where a DNS client attempts address based authentication by first looking up the hostname for an IP address and subsequently checking the IP address(es) for that hostname, requiring a match to proceed. But for double-reverse lookups, most dynamic address clients would probably work happily by simply registering their server address with the appropriate name server; a wildcard PTR record would yield the same name for all possible TARP addresses.

IPv6-to-hostname lookups pose no problem, as the DNS specification for IPv6 [TH95] provides the necessary wildcarding. For example a name server could associate a wildcard to a PTR record for, say, **.12-nibble Ethernet Address.16-nibble Subnet Prefix.IP6.ARPA* (ordered appropriately) and correctly return the hostname for a host's Process Group Dynamic Addresses. Another possibility is to query the host directly with an ICMPv6 FQDN query, as described in an IETF draft "IPv6 Node Information Queries" [Cra99] and already implemented by *kame*.

Hostname-to-address queries are more problematic. A standalone name server presumably has no information about a host's active addresses, and absent refinements in the name server/resolver software and protocols, a complete reply to a hostname-to-address query would consist of all 2^{16} Process Group Dynamic Addresses. Unfortunately, simply listing a host's server address is insufficient, as a double-reverse lookup will find inconsistency when comparing a client's dynamic address to the its server address.

Three straightforward solutions to the basic problem come to mind. The simplest treats a Process Group Dynamic Address client as a domain. Its parent DNS server delegates authority to the client and simply refers queries to the clients themselves for final resolution. The clients can then run a pared down name server which responds with the appropriate set of addresses for a name-to-address query.

In another method, the TARP clients use the DNS UPDATE message [VET+97] to inform the relevant name servers of their evolving address state. It is not clear how well this would work in practice, as problems could arise in situations where the querying agent has a faster path to the name server than the TARP client. A clever TARP client could conceivably delay outgoing datagrams from a new TARP address until a DNS update could be verified, but this might prove unnecessarily cumbersome.

A third method embeds the process group portion of the address into a synthetic hostname and places the corresponding 2^{16} possible hostnames as entries in the nameserver database. For example the fictitious TARP client host *foo.bar.com* would have name server entries for *0%foo.bar.com* - *65535%foo.bar.com*, where we use the % to mean a meta-character (determined locally by bar.com) denoting that the hostname uses Process Group Dynamic Addressing. The name server would conceptually have 2^{16} PTR records mapping each Dynamic Address family member to its corresponding expanded hostname (e.g. pgid%foo.bar.com) The cross-check query invokes an address-to-hostname query for *foo.bar.com Subnet Prefix:foo.bar.com Ethernet Address:pgid*, which eventually returns *pgid%foo.bar.com*; this is followed by a name-to-address query about *pgid%foo.bar.com*, which will provide a consistent check if the name servers are configured correctly.

This third method scales poorly, taxing the DNS servers serving TARP client domains by forcing them to store 2^{17} records per client. While some of this can be dealt with by "syntactic sugar"—the servers need not store all 2^{17} records, but can simply generate them on the fly—we are left without a canonical name for the host. This affects host configuration files (`/etc/hosts`, `.rhosts`, *etc.*), email, and other services that require one true name for each machine. (In a sense, this is carrying our virtual machine metaphor too far.)

Finally, we note that the problem only arises because hosts use name-based authentication, and hence need the extra protection of the double-reverse lookup. If this practice were to be abandoned—and we strongly suggest that that be done in any event—the name-to-address lookup could be omitted, thereby eliminating the problem.

8 Implementation Details and Performance

8.1 Implementation

Our testbed implementation consists of a handful of subroutines and about 400 lines of C code inside the *kame*/FreeBSD 3.4 kernel. The kernel enhancements appear to port easily to the other BSD families *kame* supports (NetBSD, BSD/OS and OpenBSD). The essential code flow remains unchanged, and the bulk of the software changes sit inside a single *kame* subroutine, `in6_selectsrc` (basically an IPv6 re-work of parts of the function of `in_pcbconnect` [WS95]), called when assigning a local IPv6 address to a protocol control block.

The *kame*/FreeBSD code has a parameter for the maximum process identifier, PID_MAX, which comes configured at 99999. Our implementation reduces this maximum to 65535 to insure the resulting process group identifiers fit entirely into the corresponding two bytes of the TARP address.

