Dynamic Distributed Key Infrastructure DDKI

Tunnel Project with GateKeeper and KeyVault

By
Albert Meyburgh
A00214314
COMP 8045 & COMP 8046
18 Credit Practicum Report
Revision 5
For Whitenoise Laboratories Canada Inc.

Project note:

This project examines a dynamic distributed key framework with two separate components: GateKeeper and KeyVault.

The GateKeeper tunneling system could be used on its own to only facilitate the traditional notion of static point-to-point tunnels that would be useful for ISPs, governments, embassies, or corporations.

The KeyVault architecture to distribute session keys based on a distributed key allowing for point-to-point dynamic connections can be applied on other areas apart from the tunnel. There are some implications in implementing a secure tunneling system combined with the KeyVault system. Not only does the system create a secure point-to-point communications layer, but it also provides a way for dynamically adding new GateKeepers to the system without having to copy the key manually to every other client before communication can commence. At the same time it is satisfying the authentication requirement.

Introduction

Whitenoise Labs Inc. has reviewed and accepted my practicum deliverables.

The secure tunnel system is functional with the two proposed applications: The GateKeeper and the KeyVault work together to create a dynamic distributed key environment for TCP/UDP tunneling.

The GateKeeper creates and encrypts tunnels based on simple standard netfilter rules, while the KeyVault facilitates the retrieval of point-to-point keys as required by GateKeepers as they talk to each other.

In short, the system currently facilitates near-transparent, dynamic, encrypted point-to-point communication between networks on a network.

In this report, I shall go over possible alternative solutions, the evolutions of this system from the initial proposal as technical barriers were encountered, give some examples of known information stealing attacks that are ineffective when these tunnels are present, identify some weaknesses in the system that need to be addressed before the product could go commercial, and provide some vectors of approach to overcoming these weaknesses.

Included in the submission you should find:

- Source Code Folder
  - GateKeeper
  - KeyVault
- AllanAlton-Layer2Security.pdf
  - Referenced for layer 2 data link attacks
- Whitenoise Technical Reference.ppt
- Diagrams in Visio Format
  - GateKeeper Process.vsd
  - GK Classes.vsd
  - KeyVault Process.vsd
  - KV Classes.vsd
  - WN Header.vsd
- Software Libraries
  - libnet.tar.gz
  - libpcap-0.9.3.tar.gz
  - qt-x11-opensource-desktop-4.0.0.tar.gz
- DRTCP021.exe (for setting the MTU of the network card)

Body

Company Background

Whitenoise Laboratories Inc. is in secure information technologies. WNL protects Intellectual Property and data at rest and in motion.

The Whitenoise Superkey Encryption Algorithm (WSEA), a new generation symmetric stream cipher, powers our next-generation products. Stream ciphers convert plaintext to ciphertext one bit at a time. WSEA, completely random and non-repetitive, can encrypt never-ending streams of communications traffic. This exciting new advance provides unsurpassed security for a vast array of business-to-business and retail applications, including wireless. Whitenoise Superkey Encryption algorithm is broadly patented.
[WIPO No. WO 2005/076521 A1].

Whitenoise encryption provides the highest level of security with the greatest speed with virtually no overhead and with no latency in telecommunications of all kinds. Whitenoise can be deployed either in software or on silicon. Whitenoise is available for licensing to universities and businesses.

For more information on the Whitenoise Stream Cipher used in this project, please refer to the WN technical addendum PowerPoint included with the submission.

Project Statement

The KeyVault and GateKeeper systems work together to create a layer on any IP based network, like the Internet, that allows communications to remain secure and confidential.

The innovative component of the project has been the development and implementation of a dynamic distributed key system. Traditionally distributed key systems require that a key be delivered through courier or in person to each person that you wish to establish a secure link with—this project has eliminated this encumbrance.

At any time, you can start communicating to someone else that uses this software without having to wait for a distributed key to be delivered.

A distributed key is a key that has been pre-distributed by some manual means to the party involved. This is the most secure method of ensuring key privacy; however this is a problem when new dynamic sessions wish to be established with parties who do not have pre-shared key information.

This project uses that distributed key, not as a key for a point-to-point link, as would traditionally be done, but instead that key is used to distribute encrypted "session" keys to be used for the original intention of establishing secure links of communication.

Distributed keys by their nature, not only allow for the encryption of traffic, but also the authentication of the other party. This is a huge advantage as will be discussed when comparing this system to the PKI, public key infrastructure, system.

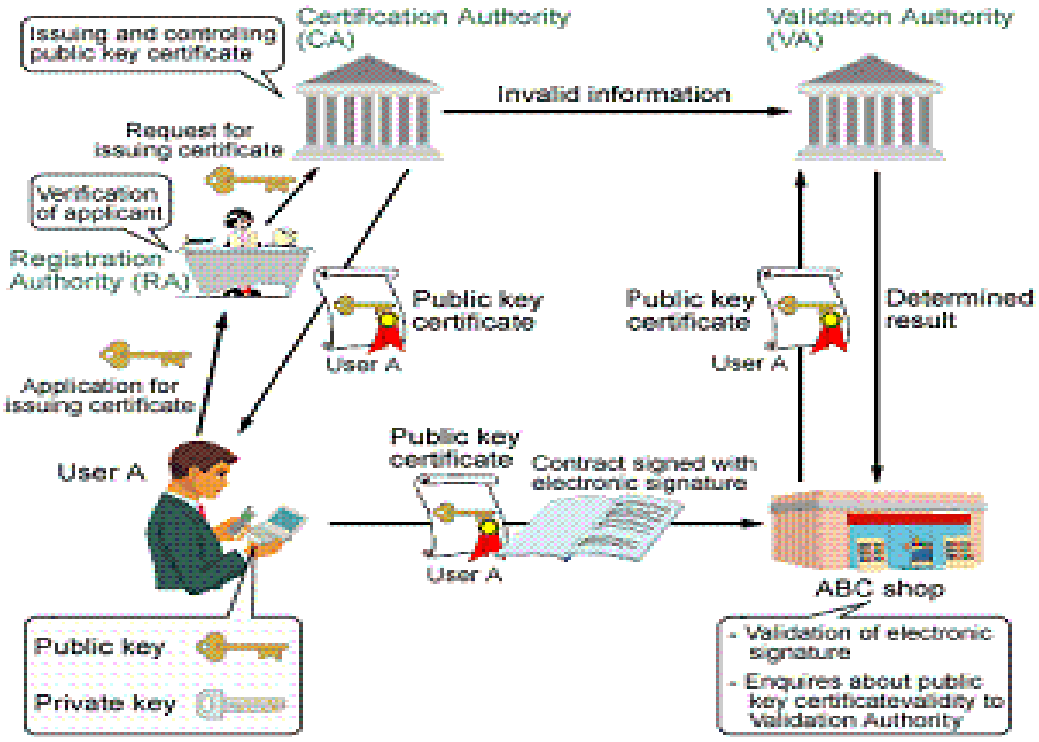
This project is useful to Whitenoise Laboratories, as it showcases the efficiency of their algorithm for large scale commercial applications like cloud computing, teleconferencing, first responding etc.

Alternative Solutions, Solution and Rationale

The most widely used way of providing security online for authentication and encryption is using asymmetrical encryption systems in the PKI design where authentication relies on certificates issued by certificate servers.

The problem with PKI is that it's confusing, requires many components, costs a lot of money to implement, and it relies on slow encryption technologies.

The PKI Architecture Overview



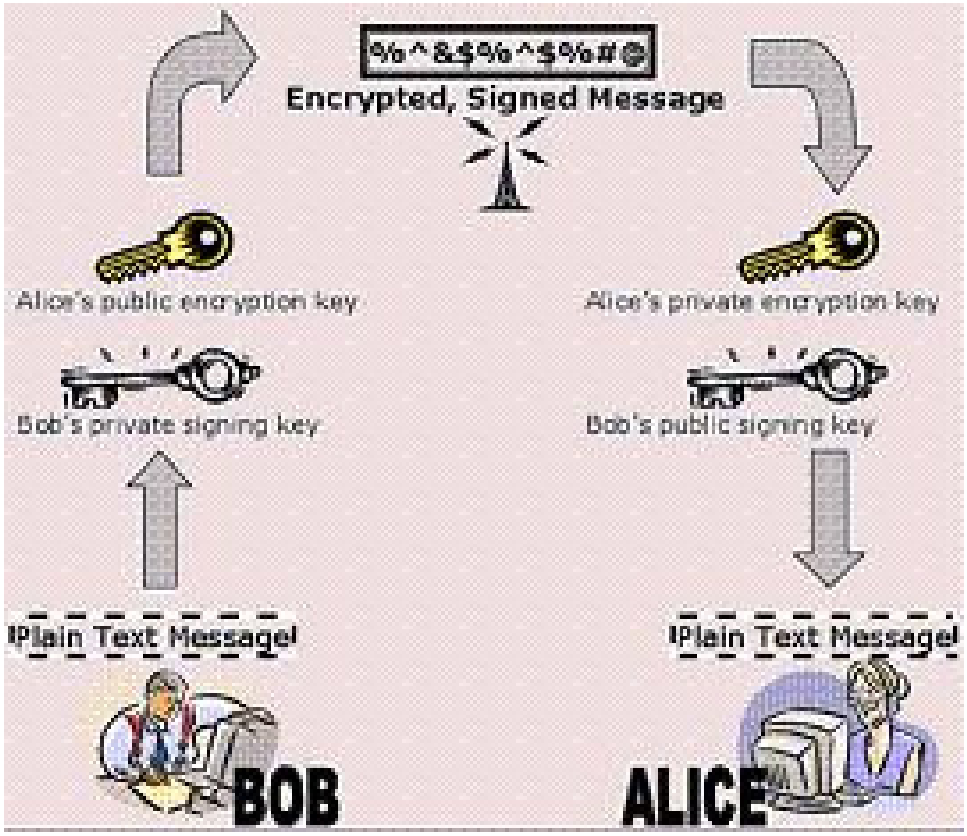
As can be seen from the diagram above, the PKI architecture is confusing, and the security is suspect—there are many known attacks for spoofing certificate authorities and issuing fake certificates.

The overhead of the PKI system is pretty high, not just because of all the steps involved in the architecture, but also their choice of cryptography.

