PL2 Phase 1: Anonymous Cloud Routing

Dino Dai Zovi <ghandi@cs.unm.edu>
Antony Saba <awsaba@cs.unm.edu>
Trek Palmer <tpalmer@cs.unm.edu>

February 8, 2002

Abstract

Programmable Landscape 2 (PL2), is a complete networking and programming suite to enable completely pseudonymous, platform independent, distributed interaction. The networking portion of PL2 is intended to utilize the existing Internet infrastructure. The networking layer utilizes encryption to ensure point to point security. To ensure the capacity for interactive communication, each networking node is identified by a unique “pseudonym”, which allows explicit point-to-point connections to be established without ever associating a specific network address with the pseudonym. Communicating through the networking layer will be the other two major segments of the PL2 architecture: The Distributed Scheme Interpreter, and the Site-Describing Meta Language. Because each successive layer will be communicating with the lower layer through an API, different scheme interpreters can be using different networking layers without any compatibility being lost. Because the scope of this project is quite large, we have split it up into several phases: phase 1 is the implementation of the networking layer and the formulation of its associated API.

1 Network Layer Architecture

The overall layout of PL2 is a collection of nodes which communicate over encrypted link-to-link and end-to-end connections.

1.1 Addressing

Each node is identified by a 128-bit ACR address. The address, or node identifier, is formed by computing a 128-bit cryptographic hash of the node’s public key for unique and more compact addressing.

1.2 Link-to-Link Communication

Each node is connected through symmetrically encrypted links to its neighbor nodes. The links are formed by performing an initial key agreement to negotiate the secret key used in a symmetric cipher to encrypt all further communication. It must be stressed that each node will only have a direct connection to its neighbors, and all outgoing connections are only known by the neighbor that they go through.

1.3 Routing

Actual routing is not done in a completely random fashion, as PL2 is meant to support interactive services; some concessions to efficiency must be made. However, only next-neighbor routes are known definitely and each node can only send a particular packet towards the neighbor most likely to be closer to the destination. To accomplish this each node will maintain some limited life cache (like a fixed size circular queue), with some mapping from destination to neighbor. This mapping is established by a route discovery process, in which a random number from the client is exchanged with the server to establish a routing bias for that session. Each node will effectively have two caches, a potential route cache and a real route cache. When a route discovery is initiated the potential router stores the information in the potential cache. If it receives another packet with that route ID from the same source, the router can assume that a temporary route has been established, and can then move the routing information from the potential cache to the real cache. Then, if the turn-around time in the potential cache is less than in the real cache, unused routing information will not clutter up the actual routing cache.

1.4 Route Obfuscation

In order to further obscure the route, each server acting as a router will, upon receipt of a route discovery
packet, generate its own route discovery packets using a random number generated from the received packet (perhaps by using a hash of the message as a seed for a random number generator). This means that it is impossible to deterministically guarantee that a route discovery packet leaving a host was the mate of one that arrived previously, as it is indistinguishable from a routing packet originating from the server in question. Thus, all effective routers will maintain a list of temporary routes from one neighbor to another. Therefore in order to identify an entire route all nodes along the route (and their neighbors) will have to be constantly monitored. So, in a route of length \( N \), with an average number of neighbors per node of \( K \), all \( N \times K \) nodes would have to be compromised in order to be able to capture a single route. But, for a perspective intruder this is not enough. Because the routes are changing on a regular basis, all potential routes will have to be monitored. Therefore, for a route of average length \( N \), with an average number of neighbors per node of \( K \), the number of compromised nodes will be on the order of \( K^N \), to contain the combinatorially explosive number of potential routes.

1.5 End-to-End Communication

End-to-End communication is possible through two network transports: unreliable datagrams and reliable sequenced connections.

![ACR Packet](image)

**Figure 1:** ACR Packet

<table>
<thead>
<tr>
<th>16 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

#### 1.5.1 Datagrams

The simplest network transport involves simple datagram routing. Several simple higher-level protocols (for routing, key distribution, etc) are built upon datagram passing. The datagram packet is illustrated in Figure 1. A datagram packet header simply specifies the packet header version, protocol, time-to-live, data length, and destination. The basic packet header requires 32-bytes. The packets are routed solely by their specified destination address.