Given the uniqueness properties implicit the Ethernet address based addresses, the implementation does not perform Duplicate Address Detection [TN98] when adding a TARP address to an interface. This enhances performance by preventing unnecessary delays when configuring an interface with a new address, and yet still complies with the RFC's notion that addresses based on unique interface identifiers need only check for duplicate addresses at initialization ([TN98], Section 5.4). When bringing up an interface, the *kame* IPv6 implementation configures it with a proper Ethernet-derived, EUI-64 compliant link local address. This address receives proper duplicate address detection, and so any (presumably relatively rare) problems caused by locally duplicate Ethernet addresses should be caught when the interface goes up.

Regrettably, the IPv6 Address Autoconfiguration Standard [TN98] lacks a sub-net mask capability that would allow hosts to reserve a range of addresses under a mask. Were such a facility available, a TARP host could make a single duplicate address detection query to reserve a whole range of an address family with a single query. In this case, hosts would be freed of needing to use the Ethernet address-based connection for its implied uniqueness properties, and could still safely reserve a range of without fear of address collisions. Local administrators would then have the freedom to assign the host address portion according to their own rules, and could conceivably use even larger address families.

We could, in principle, do duplicate address detection for each TARP address. However, the overhead, and particularly the 1-second timeout, are prohibitive.

There is some conflict between this scheme and the IPv6 privacy extension standard [ND01]. The easiest solution is to use the suggested algorithms to generate just a replacement for the MAC address portion of the address, rather than the full low-order 64 bits.

8.2 Performance

In our tests thus far, there has been virtually no impact on performance. It is clear, however, that current kernel algorithms will not scale well. The list of IP addresses per interface is kept on a linked list, which implies a linear search for each packet received. Clearly, this is inadequate if there are many process groups active at any time. The obvious alternative is to use a more sophisticated data structure, though it would have to be one that permitted speedy additions and deletions.

An alternative would be to compare the prefix of the address in incoming packets to the base address of the interface, and use existing speedy look-up mechanisms to ascertain if the associated process group exists. That is, assume that a packet is destined for a machine if the base address is valid and the process group exists, rather than checking if such a process group has actually performed network operations.

There is one other area of some concern. As noted earlier, for each new TARP address allocated it is necessary to join the appropriate Solicited-Node multicast address group. For some hardware designs, it is necessary to load the group address onto the controller chip. Depending on the chip and driver design, this may be an expensive operation.

9 IPSEC Interaction

Successful IPSEC interaction essentially depends on the ability of hosts running TARP addressing to conduct the necessary key exchanges. Fortunately, ISAKMP and IKE provide an adequate framework to support TARP. A simple solution uses either of ISAKMP's [Pip98] address range or subnet identification payloads to designate that the ISAKMP peer negotiating the key exchange is the address range or subnet of the TARP Family. Using keys negotiated for the address family thus permits processes to use IPSEC in conjunction with TARP addresses.

Depending on the threat model, installations may wish to eschew IKE's Base Quick Mode [HC98, p. 16]. Lacking perfect forward secrecy, Base Quick Mode admits the possibility that an authorized host process able to obtain its keying material can use that knowledge to determine keying material for other processes, including those belonging to other users. The Quick Mode key exchange payload option [HC98] prevents this problem by providing the necessary perfect forward secrecy at the cost of an additional exponentiation. Alternatively, use of cryptographically strong random number generators, ciphers resistant to chosen-plaintext attacks, and suitable crypto-APIs (i.e., those that will not, under any circumstances, disclose a session key to an application) can prevent this attack.

10 Interaction with Other Systems

At this point in the evolution of IPv6, it would not be acceptable to introduce a new scheme that would break compatibility with existing systems. For the most part, we have not done so.

Servers do not notice anything different, with the possible exception of contacts from many more clients. That depends on whether the server is noting addresses or names, and in the latter case, on what mechanism is selected for address-to-name resolution.

Servers that do rely on host names for authentication may have problems. As noted, we recommend that that practice be abandoned in any event; we do not consider its difficulties to be a disadvantage of our scheme.