The encryption strength used by the PKI has been called into question recently because:

- Public keys are compound primes and they are always available for attack
- It is very easy to multiply two prime numbers together
- In the ‘70s factorization was infeasible
- New techniques exist to factor compound primes
- Fast computers factor compound primes by simplified techniques like the “sieve” method
- What used to take years now can be done in hours
- Significant strides in prime numbers and factoring theory

The speed of legacy AES on a Pentium 3 is 20 million bits per second; when compared to the 158 million bits per second produced by the Whitenoise super key algorithm it just seems like a better choice for Internet encryption. (See UVIC performance analysis in appendix)



As the above diagram illustrates the method of sending encrypted messages, it's clear that having the public keys available for anyone to see is a dangerous gamble in cryptography, and allows for a point of attack for hackers.

Top 10 Reasons Not to Trust PKI
Bruce Schneier - Carl Ellison – 10 risks of PKI
http://www.schneier.com/paper-pki.pdf

1. Why is the Certificate Authority [CA] trusted?
2. How is a private key on a computer protected?
3. Certificate verification uses only public keys. They are freely available. It uses root keys [these are the keys of those who are doing the authorizing]. A hacker adds his own key to the root keys and can then issue certificates.
4. Certificate association with a name is weak. How many John Smiths are there and how do you tell them apart?
5. Are CA's authorities? They don't decide the kind of information being transmitted.
6. CA's don't bring users into the design.
7. CA plus Registration Authority even weaker.
8. How do you authenticate the Certificate Holder?
9. How do we revoke keys?
10. Do the keys last longer than cryptographically secure?

This list of reasons not to trust PKI is a compelling argument as to why this project, recently named as the (DDKI) Distributed Key Infrastructure, is a necessary evolution in Internet communications security.

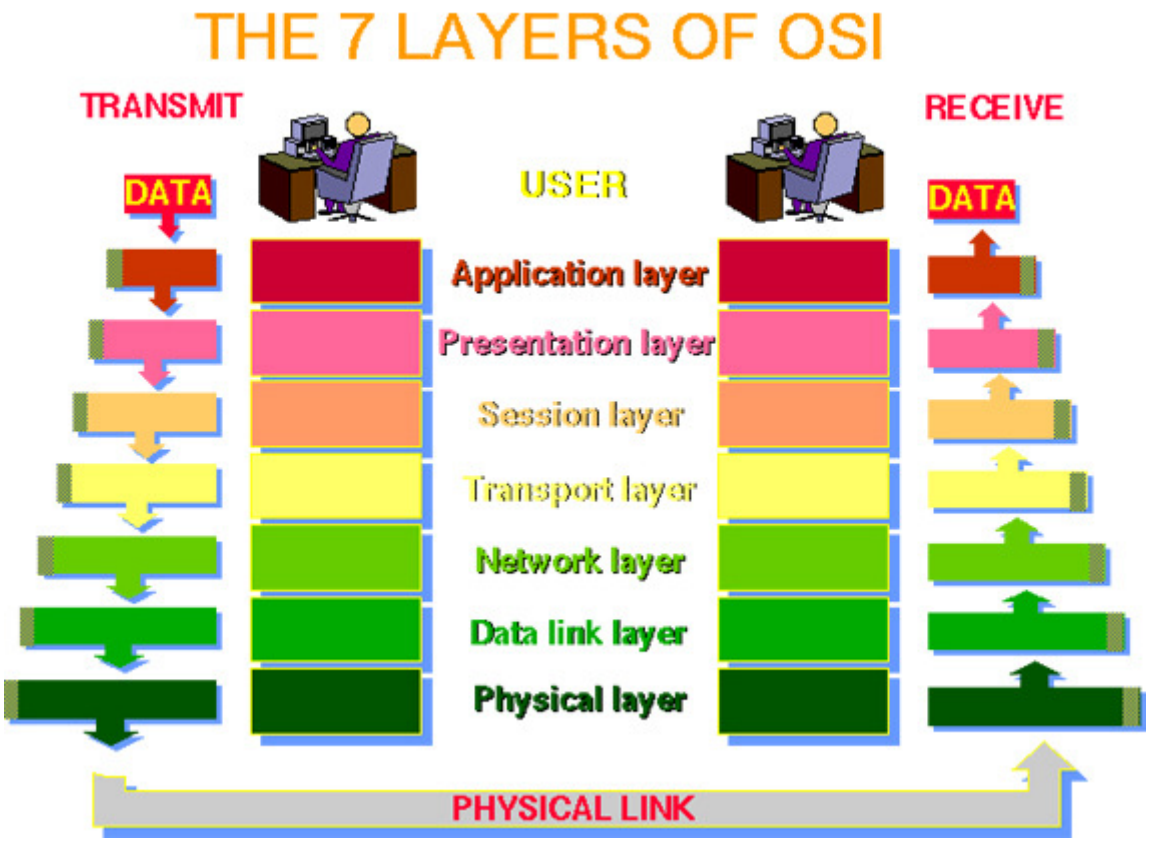
When dealing with hackers abroad, one must remember that if it's possible to break something, then it will be broken. PKI makes it more difficult for hackers to simply just grab information, but the way it does it, makes PKI unattractive to the average user that doesn't even understand the security problem in the first place.

Trust should never be in the hands of the session establishment as with PKI. There should never ever be any cryptographic key information publicly available, as hackers will use everything they can get to eaves drop on their target.

Development Details

Feasibility Assessment

Windows doesn't allow a user to write and read network data at a low enough level for this project to have been feasible on that platform in the time allowed. Linux on the other hand, allows developers to really get down into the guts of the operating system giving one access to the data link layer.



This graphic is taken from [The Abdus Salam International Centre for Theoretical Physics](http://www.ictp.trieste.it/%7Eradionet/1998_school/networking_presentation/index.html).

http://www.ictp.trieste.it/%7Eradionet/1998_school/networking_presentation/index.html

After establishing the ability to read and write from the data link layer in the Linux environment, the project definitely seemed feasible.

Uses of the System

The GateKeeper and KeyVault servers can really be used in any tier of network architectures traveling from IP to IP. Either right from computer to computer, or alternatively, from network to network, or even computer to network—literally any IP to another IP: wired-to-wired, wireless-to-wired, and wireless-to-wireless. It really doesn't matter.

The reason that the system is able to plug anywhere into a network so easily, is because of the fact that the system relies on the data link layer between systems. Some other encryption systems rely on the application level (SSH is an example of this). When the application level is used, the secure tunnel is application specific and needs to be re-integrated with each application that wishes to utilize it such as VOIP, e-mail, secure messaging or web surfing. Using the datalink layer instead, allows immediate integration with every IP based application with no delay. The applications don't even know that the tunnel is there at all. There is no need to involve programmers or do rewrites on existing applications.

One great aspect of this project is that two distinct applications found their way out of it: the KeyVault, and the GateKeeper. Fundamentally these applications could work separately, though they obviously strengthen each other.

The GateKeeper tunneling system could be used on its own to only facilitate the traditional notion of static point-to-point tunnels that would be useful for ISPs, governments, embassies, or corporations.

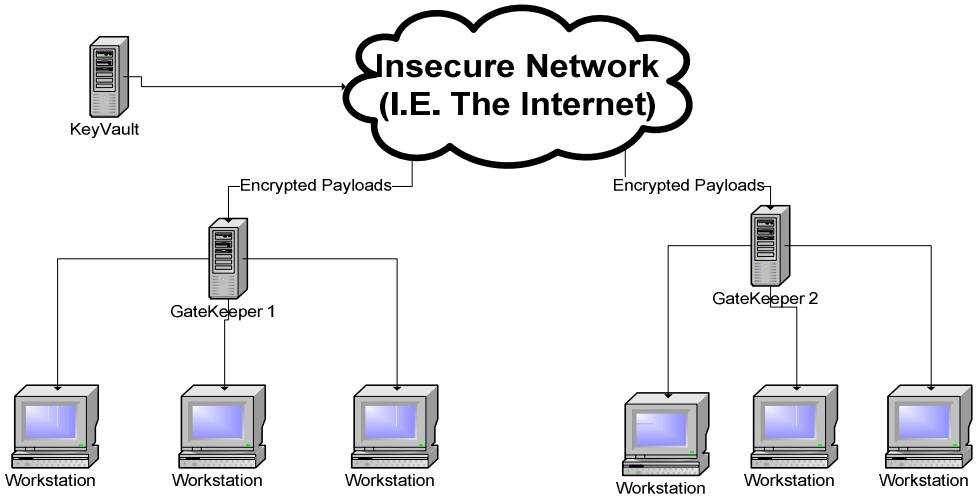
The KeyVault architecture to distribute session keys based on a distributed key allowing for point-to-point dynamic connections can be applied on other areas apart from the tunnel. These other products could include:

- Cell phones
    - Secure calls.
- E-Mail systems
    - Secure and authenticated emails.
- Satellites
    - Military satellite image streaming.
- Peer-to-Peer networks like Bit Torrent
    - Many ISPs filter peer-to-peer network traffic and give users a slower throughput on those connections; encrypted traffic however can't be analyzed.
- And many more!

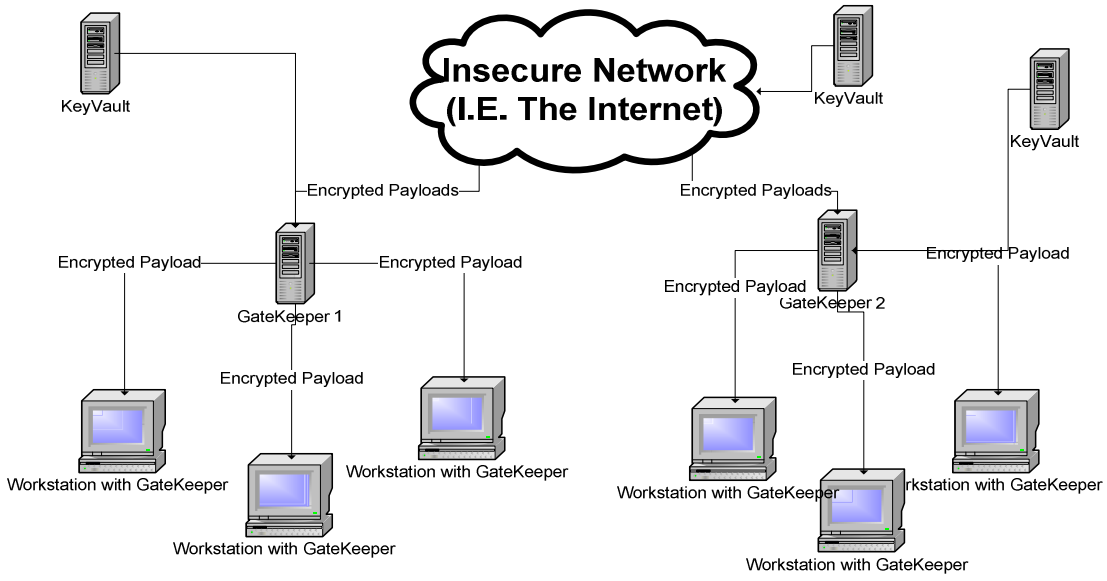
The following diagram illustrates possible configurations that could use the secure communication links using traditional computing networks.