Each packet contains a fixed-length (1024 bytes) data segment. The data length specified in the header describes how much of the segment is used for real data. The rest of the data segment is filled with random bytes to further obscure communication.

When using pure datagram packets (no portion of the data segment is used for upper-protocol headers that must be preserved in plaintext), the entire data segment is asymmetrically encrypted to the destination host. This will be done by generating a random symmetric algorithm key, encrypting the data with that key, and including the key, encrypted with the destination’s public key, at the end of the data segment.

#### 1.5.2 Sequenced Connections

A sequenced connection begins with a route discovery process. A series of packets are sent out from the calling node to discover the shortest route to the destination. The route discovery packets are broadcast to all neighbors. Each route discovery packet is identified with a unique route identifier. As a packet is relayed, the intermediate node records the route identifier with the link that it received the packet on. When the destination node receives a route discovery packet, it sends a connection acknowledgment packet using the route identifier from the received packet. This ensures that the connection acknowledgment packet follows the most efficient route back to the node that initiated the connection. From here on, all communication will proceed over this route. At arbitrary times (or when an intermediate link goes down), the route will be rediscovered. Sufficiently paranoid nodes may set a route-decay time, after which the route will be forcibly rediscovered.

The sequenced connection protocol will require additional header fields (such as the route-discovery identifier). The header will be divided into two sections, a public (preserved as plaintext) header, and a private header (encrypted to the recipient). The private header and data section will be encrypted to the recipient using the same scheme as the datagram packet.

1.6 Data Fountains and File Streaming

File sharing and file propagation require completely different attributes from a connection. Responsiveness and reliability are unnecessary insofar as encoding permits error correction and file reconstruction even through a shaky connection. Essentially the only requirement for an effective file sharing network is **bandwidth**. Anonymity can be preserved as long as no negotiations between client and server have
to take place in order to share a file and move it around the network.

The PL2 solution is to use a data fountain. A data fountain is essentially a continuously broadcasting data archive, that anyone can tap without having to notify the server. If the server were to concatenate all shared files in some well-known archive format and then rigorously encode it with error checking and correcting codes (ECC) (bloating it significantly, upwards of 30%), it could then just broadcast data to the world, and anyone listening could recieve the files being shared.

This approach to streaming data has several advantages:

- Because no communication must occur between file server and client, the connection is completely anonymous.
- Because of the ECC encoding of the data, the fountain packets can be recieved in any order and missing data reconstructed. This means no overhead with synchronization between server and client (unlike current streaming data schemes), and that a partial archive can be completed by receiving a sufficient number of packets from the fountain to reconstruct missing or partial files.
- Because the cost of a physical network connection is the same regardless of whether its being used or not, an intelligent routing scheme would fill any gaps in traffic load with data fountain packets. This provides the sort of bandwidth necessary for large-scale file sharing without degrading interactive connections of their responsiveness.

2 Current Implementation

The current ACR implementation is written in Java for portability and because of the utility of the Java Cryptography Extension (JCE) frameworks. As the design and implementation of ACR stabilizes, it will be re-implemented in a faster language such as C, Objective-C, or C++.

2.1 Node

The Node module represents the state and functions of a node on the ACR network. On first use, the Node module generates the various keys needed for link connections and node identification. Both Diffie-Hellman (for key agreements) and RSA (for end-to-end encryption and node identification) are generated. The keys are stored in a DES encrypted file, protected by the user’s passphrase.

Once the Node is started up, it may begin listening for TCP connections or connect to other Nodes. When either a connection is received, or a connection is initiated to another Node, control is passed to an appropriate Link module to perform the connection handshake and control communication.

2.2 Links

The link modules perform the ACR link-level protocol handshake and performs the Diffie-Hellman key agreement protocol [7]. The key agreed on by both nodes is used to encrypt the TCP socket connection using the Blowfish cipher [8]. After the link encryption is established, a thread is launched to listen for incoming packets on the link. Upon receipt of a packet, it is placed in the Node’s recieved packet queue (described below).