11 Security Implications

In some sense, the simplicity of the security enabled depends on the assortment and type of network services a protected host needs to provide. TARP addressing offers simpler, more robust protection for hosts acting primarily as clients rather than servers. This does not mean that many server configurations cannot receive adequate protection using the method and a compatible firewall, rather that servers will need to be much more careful about their configuration to achieve results comparable to hosts acting entirely as clients.

11.1 Client Security

Clients offering no network services can be well-protected by TARP addressing and an accompanying firewall. Such clients will also operate reasonably unimpeded by their firewalls. The main observation is that absent user-directed activities, the typical client providing no network services (a dedicated workstation, for example) could be protected from all TCP and UDP datagrams from outside the protected bastion. It thus shares many of the security features of the proverbial unconnected host. Yet if some user directed activity makes it necessary to leave the protected enclave (a user wishes to run `ftp` to retrieve a file from the Internet for example), sufficient connectivity is enabled to allow the user to conduct transactions, unimpeded by tight security policies concerning connectivity to the outside. Furthermore any vulnerability created by the FTP client's network activity terminates with the FTP client process.

11.2 Server Security

For services using an evolving set of ports, the principal filtering domain has shifted from ports to addresses. Many servers are programmed with wildcard listening sockets, and the ensuing address promiscuity of these sockets poses severe challenges for address based filtering. The problem with protecting servers is that after an initial connection, the server's firewall no longer follows the ports of the packets coming and going to the server. If an initial connection is permitted from a host address to a server, then all subsequent packets between the two addresses will flow freely. This can result in inadequate control, as shown in the following example.

Again, consider a server that wishes to permit incoming access to `ftpd` services from the network, but not `telnetd`. A devious client could start an FTP session to the server, which the server firewall permits, and once the firewall enables the rule $\{$Devious Host.* $\longleftrightarrow$ Server:0.*$\}$ the client can now connect to the server's `telnet` port from the network in violation of the intended security policy.

One obvious solution to this problem is to use an embedded server access control mechanism (something like TCP Wrappers [Ven92]) to regulate what will be served to the outside on the *Ethernet:0* address. After breezing by the firewall with an attempted `telnet` request, the previous paragraph's devious client could be refused a connection to `telnetd` according to its source address. Similarly, the `inetd` super-server has a address option that will bind the server to a particular address, and this should provide adequate control for services running under `inetd`.

A simpler approach would be for the firewall to continue to monitor connection attempts to well-known services and filter accordingly. This deviates from the concept of filtering solely based on address, but not necessarily to the point of continuously following application level protocols. In our example, connection attempts to all privileged ports other than the ftp ports would be blocked at the firewall, and the ftp could proceed.

Clearly, relying solely on a packet filtering approach to provide server security becomes more unworkable as the variety of externally-accessible services grows. As the number of services increases, sensible packet filter rules become more difficult to specify, and the likelihood of consistency problems caused by interaction effects increases correspondingly. This is an old and familiar firewalling problem, traditionally best solved with a mixture of strategies combining packet filters and proxies. We make no claims that TARP Addressing solves any of these complexities, only that there should be cases where firewalls can use the addressing scheme to provide equivalent security without following application level protocols.

12 Limitations

Server facilities using a multi-homed server strategy to serve a variety of domains from the same host [Ste90, p. 93], may not be a good match for TARP's preference to serve from a single fixed address per interface. At the least, they seem incompatible with `inetd`'s wildcard listening sockets and would seem better off if the relevant servers bound only to the the address they are serving. This is more a limitation of BSD, rather than TARP addressing, *per se*.

The security concepts make no contributions to solving problems of inside threats, but this is a recognized limitation of firewalls in general ([Cha92], [CB94]).

Our implementation cannot support more than a single TARP address per interface, and doing so would require extensive kernel modifications. This is for two reasons. First, when faced with an outgoing address decision, the kernel already knows which interface to use, and the implementation determines the Ethernet address of the outgoing interface for use in address computation. Even if we had an extra, valid Ethernet address to use, it would be difficult for the kernel to determine when to assign it. The second set of problems is that all of the problems of multi-homed BSD servers described above would occur.