GateKeeper

Scenario 1: All data sent over the internet between designated networks are encrypted; this is ideal for companies sharing who want a virtual private network.



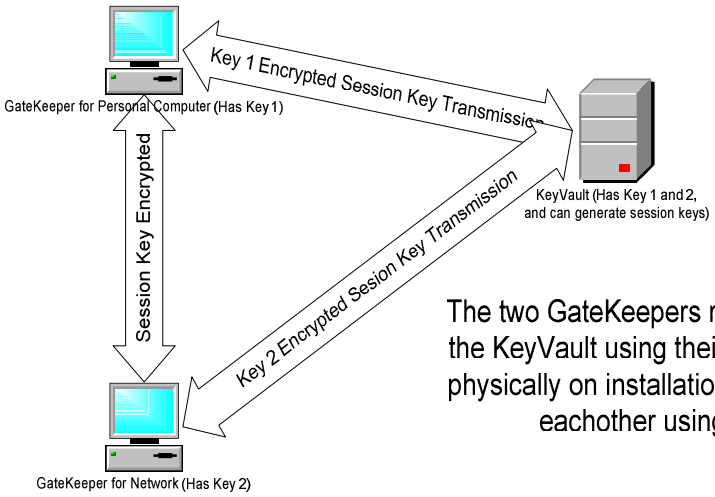
Scenario 2: All data sent between all the nodes on the network is encrypted (including internet); the drawback here is that each computer will have to have the GateKeeper installed.



In scenario 2, each endpoint needs its own Gatekeeper. This can be a simple software upgrade and a one-time secure key distribution.

Sequence Diagrams

Distributed Key Paradigm: Each GateKeeper will have a unique key-pairing with it's KeyVault



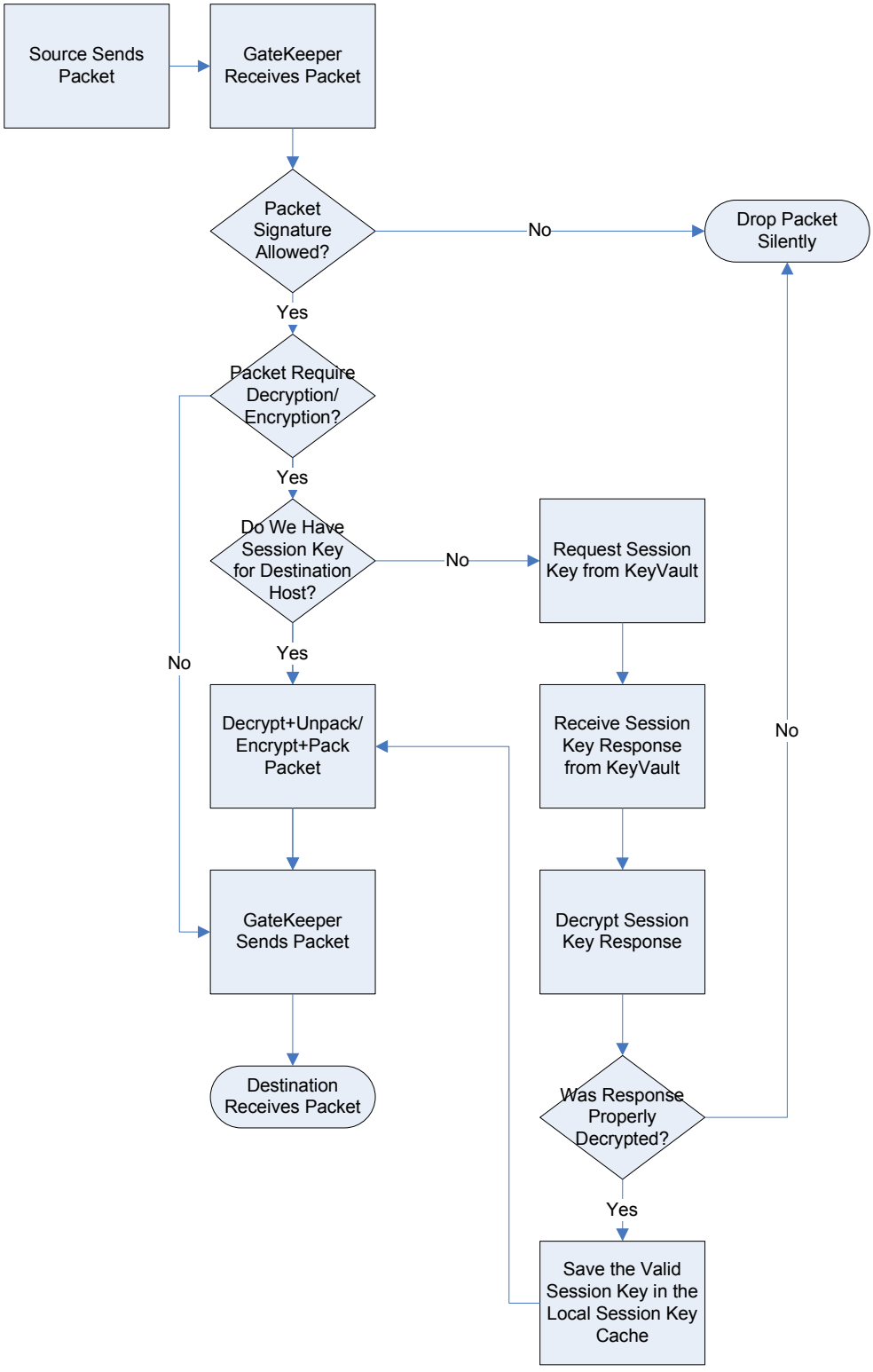
The two GateKeepers request a session key from the KeyVault using their assigned keys (assigned physically on installation), then communicate with eachother using that session key.

As shown in the above diagram, no one GateKeeper can decrypt arbitrary data. When encrypted data needs to be decrypted, only the destination computer can decrypt it, since only the two computers involved in the transmission can obtain the session keys from the KeyVault and since the session keys are encrypted by a per host unique key pairing with the key vault.

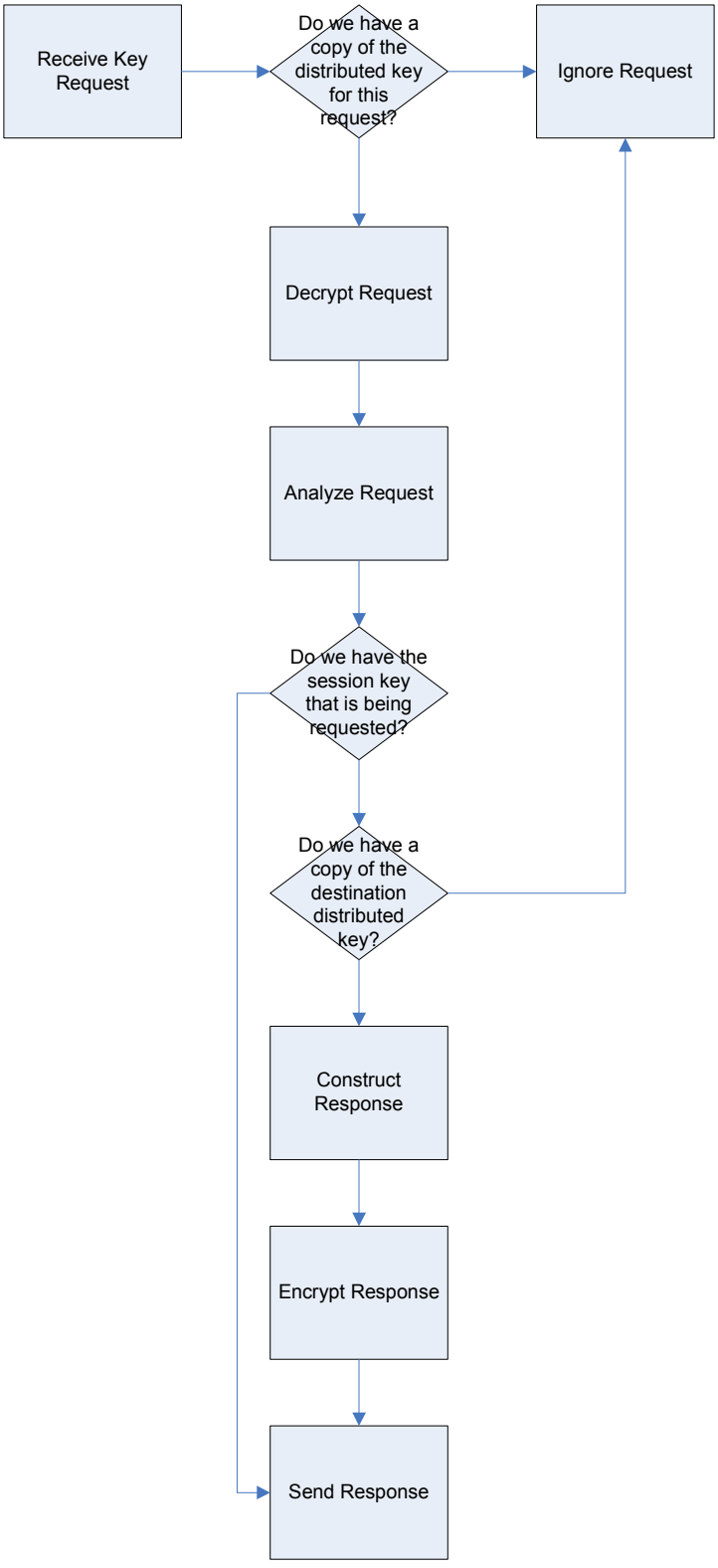
In other words, the GateKeeper client creates and encrypts the request for the session key with the other GateKeeper with its private distributed key that only the KeyVault that holds the session key has a copy of. Only the two GateKeepers involved in the session could request the session key, as their private keys authenticate their requests with the KeyVault.

The sequences of events that drive a secure link start with the GateKeeper on the initiating side, move on to the KeyVault, and finally end at the receiving side. This can be seen in the subsequent diagrams.