2.3 Packet

The packet class is a simple representation of a link-level packet. It is instantiated directly from a link, or created for data transmission by the Node. It provides simple accessors and mutators for packet fields.

2.4 Router and Queues

When the node needs to transmit a packet, which will be done regularly for security purposes (see below), it requests a packet from the NodePacketQueue. This queue sorts the packets according to type. Normal data packets are mixed by being randomly placed on different queues. Data fountain packets are placed in a special queue for the data fountain. All noise packets are discarded. When the queue is asked for a packet, it first tries to send a normal data packet by dequeuing one of the normal data queues. If that fails because the chosen queue is empty, the Node then tries to dequeue a data fountain packet. If there is no data fountain packets in the queue, a noise packet is generated and sent off.

In this way, the connection is being constantly used, guaranteeing the speedy propagation of data fountain packets as well as confusing attempts at traffic analysis.
The router module (which each Node possesses), is responsible for taking packets off the received queue, examining them, and then either placing them in the NodePacketQueue for retransmission or storing them for processing. A packet is examined by looking at the destination, if it matches the Node’s pseudonym, the packet is then stored for later processing. If the packet is noise, or it’s TTL is \( \leq 0 \) the packet is discarded. All other packets are placed in the NodePacketQueue. Packets taken off the NodePacketQueue are then passed to the Router’s dispatcher thread, which will consult the internal routing tables to determine down which link to pass this packet.

### 2.5 Simulation

The Simulator utilizes simplified versions of the Node and Link class. Because the simulation (in its current incarnation) is intended to simulate only routing, the SimNode and SimLink classes contain no encryption code. The simulator constructs a pre-specified number of Nodes and then assigns a number of connections to other randomly determined Nodes. Each node then generates empty packets to random destinations and the simulation records the number of hops required to reach other nodes as well as the number of dropped packets.

### 3 Attacks and Defenses

#### 3.1 Traffic Analysis

Traffic analysis can be used to easily determine the source and destination of packets in current networks. Much of the PL2 networking infrastructure was designed to foil such analysis. Primarily, all packets are fixed size, so no analysis attempting to flood a Node with exceptionally large or small packets will work. Such attacks will simply cause a large or small number of fixed size packets to be generated. Also the order of retransmission of packets cannot be determined from their arrival order. The PacketQueue performs mixing which should rearrange the received packets somewhat. Also, the per Link encryption ensures that it will be very difficult to extract the destination address (which would be the only real packet identifier). This encryption will also ensure that it is effectively impossible to discern a retransmitted packet from one generated by the Node itself.

#### 3.2 Packet Known-Plaintext Attacks

Because the packets are in a standard format, link-to-link packets may be vulnerable to known-plaintext attacks by an eavesdropper on the network. In particular, each packet begins with the same 4-byte header version identifier followed by a 4-byte protocol identifier with very few possible values. Combined with the fact that all packets are of constant length (to prevent traffic analysis), this can become a pattern for a known-plaintext attack to determine the secret key used for link-to-link communication.

However, the damage to the network as a whole caused by this attack is minimal. The data segments are still encrypted to their final destination, so a further compromise of the encrypted message would be necessary to recover private data. Cracking the link between two nodes would give an eavesdropper more information about the flow of packets only between those two nodes. To identify the actual location of a node would require the compromise of the majority of the intermediate links in use between the communicating nodes.

### 4 Future Enhancements

#### 4.1 Movable Packet Headers

To protect against the packet known-plaintext attacks described above, the link-to-link packets should allow for arbitrary and random placement of header information inside the packet. Each intermediate node would have to search the entire packet for the header information.

#### 4.2 Dynamic Routing Policies

To allow for easier routing policy experimentation, all routing decisions will be externalized to a “routing policy object”. This object can be dynamically updated on a running node or replaced at build-time.
4.3 Background Noise Generation

The ACR suite (like most other cryptographic applications) uses a great deal of randomness to provide security functions. On some modern system implementations, a system-wide entropy pool is used to provide randomness to the random number generator. To protect against entropy depletion, a background thread will continually fill the fixed-size noise queue.