13 Applicability to other Operating Systems

Although our implementation was built on the UNIX notion of "process groups", it is clearly not necessary to do it that way. Two preconditions are necessary for a substitute mechanism.

First, and most obvious, there has to be some way to generate a 16-bit number not currently in use. Clearly, a counter and an in-use list will suffice. The harder problem is somehow assigning this number to a program or group of related programs. All "related" programs—we define "related" as meaning "they would expect to have the same IP address"—must somehow be linked to this number. The notion of a process group in UNIX captures these semantics quite well; the fact that process groups have the right sort of number is simply a happy accident.

14 Conclusions

We have shown TARP to be a useful new possibility made available by IPv6's Addressing Architecture. Using TARP addresses greatly simplifies firewalling decisions for the machines protecting either clients or servers. Furthermore, by partitioning the address space into client and server addresses and only configuring client network addresses as needed by client activity, it looks to provide particular advantages for hosts acting mostly as network service consumers.

15 Acknowledgments

A number of people involved in the IETF's IPv6 effort (Steve Deering, Thomas Narten, Erik Nordmark, Keith Moore, and others) provided valuable advice. Jon Crowcroft noted that similar facilities could be built on top of Linux's IP Address Masquerading facility. Also, great thanks are due to John Ioannidis for helpful and generous use of the computer laboratory facilties he maintains. We gratefully acknowledge the code, excellent assistance and cheerful bug fixes provided by the *kame* project.

References

[Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[Bel94] S. Bellovin. On many addresses per host. Request for Comments 1681, Internet Engineering Task Force, August 1994.

[Bel99] Steven M. Bellovin. Distributed firewalls. *;login:*, pages 39–47, November 1999.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, 1994.

[Cha92] D. Brent Chapman. Network (in)security through IP packet filtering. In *Proceedings of the Third Usenix* UNIX *Security Symposium*, pages 63–76, Baltimore, MD, September 1992.

[Cra99] Matt Crawford. IPv6 node information queries, 1999. Work in progress.

[GS96] Simson Garfinkel and Gene Spafford. *Practical Unix and Internet Security*. O'Reilly, Sebastopol, CA, 1996.

[HC98] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments 2409, Internet Engineering Task Force, November 1998.

[HD98] R. Hinden and S. Deering. IP version 6 addressing architecture. Request for Comments 2373, Internet Engineering Task Force, July 1998.

[Ins97] Institute of Electrical and Electronics Engineers. Guidelines for 64-bit global identifier EUI-64 registration authority, 1997.

[Joh93] M. St. Johns. Identification protocol. Request for Comments 1413, Internet Engineering Task Force, January 1993.

[KA98] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments 2401, Internet Engineering Task Force, November 1998.

[KAM] http://www.kame.net.

[MDM96] J. McCann, S. Deering, and J. Mogul. Path MTU discovery for IP version 6. Request for Comments 1981, Internet Engineering Task Force, August 1996.

[ND01] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in IPv6. Request for Comments 3041, Internet Engineering Task Force, January 2001.

[NNS98] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for IP version 6 (ipv6). Request for Comments 2461, Internet Engineering Task Force, December 1998.

[Pip98] D. Piper. The internet IP security domain of interpretation for ISAKMP. Request for Comments 2407, Internet Engineering Task Force, November 1998.

[Ste90] W. Richard Stevens. UNIX *Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990.

[TH95] S. Thomson and C. Huitema. DNS extensions to support IP version 6. Request for Comments 1886, Internet Engineering Task Force, December 1995.

[TN98] S. Thomson and T. Narten. IPv6 stateless address autoconfiguration. Request for Comments 2462, Internet Engineering Task Force, December 1998.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third Usenix* UNIX *Security Symposium*, pages 85–92, Baltimore, MD, September 1992.

[VET+97] P. Vixie, Ed., S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system (DNS UPDATE). Request for Comments 2136, Internet Engineering Task Force, April 1997.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, Reading, MA, 1995.

[Ylo96] Tatu Ylonen. SSH – secure login connections over the internet. In *Proceedings of the Sixth Usenix* UNIX *Security Symposium*, pages 37–42, July 1996.

[ZCC00] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. O'Reilly, Sebastopol, CA, second edition, 2000.