GateKeeper Process



KeyVault Process



As seen from the two previous diagrams detailing the flow of events, in both the GateKeeper and the KeyVault, the two systems work together to form the distributed key system in establishing secure point-to-point communication.

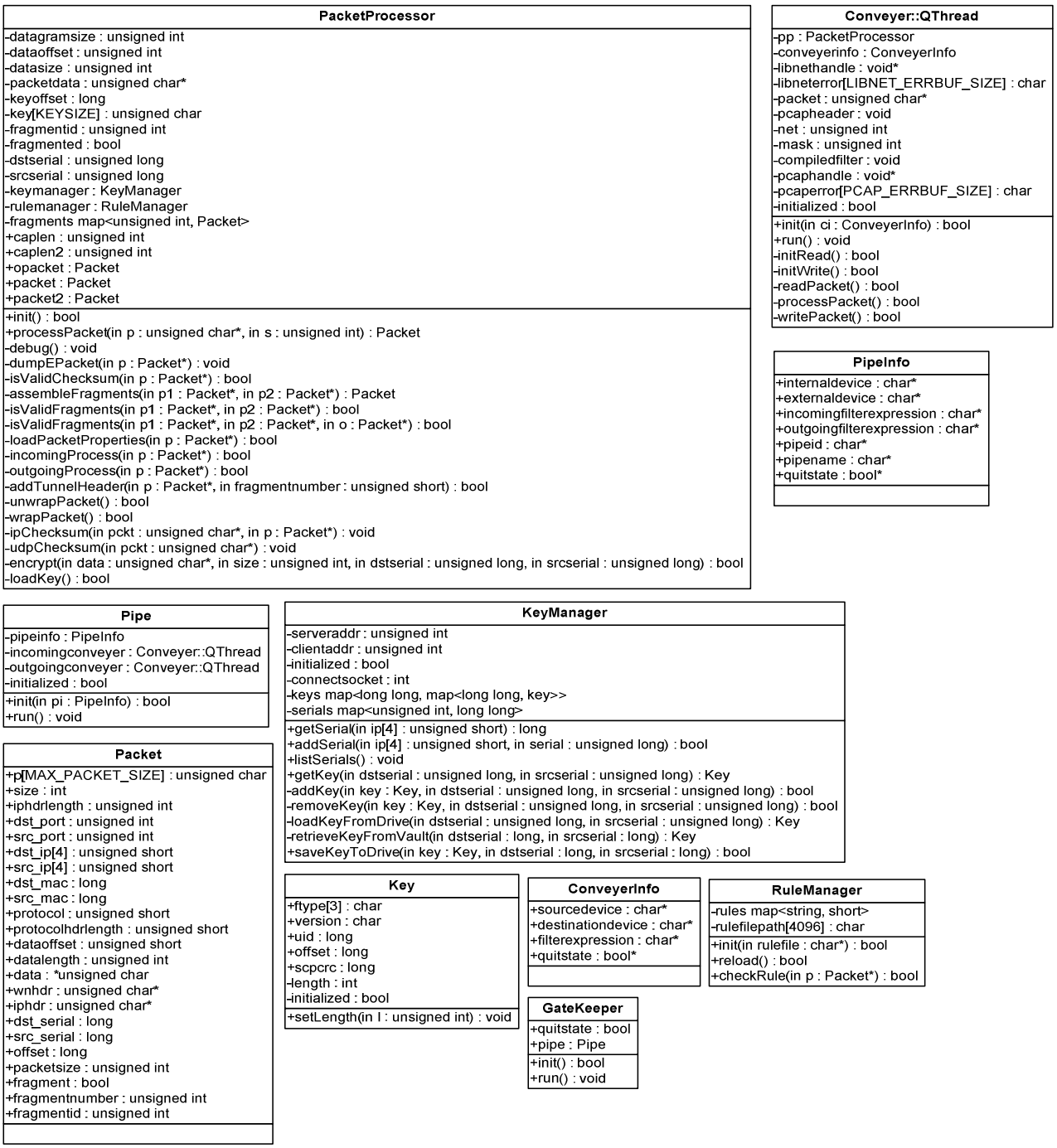
The GateKeeper communicates through tunnels to other GateKeepers using existing cached keys, and retrieves any needed session keys from the KeyVault as needed.

The KeyVault simply receives and respond to key requests.

Class Diagrams

(For a larger view of these diagrams, please see the included MS Visio Files, or just zoom in to 200%+ works just as well)

GateKeeper Classes



The GateKeeper application may consist of one or more pipes, each pipe consists of an incoming and outgoing packet conveyor that is responsible for filtering and encrypting the packets based on the rules from the rule manager in their packet processor, retrieving keys as necessary through the key manager.

KeyVault Classes

Key
+ftype[3] : char +version : char +uid : long +offset : long +scpcrc : long -length : int -initialized : bool
+setLength(in l : unsigned int) : void

KeyVault
-initialized : bool +quitstate : bool* -listensocket : unsigned int -serveraddress -clientaddress -keyvaultip[4] : unsigned short -keys map<long long, map<long, Key>>
+saveKeyToDrive(in key : Key, in dstserial : unsigned long, in srcserial : unsigned long) : bool -getKey(in dstserial : unsigned long, in srcserial : unsigned long) : Key -addKey(in key : Key, in dstserial : unsigned long, in srcserial : unsigned long) : bool -removeKey(in key : Key, in dstserial : unsigned long, in srcserial : unsigned long) : bool -loadKeyFromDrive(in key : Key, in dstserial : unsigned long, in srcserial : unsigned long) : bool -_sendto(in s : int, in buff : void*, in length : unsigned int, in sockaddr : unsigned int, in tolen : unsigned int) : bool -_recvfrom(in s : int, in buff : void*, in length : unsigned int, in sockaddr : unsigned int, in fromlen : unsigned int) : bool +listen() : bool +init(in q : bool*) : bool

The KeyVault application has one main loop that listens for incoming key requests, and fulfills the requests with key responses.

Code Notes

When writing packets, something to note is that the functions are ordinarily not available unless you initialize libnet in advanced mode as such:

```
libnethandle = libnet_init(LIBNET_LINK_ADV,
conveyerinfo.destinationdevice, libneterror);
```

As you can see in the code above the defined value for LIBNET_LINK_ADV is used to initialize the libnet handle in advanced mode and on the datalink layer.

Also when reading packets, something to note about the types of packets you read back are determined by a compiled "netfilter" style expression.

```
pcap_lookupnet(conveyerinfo.sourcedevice, &net, &mask, pcaperror);
pcap_compile(pcaphandle, &compiledfilter,
conveyerinfo.filterexpression, 0, net);
pcap_setfilter(pcaphandle, &compiledfilter);
```

As seen by the code above, we open up a handle to a device we want to read from, compile, and assign a filter to be used. This is where you would integrate the system with IPTables firewall rules. You could for example ignore any traffic that is on ports 21 and 20 to block common ftp services.

In the PacketProcessor class is where the actual Whitenoise header get's appended to the end of the "wrapped" packet, by "wrapped" I mean that the original packet has been re-encapsulated ready to be encrypted. This encapsulation is after all the whole point of using a tunnel since encapsulated can be transformed by encryption without making the packet useless in terms of routing.

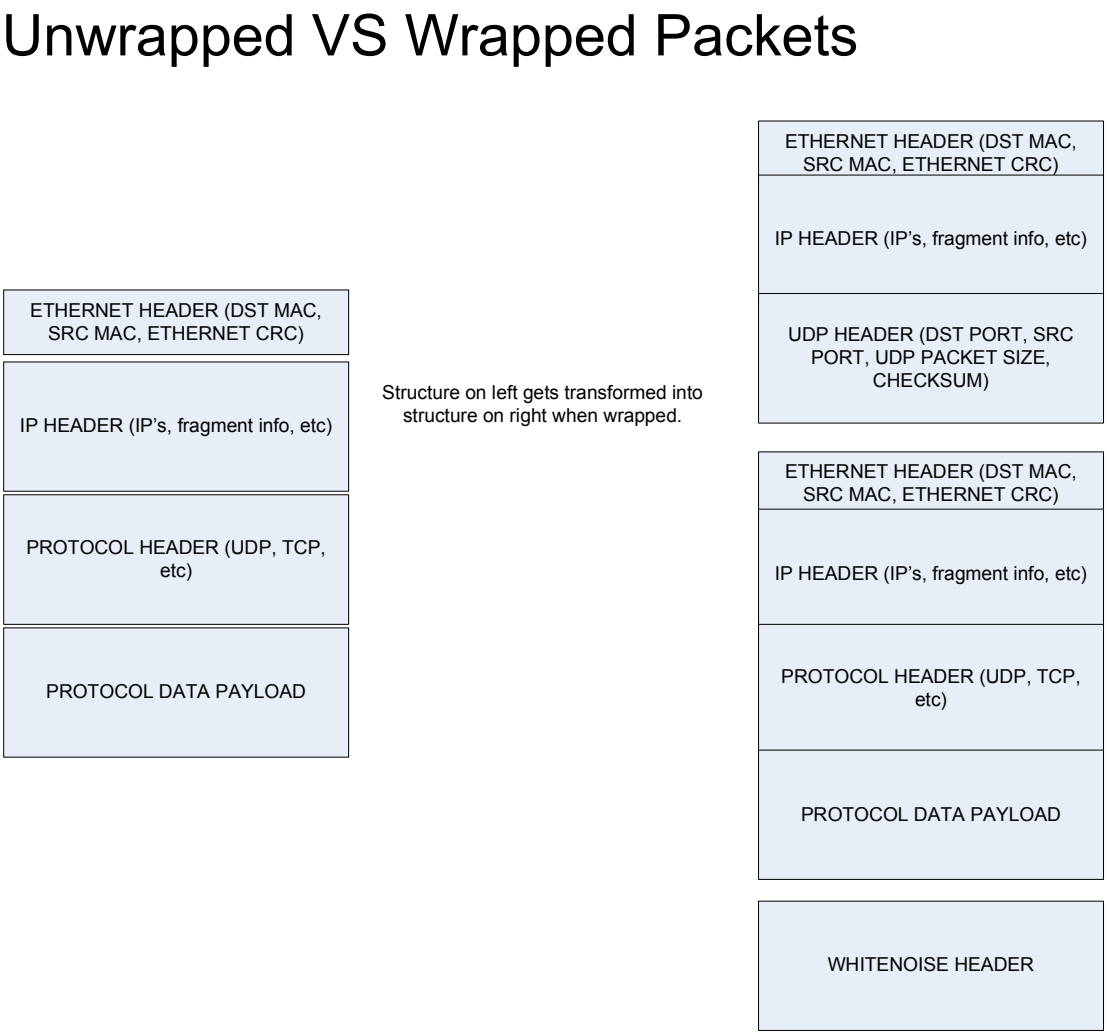
```
// create a UDP headers
*((unsigned short*)(packet.iphdr + packet.iphdrlength)) =
htons(TUNNEL_PORT);                  // src prt
*((unsigned short*)(packet.iphdr + packet.iphdrlength + 2)) =
htons(TUNNEL_PORT);            // dst prt
*((unsigned short*)(packet.iphdr + packet.iphdrlength + 4)) =
htons(UDP_HEADER_SIZE + datalength1); // lngth
udpChecksum(packet.p);

*((unsigned short*)(packet2.iphdr + packet2.iphdrlength)) =
htons(TUNNEL_PORT);                  // src prt
*((unsigned short*)(packet2.iphdr + packet2.iphdrlength + 2)) =
htons(TUNNEL_PORT);            // dst prt
*((unsigned short*)(packet2.iphdr + packet2.iphdrlength + 4)) =
htons(UDP_HEADER_SIZE + datalength2); // lngth
                  udpChecksum(packet2.p);
```

This above code showcases exactly where the custom made UDP header gets created to use in the new encapsulated packet. As you can see there is a call made to the

host to network byte order changing function for short data types, “htons,” for all the information packed into the header bit by bit.

The actual composition of the encapsulated packet is as follows:

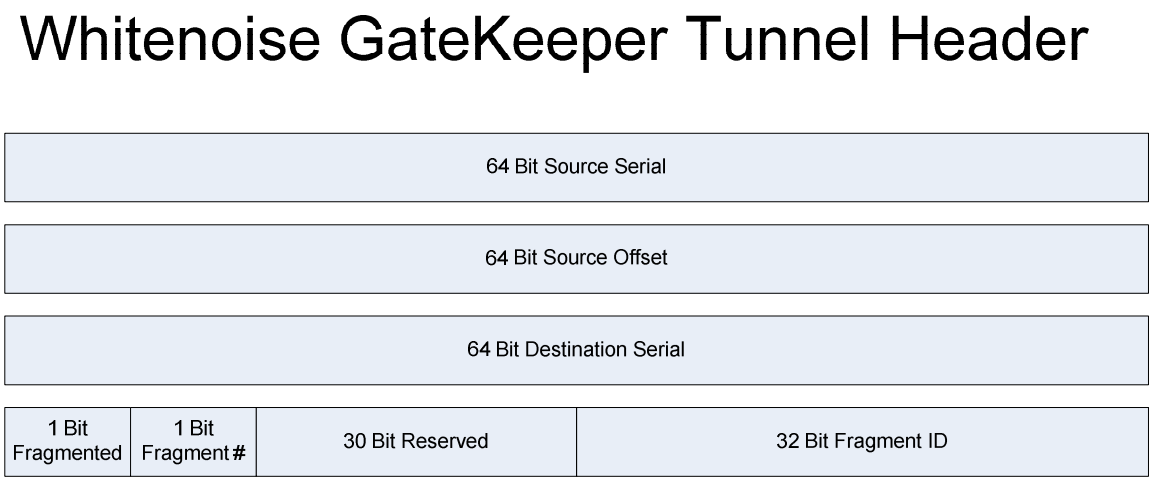


Once the packet has been encapsulated into the new packet with the Whitenoise header, the embedded packet can be encrypted with the appropriate session key.

The reasons UDP packets were chosen to encapsulate the encrypted traffic are twofold:

- UDP is the only common protocol that includes the data size in the protocol, thereby allowing additional headers to be appended
- Since this is a tunnel protocol, if any re-transmission of data is required, the clients can request it, and it’s not needed for the Tunnel to keep track of lost data.

The Whitenoise header consists of information to use the encryption, and some information regarding fragmentation for when the tunnel needs to fragment the data packets due to the MTU being exceeded.



The first serial is the serial of the originating system, the second serial is the destination system serial, and the offset is the offset into the Whitenoise stream that was used to encrypt this particular packet.

The fragmented bit indicates if this is a fragmented tunnel packet, the 1 bit fragment number indicates if it's the first or second fragment, 30 bits have been reserved for a an authentication pad (see the future enhancements section) and 32 bits are used for the fragment id used to distinguish these fragments to other fragments. There is a 1 in 2^32 chance that fragments may have overlapping fragment ids and this would corrupt the re-assembly.

This header, consisting of 256 Bits, plus the additional Ethernet, IP, and protocol headers, in the encapsulated packet, make up the overhead in the overall tunnel system. This overhead is per packet, so if many small packets are sent out, then the percentage overhead is relatively large, however if large packets from file transfers are used then the overhead is very low.

GateKeeper Application Output

In the following output from the GateKeeper application, the tunnel packet fragmentation is shown. A packet that is too large to be transmitted after the Whitenoise header is added to the packet, is split into two fragments. Each fragment maintains the original IP header to make sure the packet gets delivered properly, and has fragmentation information in the Whitenoise header.

```
GateKeeper::init();
Pipe::init(); 1
Conveyer:initread() ether src not 00:00:00:21:a0:1a and ether src not 00:04:E2:D7:32:9C
Conveyer::initwrite()
KeyManager initializing
Conveyer:initread() ether src 00:00:00:21:a0:1a
Conveyer::initwrite()
KeyManager initializing
  incomingconveyer.init(); 1
  outgoingconveyer.init(); 1
GateKeeper::run();
Pipe::run();
Outgoing: Fragmentation=TRUE copying ip and ethernet headers
setting new sizes
splitting up packet into fragments
adding 0xA to wnhdr
adding 0x8 to wnhdr
encrypting data sections of the two fragments
fragment checksums
done creating fragments
       display fragment1:

00 04 e2 d7 32 9d 00 00
00 21 a0 1a 08 00 45 00
03 17 ae 40 40 00 40 11
06 39 c0 a8 01 08 c0 a8
01 04 26 19 26 19 02 e3
00 00 00 4d 00 61 00 74
00 74 00 65 00 72 00 73
00 2e 00 6d 00 70 00 33
00 74 00 00 00 00 00 00
00 00 6a 8e 79 91 cb c5
01 00 6a 8e 79 91 cb c5
01 00 da c3 5e 2f d5 c5
01 00 da c3 5e 2f d5 c5
01 00 00 00 00 00 00 00
00 00 00 10 00 00 00 00
00 10 00 00 00 16 00 00
00 00 00 00 00 10 00 47
00 34 00 37 00 4e 00 4f
00 56 00 7e 00 56 00 00
00 00 00 00 00 00 00 67
00 63 00 6f 00 6e 00 66
00 64 00 2d 00 72 00 6f
00 6f 00 74 00 7c 00 00
00 00 00 00 00 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 00 00 00
00 00 00 00 00 00 00 10
00 00 00 00 00 10 00 00
00 1c 00 00 00 00 00 00
00 10 00 4b 00 42 00 35
00 43 00 34 00 31 00 7e
00 4a 00 00 00 00 00 00
00 00 00 6b 00 65 00 79
```

```
00 72 00 69 00 6e 00 67
00 2d 00 77 00 32 00 37
00 6c 00 6d 00 73 00 00
00 88 00 00 00 00 00 00
00 80 cf 21 b1 37 d4 c5
01 80 79 6f e1 dc d4 c5
01 80 cf 21 b1 37 d4 c5
01 80 cf 21 b1 37 d4 c5
01 d0 34 64 00 00 00 00
00 00 00 10 00 00 00 00
00 20 02 00 00 2a 00 00
00 00 00 00 00 18 00 41
00 32 00 32 00 43 00 4e
00 46 00 7e 00 59 00 2e
00 45 00 58 00 45 00 61
00 6f 00 65 00 33 00 70
00 61 00 74 00 63 00 68
00 2d 00 31 00 30 00 74
00 6f 00 31 00 30 00 31
00 2e 00 65 00 78 00 65
00 60 00 00 00 00 00 00
00 80 a1 28 42 31 d5 c5
01 80 e3 5b ef 4a d5 c5
01 80 a1 28 42 31 d5 c5
01 80 a1 28 42 31 d5 c5
01 00 00 00 00 00 00 00
00 00 00 10 00 00 00 00
00 10 00 00 00 02 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 2e
00 7c 00 00 00 00 00 00
00 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 00 00 00 00 00 00 00
00 00 00 10 00 00 00 00
00 10 00 00 00 1c 00 00
00 00 00 00 00 10 00 4b
00 31 00 5a 00 36 00 51
00 39 00 7e 00 31 00 00
00 00 00 00 00 00 00 6b
00 65 00 79 00 72 00 69
00 6e 00 67 00 2d 00 77
00 57 00 59 00 45 00 73
00 69 00 00 00 70 00 00
00 00 00 00 00 00 3d 5a
24 2f d5 c5 01 00 3d 5a
24 2f d5 c5 01 80 d3 f2
24 2f d5 c5 01 80 d3 f2
24 2f d5 c5 01 00 00 00
00 00 00 00 00 00 00 10
00 00 00 00 00 12 00 00
00 12 00 00 00 00 00 00
00 10 00 5f 00 39 00 46
00 54 00 53 00 43 00 7e
00 4f 00 00 00 00 00 00
00 00 00 2e 00 58 00 31
00 31 00 2d 00 75 00 6e
00 69 00 78 00 01 00 00
00 00 00 00 00 02 00 00
00 00 00 00 00 0a 00 00
00 00 00 00 00 00 00 00
80 47 81 b5 09         end of display fragment1
sending a second fragment
       display fragment2:

00 04 e2 d7 32 9d 00 00
00 21 a0 1a 08 00 45 00
```

```
05 a8 0a a1 40 00 40 11
a7 47 c0 a8 01 08 c0 a8
01 04 26 19 26 19 02 e3
00 00 00 4d 00 61 00 74
00 74 00 65 00 72 00 73
00 2e 00 6d 00 70 00 33
00 74 00 00 00 00 00 00
00 00 6a 8e 79 91 cb c5
01 00 6a 8e 79 91 cb c5
01 00 da c3 5e 2f d5 c5
01 00 da c3 5e 2f d5 c5
01 00 00 00 00 00 00 00
00 00 00 10 00 00 00 00
00 10 00 00 00 16 00 00
00 00 00 00 00 10 00 47
00 34 00 37 00 4e 00 4f
00 56 00 7e 00 56 00 00
00 00 00 00 00 00 00 67
00 63 00 6f 00 6e 00 66
00 64 00 2d 00 72 00 6f
00 6f 00 74 00 7c 00 00
00 00 00 00 00 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 80 e2 a0
94 75 a3 c5 01 00 00 00
00 00 00 00 00 00 00 10
00 00 00 00 00 10 00 00
00 1c 00 00 00 00 00 00
00 10 00 4b 00 42 00 35
00 43 00 34 00 31 00 7e
00 4a 00 00 00 00 00 00
00 00 00 6b 00 65 00 79
00 72 00 69 00 6e 00 67
00 2d 00 77 00 32 00 37
00 6c 00 6d 00 73 00 00
00 88 00 00 00 00 00 00
00 80 cf 21 b1 37 d4 c5
01 80 79 6f e1 dc d4 c5
01 80 cf 21 b1 37 d4 c5
01 80 cf 21 b1 37 d4 c5
01 d0 34 64 00 00 00 00
00 00 00 10 00 00 00 00
00 20 02 00 00 2a 00 00
00 00 00 00 00 18 00 41
00 32 00 32 00 43 00 4e
00 46 00 7e 00 59 00 2e
00 45 00 58 00 45 00 61
00 6f 00 65 00 33 00 70
00 61 00 74 00 63 00 68
00 2d 00 31 00 30 00 74
00 6f 00 31 00 30 00 31
00 2e 00 65 00 78 00 65
00 60 00 00 00 00 00 00
00 80 a1 28 42 31 d5 c5
01 80 e3 5b ef 4a d5 c5
01 80 a1 28 42 31 d5 c5
01 80 a1 28 42 31 d5 c5
01 00 00 00 00 00 00 00
00 00 00 10 00 00 00 00
00 10 00 00 00 02 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 2e
00 7c 00 00 00 00 00 00
00 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 80 70 5c 5f 2f d5 c5
01 00 00 00 00 00 00 00
```

```
00 00 00 10 00 00 00 00
00 10 00 00 00 1c 00 00
00 00 00 00 00 10 00 4b
00 31 00 5a 00 36 00 51
00 39 00 7e 00 31 00 00
00 00 00 00 00 00 00 6b
00 65 00 79 00 72 00 69
00 6e 00 67 00 2d 00 77
00 57 00 59 00 45 00 73
00 69 00 00 00 70 00 00
00 00 00 00 00 00 3d 5a
24 2f d5 c5 01 00 3d 5a
24 2f d5 c5 01 80 d3 f2
24 2f d5 c5 01 80 d3 f2
24 2f d5 c5 01 00 00 00
00 00 00 00 00 00 00 10
00 00 00 00 00 12 00 00
00 12 00 00 00 00 00 00
00 10 00 5f 00 39 00 46
00 54 00 53 00 43 00 7e
00 4f 00 00 00 00 00 00
00 00 00 2e 00 58 00 31
00 31 00 2d 00 75 00 6e
00 69 00 78 00 01 00 00
00 00 00 00 00 02 00 00
00 00 00 00 00 0a 00 00
00 00 00 00 00 00 00 00
a0 47 81 b5 09         end of display fragment2
```

This above fragmentation is not completed, as even though the packets are re-assembling okay, there are still cases of fragmentation not being handled properly resulting in corrupted packets being produced. This corruption is not critical in system operation however, as clients simply have to set their MTU to 1300 in order to accommodate packets what would never need to be fragmented.

In the following output from the GateKeeper Application it's easy to see the key retrieval process:

```
GateKeeper::init();
Pipe::init(); 1
Conveyer:initread() ether src not 00:00:00:21:a0:1a and ether src not 00:04:E2:D7:32:9C
Conveyer::initwrite()
KeyManager initializing
Conveyer:initread() ether src 00:00:00:21:a0:1a
Conveyer::initwrite()
KeyManager initializing
  incomingconveyer.init(); 1
  outgoingconveyer.init(); 1
GateKeeper::run();
Pipe::run();
Incoming: Detecting header
 HeaderFound!
Detecting fragmentation
wnhdr[24]: 112233
failed to open file for reading 0x409fd238retrieve key from fault
creating request: 1:2
checking response to 12
sizeof unsigned long long: 8
key was found on fault responsesize: 50
key found had UID: 69
key found had offset: 10
key found had scpcrc: 10
key found had length: 18
copying key
```

```
done copying key
key on vault
save key to drive path: /tmp/Keys/0000000000000001/0000000000000002.key
```

As can be seen, the GateKeeper receives a packet, realizes it doesn't have the key in the local memory, or hard disk cache, and so it requests it from the KeyVault and saves it to the local cache.

In the screen output below, the rule system is illustrated a bit. The protocol of the incoming packet is displayed (as it's numeric code) and the rule as to ACCEPT / DROP / ENCRYPT is shown as well:

```
GateKeeper::init();
Pipe::init(); 1
Conveyer:initread() ether src not 00:00:00:21:a0:1a and ether src not 00:04:E2:D7:32:9C
Conveyer::initwrite()
KeyManager initializing
Conveyer:initread() ether src 00:00:00:21:a0:1a
Conveyer::initwrite()
KeyManager initializing
  incomingconveyer.init(); 1
  outgoingconveyer.init(); 1
GateKeeper::run();
Pipe::run();
 $ <LPP>PMIHPDS</LPP>
================
Incoming:6 ACCEPT ← as you can see here is an incoming 6/TCP packet market to ACCEPT
 $ <LPP>PMIHPDS</LPP>
++++++++++++++++14:0:20

00 0e a6 14 1e 8e 00 00
00 21 a0 1a 08 00 45 00
00 34 df a8 40 00 40 06
d7 5e c0 a8 01 08 c0 a8
01 64 80 2a 00 8b ab 6f
9e b7 55 2a bb 33 80 10
05 b4 6a be 00 00 01 01
08 0a 00 04 7d f7 00 15
29 43
================
 OutgoingData  ACCEPT ←here is an outgoing packet market as ACCEPT
 $ <LPP>PMIHPDS</LPP>
++++++++++++++++0:0:20
ff ff ff ff ff ff 00 00 ←here you can see that this packet is a broadcast packet so
possibly could be filtered.
00 21 a0 1a 08 06 00 01
08 00 06 04 00 01 00 00
00 21 a0 1a c0 a8 01 08
00 00 00 00 00 00 c0 a8
01 04 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
================  The packet below has been marked as ACCEPT_ENCRYPT
OutgoingData  ACCEPT_ENCRYPT <LPP>PMIHPDS</LPP>  Fragmentation=FALSE  CopyIP&EHeader:
ChangeProtocol  ChangeSizeInIPHeader  CreateUDPHeader  CreateTunnelHeader
getserial()19216818
c0a80108
getSerial: c0a80108
getserial()19216814
c0a80104
getSerial: c0a80104
Getting key: 2:1  ←Here the key has to be retrieved from the KeyVault
failed to open file for reading 0x41400a08retrieve key from fault
creating request: 2:1
 $ <LPP>PMIHPDS</LPP>
++++++++++++++++0:0:20
```

```
00 04 e2 d7 32 9c 00 0e
a6 14 1e 8e 08 06 00 01
08 00 06 04 00 02 00 0e
a6 14 1e 8e c0 a8 01 64
00 04 e2 d7 32 9c c0 a8
01 65 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
================
Incoming:11 ACCEPT
checking response to 12
sizeof unsigned long long: 8
key was found on fault responsesize: 58
key found had UID: 23
key found had offset: 10
key found had scpcrc: 7318349394477056
key found had length: 825229312
copying key
```

Most of these debugging output statements are disabled by default, but are still in the code for developers to use. The reason these output statements are suppressed in the final system is for performance reasons as there is no practical need to see the system running while it's running.

Methodology Used

Mixes of methodologies was used—combining the traditional design, implement, test methodology, with iterative prototyping methodology and a spiral approach.

The system was designed, exploring various implementations, subsequently implemented on an iterative prototyping methodology, then tested more fully for stability and performance then finally re-implemented with refinements and re-tested for stability and performance.

Iterations represented discrete additions of functionalities or fixes of problems arising out of previous ones.

Daily mini-milestones were used to ensure project velocity and measurement. Personally I found these small breakdowns of every task a very useful way to plan my personal time and continuously help assess whether the project was still feasible based on the completion rates of these tasks. An example of a milestone would be:

- Figure out why network traffic grinds to a halt when I send out a ping packet.
    - Do a network trace and identify what is happening.
    - Sketch out a solution to the problem
    - Figure out how a solution would fit into the existing code or if re-design needs to occur.
    - Implement solution
    - Re-test

I would set out these small tasks within the time frame allotted for the particular part of the project I was working on based on the original milestone schedule I planned on in the initial project proposal.

Deliverable Comparison

Deliverables Initially Proposed:

Software

- **GateKeeper**
  - Applies the rules for encryption and fire-walling
  - Provided point to point encryption
- **KeyVault**
  - Facilitates the dynamic distributed key assignment to various GateKeepers

Documentation

- Project Proposal
- Design Exploration Document
- Bi-weekly Project Updates and Logs
- Testing Document
- Performance Document
- Practicum Report

The Deliverables Delivered:

- **GateKeeper**
  - Point to point data link layer tunneling system.
  - Uses KeyVault
- **KeyVault**
  - Provides keys to GateKeepers as they request them.
- **Updates and code version history submitted to client.**
- **Proposal Document**
- **Report Document**

As can be seen when comparing the original plan for deliverables to the actual list of deliverables, there isn't much deviation. Testing was done, but not presented in a formal document—though limitations of the system are outlined in the report document as they were identified during testing. A separate performance document also wasn't included as performance numbers are included in the final report.

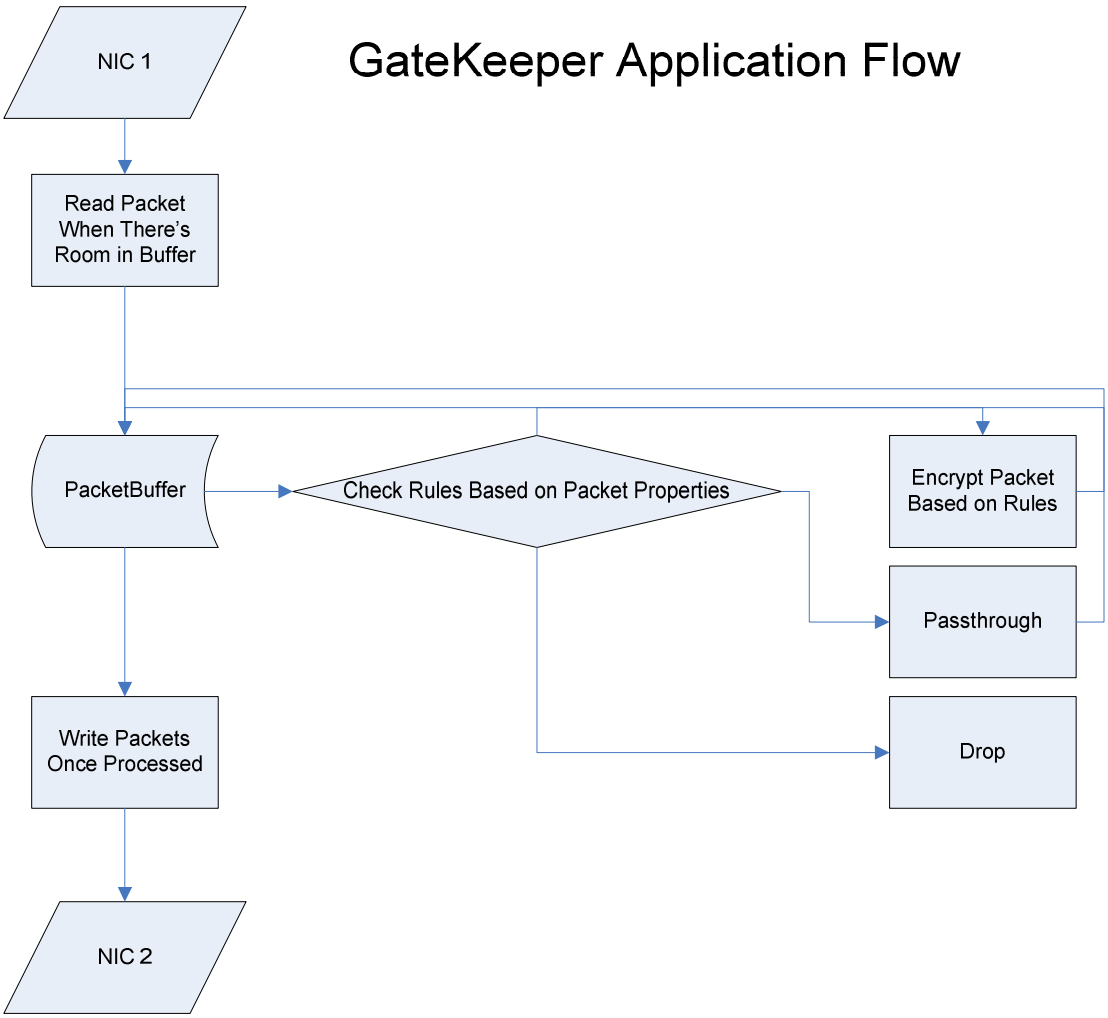
Evolution of the System

Many things changed from the original proposed system. These changes were usually either because the initial design was technically infeasible or because after testing, better solutions were found.

Initially when starting out this project, I had planned to put the Whitenoise tunnel header right after the data section of the actual packet, and just encrypt the whole data section leaving the header in tact for traveling. I had learned however, that this solution wouldn't work since the TCP protocol had no field in its protocol header to indicate the length of the data payload. This meant that I had no way of detecting whether or not another header was present at the end of a packet, or whether the application on the other end could ignore the appended header.

This caused me to switch the design to encapsulate the whole packet (regardless of protocol) into a new custom UDP packet, since the UDP protocol does indeed have a field that specifies how much data the payload carries—therefore allowing detectable appended headers.

The initial system design in the original proposal had a much different threading model. The initial system had a producer/consumer pattern as shown in the subsequent diagram:



This design was initially implemented, but abandoned due to performance issues. As it turns out, the slight delay caused waiting for the one thread to signal the other thread through a semaphore, was enough to cause high latency issues. Ping times through the tunnel were at 300ms on average.

After switching away from the producer/consumer pattern and instead just using "conveyor" threads that read, processed and wrote all in one swoop the ping times fell down to unnoticeable (0ms to 1ms which are typical on a LAN anyhow.)

The CPU usage was also drastically different under the two threading models. Under the producer/consumer threading model, even with the higher latencies, the CPU usage was at 100% usage during a full-blown network transfer. The new threading model dropped CPU usage to 5-7%. This is obviously a huge difference. The high latencies may have been because the CPU was not able to keep up—in any case, the new model was much faster.

Another interesting bug encountered during development involved the GateKeeper flooding the network with nonsense and bringing everything to a grinding halt. This is a low-level network application. I was reading a packet from one network card, and writing it to the other and vice versa. This caused a problem. When I sent out a ping from one client to the other the first response was 4000 ms. This is horrid for a LAN. In fact no network traffic of any kind could go through after that.

I had to do many manual packet traces by hand to figure out that any packet that passed through the tunnel would start to flood out on both network cards due to the fact that I was reading that packet, writing it on the other card, reading it back again, and re-writing it back from where it came from, and so forth in an endless loop. This killed my switch. It caused all network traffic to go through the tunnel and made figuring out the problem even tougher. There were all sorts of traffic in the trace that I wasn't expecting.

I eventually solved the problem with the Berkeley Net Filter, applied on the reading of the packets that filters out the MAC address of the client system on the external network card.

The type of thinking that I had to apply reminded me of the debugging in the ELEX assembly course from CST. It is very low level, and very much step-by-step. Nothing is done for you behind the scenes at this level.

When development switched from the GateKeeper to the KeyVault an interesting issue encountered was the difference in data types sizes from a 64Bit AMD CPU to a 32Bit Intel CPU. In C declaring an unsigned long on a 64Bit machine creates a 64bit number; on the 32bit machine the same data type declaration is compiled to a 32bit value. This caused some issues when the two machines tried to communicate. This problem was fixed by declaring unsigned long longs instead; this would force 64bit data types regardless of platform.

Installation Process

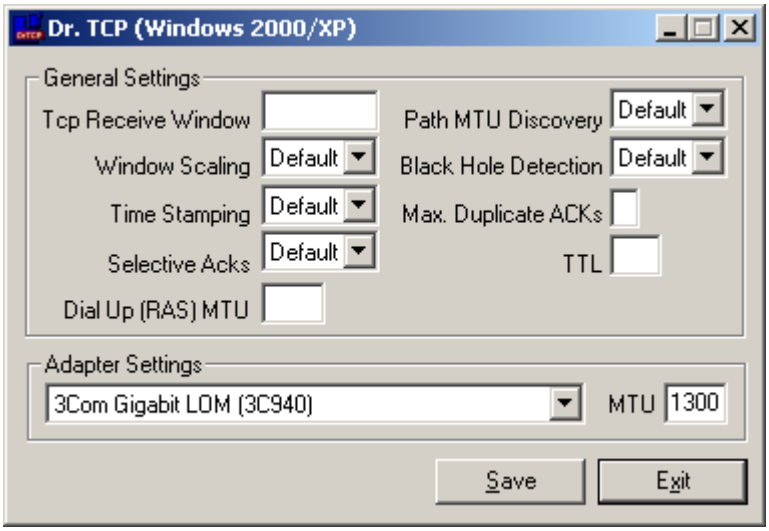
Since there are so many possible configurations for a Linux machine, I recommend using Fedora Core 4 with the full install option to test this project (unless you are an expert in Linux package configuration)

Many Linux configurations by default do not allow a regular user access directly to the datalink layer for security reasons. These applications need to be run as either root or sudo.

Requirements:

- Minimum of 5 computers
    - 1 computer to serve as the KeyVault (with Linux)
    - 2 computers to serve as the GateKeepers (64-Bit AMD Arch. Used in Testing)
        - Configured with Linux (Fedora Core 4 used in test setup)
        - Libnet libraries installed (included in submission as libnet.tar.gz)
        - Libpcap libraries installed (included in submission as libpcap-0.9.3.tar.gz)
        - QT libraries installed (included in submission as qt-x11-opensource-desktop-4.0.0.tar.gz)'
        - 2 network cards
    - 2 computers to transparently use the Tunnels
        - These systems may be configured with any operating system and use any applications.
        - Configured to work on a local area network
        - Network MTU set to 1300 Bytes in Test Setup
        - Use DRTCP021.exe to set the MTU on a windows machine or do man ifconfig in linux to see how to set the MTU

Here is a screenshot of Dr.TCP used to change the MTU to 1300 in Windows.



As you can see in the above screenshot, the MTU text box is in the lower right hand of the GUI. After setting the MTU, please restart your windows machine.

Linux machines do not need to reboot after using ifconfig to set the MTU.

After you have installed all the necessary libraries and compilers on the GateKeeper machines, simply set the included "compile" file to executable (chmod +x ./compile) and execute the compile script. This will compile the included source code and inform you of any missing packages your system requires.

After you have installed all the necessary compilers on the KeyVault machine and set up a "/tmp/Keys" folder, you simply need to set the "compile" file to executable (chmod +x ./compile) and execute the compile script to compile the KeyVault for the platform it's being run on. This script will also tell you of anything else you need to install.

Configuration Process

All configuration of the GateKeeper system needs to be done in the “Include.h” file in the GateKeeper source folder.

The section:

```
// the ip of the keyvault server
#define KEY_VAULT_IP    "192.168.1.100" // put the server IP here!
#define KEY_VAULT_PORT  1357 // put the port you configured the KV as
here! (and make sure your firewall allows outgoing and incoming UDP
packets on this port
```

This needs to be modified to reflect the IP address and port being used by the KeyVault Server.

The sections:

```
// GK2
//#define INCOMINGFILTER "ether src not 00:04:e2:d7:32:9d"
//#define OUTGOINGFILTER "ether src 00:04:e2:d7:32:9d"
//#define MAC 0x0004e2d7329d
//#define INTERNAL_SYSTEM_IP      "192.168.1.4"
//#define EXTERNAL_SYSTEM_IP      "192.168.1.8"
//#define OUR_KEY_SERIAL      2
//#define OTHER_KEY_SERIAL    1

// GK1
#define INCOMINGFILTER "ether src not 00:00:00:21:a0:1a and ether src
not 00:04:E2:D7:32:9C"
#define OUTGOINGFILTER "ether src 00:00:00:21:a0:1a"
#define MAC 0x00000021a01a
#define INTERNAL_SYSTEM_IP    "192.168.1.8"
#define INTERNAL_SYSTEM_IP_A  {192, 168, 1, 8}
#define EXTERNAL_SYSTEM_IP    "192.168.1.4"
#define EXTERNAL_SYSTEM_IP_A  {192, 168, 1, 4}
#define OUR_KEY_SERIAL        1
#define OTHER_KEY_SERIAL      2

#define EXTERNALDEVICE "eth0"
#define INTERNALDEVICE "eth1"
```

This needs to be modified to reflect the actual MAC addresses and IPs of the two systems that will be using the GateKeepers and not the GateKeepers themselves.) The MAC of the actual GateKeeper does however need to be included in the Berkley Packet Filter syntax found as the second MAC address in the INCOMINGFILTER definition.

In the above header file, the comment “GK1” refers to one of the clients, and “GK2” refers to the other client. You either comment out the whole “GK1” section or the whole “GK2” section. The reason I left both in there, is because development occurred on one of the GateKeepers so I left the code in there, and just uncommented the right information depending what computer I was using at the time.

On each GateKeeper, depending which network cable you plug into which network card, you will need to set the appropriate EXTERNALDEVICE and INTERNALDEVICE.

EXTERNALDEVICE is the network card that has a cable that leads to the switch/router.

INTERNALDEVICE is the network card that has a cable that leads to the computer that wishes to use the tunnel.

Other options like modifying the port number for the tunnel (9753 by default, make sure it’s open on both GateKeeper’s Firewall’s) are also in that header file, but it’s not necessary to alter anything else to get a quick test system up and running.

Feel free to ask me for a demonstration on the five computers I have set up.

Implementation Implications

There are some implications in implementing a secure tunneling system combined with the KeyVault system. Not only does the system create a secure point-to-point communications layer, but it also provides a way for dynamically adding new GateKeepers to the system without having to copy the key manually to every other client before communication can commence. At the same time it is satisfying the authentication requirement.

The problem with SSH (an alternative secure tunnel system) for example, is that it is vulnerable to man-in-the-middle attacks. Man-in-the-middle attacks, MITM, include any type of network attack where your information could pass through the hands of a hacker without you even realizing it.

Distributed keys, by their very nature destroy the possibility of a MITM attack; since, an unencrypted key exchange never occurs, there is never a chance for a hacker to intercept or spoof the keys.

Secure communication channels may be mandated at an organization or perhaps authentication is the goal. Imagine a scenario where a company only accepts email from other companies that have their email servers connected to a central KeyVault authority—anyone that sends an email through this system is guaranteed knowledge of the e-mail's server of origin. Trusted authentication of email is impossible at the moment because of very common email spoofing.

New Technologies Research

Whitenoise Stream Cipher

The Whitenoise stream cipher was a natural choice to the use in secure Internet tunneling, as this project required, for several reasons:

- Cryptographically Strong
  In terms of the cryptographic strength, please refer to the appendix referencing the Whitenoise stream cipher.

- Performance
  In terms of performance please reference the UVIC performance analysis in the appendix.

- Robust Bit-Independent Encryption
  The Whitenoise stream cipher provides a unique property that most other cryptography methods do not share—that is, once the data is encrypted, the bits are completely independent of one another. This is very useful when dealing with communications because often single bits will get corrupted when transferring large amount of information, and sometimes it's impossible to re-send the information, (perhaps when streaming video from a spy plane flying over Iraq for example) and so when the cryptography method used fails because of one bit being corrupted, then the data is lost or a huge performance hit is reached due to the necessity to resend the data. Whitenoise overcomes this issue by being completely bit independent. If a bit gets corrupted while being encrypted in Whitenoise, the resulting decrypted data is exactly how it would be as if it weren't encrypted in the first place.

  For a more detailed look at the construct of Whitenoise, please refer to the Power Point included in the addendum.

Libpcap

http://www.tcpdump.org/

The libpcap libraries were a necessity for this project as direct reading of packets off of the data link layer was necessary. This was necessary since this would allow for absolute transparency, and capture all traffic regardless of destination.

Libnet

http://www.packetfactory.net/libnet/

The libnet libraries were required in order to write back custom shaped packets that maintain the path established by the original packet while also encapsulating the encrypted data inside of a custom UDP packet with the WN header appended to the end. This appendage of custom headers isn't facilitated with higher-level packet transmission classes. This library suite had a higher learning curve than the libpcap libraries, as this library was used in its "Advanced mode" in order to write raw custom shaped memory blocks back to the network card.

QT

http://www.trolltech.com/

The threading libraries provided by Trolltech in the form of QT were very similar to the thread classes in Windows. There was still a learning curve in understanding how to point to the proper libraries in Linux and dealing with the lackluster documentation of the open source community.

Future Enhancements

While implementing this project, many future improvements were identified though not implement due to time constrains. Some of these improvements include:

- Packet Authentication Pad to be added to the custom Whitenoise header
    - This will be used to protect against the remote possibility that small predictable rejection responses of a server may be blocked and intercepted by a hacker in order to reverse engineer small portions of the WN Stream. This authentication pad would consist of another segment of the WN Stream interacting with WN Labs' CRC checker (which eliminates the possibility of a 100% predictable packet.)

- IP Fragmentation Completion
    - Currently the GateKeeper Tunnel Packet Fragmentation is causing approximately a 1% corruption of fragmented packets—this of course should be fixed in the system if 100% transparency is to be maintained. This fragmentation is necessary for maintaining packets under the maximum transmission size for Ethernet of 1500 bytes. In the configuration section it's mentioned that MTU should be set to 1300 bytes in order to make sure that fragmentation by the tunnel never occurs.

- The MAC address and IP addresses inside the tunnel should be replaced by the tunnel packet's MAC and IP in the unwrapped packet
    - This is necessary to ensure compatibility with subnets across the Internet. Currently the system would only work on a LAN or on an exposed Internet connection with no network address translation.
    - Some sort of MAC to IP address binding could be added as a sort of failsafe to double check the authenticity and watch for attack attempts.

- KeyVault Protocol
    - The KeyVault is currently only configured to handle maximum key sizes of 2^16. Implementing a KeyVault protocol to handle Key Fragmentation could solve this problem. Other features like GateKeeper registration and update management could be incorporated. This could also be used to add Ip addresses dynamically to the list of secure systems so that you wouldn't have to create rules manually.

- Interface
    - Right now, there is no user-friendly way of using the system. You really need to know your C++ and Linux to use the system. Shane Patel has been working on a web interface for the GateKeeper, though ultimately a little system tray icon would be nice.

- Logging
    - Adding some sort of logging facility that could watch for attack attempts or offset synchronization issues would be very useful for system administrators to identify malicious activity. Knowing the attack is there is the first step in stopping it.

- Offset Overlap Checking
    - Right now there is no checking to see if an offset is being used twice—this is of course dangerous in an OTP scheme since a pad should never be used more than once otherwise it makes it suspect to statistical analysis attacks.

Some systems in the near future that could benefit from the DDKI architecture, besides the tunnel, may include email servers/clients, and cell phones to establish secure calls in the field.

Since the system relies on Berkeley packet filter type expressions to determine the types of packets read, this system could be easily integrated with firewall features. This would be a big bonus to the users of the system.

Traditional Attacks

There are many traditional data link based attacks used to steal data in transit on a local area network. Recently at the West Coast Security Forum, a Mr. Allan Alton gave a presentation outlining some of the most common ones. (See appendix)

These attacks are usually oriented around intercepting data and viewing communications that wasn't meant for them. Some attacks are performance hit oriented.

Using the tunnel system would not protect the network from the performance hit, but would indeed prevent the data from being compromised.

The two categories of techniques for stealing data usually involve either pure pass-through interception of the traffic, or alternatively some sort of authentication simulation. In an SSH handshake for example, the man-in-the-middle (attacker) would shake hands with both parties making them believe the MITM is actually the other party. In an ARP poisoning attack on the other hand, there is no handshaking necessary—the attacker simply redirects the traffic to his machine and is able to look at it in a packet sniffer.

Some of the attacks Alton talks about are:

- Attack Within a Subnet
    - When a network has only an external facing firewall, if any client can be compromised inside the subnet on any port, then instantly the hacker can launch attacks from inside the network to other clients.
    - The tunnel system in it's current form would not protect against this type of attack; however, under the future enhancements section there is the recommendation of adding firewall functionality to the GateKeeper in which case this would remedy the problem.

- Broadcast Storm
    - Layer 2 broadcast packets are flooded onto the LAN causing all the clients to have to deal with it. This is a performance-hit attack.
    - The GateKeeper could deal with this if a firewall interface was added. Since we are at the data link layer, we could potentially filter out any unwanted traffic—as Alton suggests, a rate limit on certain traffic types could be implemented.

- MAC Flooding
    - One of the attacks we learned about in the network security specialization courses; a switch is flooded with many different MAC addresses, and since switches only have a limited MAC

buffer, sometimes the switch will break down and start to act as a hub sending all network traffic to every client which of course is a privacy issue as well as performance hit.
    - The GateKeeper would make such an attack useless from the data-capturing point of view, since any traffic captured would be encrypted. The performance hit would still be there alas.

- DHCP Rogue
    - This attack involves placing another DHCP server on the network that would assign IP addresses to clients. This is a problem since the DHCP response also assigns the DNS server and network gateway server IPs to the client. All web site domain names get resolved through the DNS so an attacker could make www.realbank.com redirect to www.fakebank.com without the user even noticing. All inter-network traffic outside the current subnet goes through the network gateway, so an attack could tell all the clients that its client is the gateway causing all Internet bound traffic to first go through their machine so they could look at it. When the tunnel system is in place on a LAN, any captured data, whether it be destined for a remote server going through the gateway, or DNS UDP data traffic would be encrypted (if so configured) and would be useless to an attacker. This is assuming the DNS or the remote systems use the same system on their end. If there were communication with non-GateKeeper systems then the attacker would indeed be able to view the traffic.

- Spanning Tree Hijack
    - This attack requires that a client be connected physically to two different switches that are also connected to each other. Switches provide a protocol known as the spanning tree protocol, STP that basically looks for loops in the network configuration and cuts them off. Great when nothing malicious is going on, but when a client causes a loop and sends out STP packets on both connections, the connection between the two switches is broken. The attacker would then place his machine in IP forwarding mode acting like a bridge. All the inter-switch destined traffic then flows through his machine. Again, just like with the DHCP rogue, any non-encrypted traffic could be viewed, but any tunnels set up between computers that communicate through the switches could not be viewed.

- ARP Table Poisoning
    - Computers on a LAN all have an ARP, address resolution protocol, table that binds IP addresses to MAC addresses. The MAC addresses are the physical addresses and the IP addresses are the "direction" they should take. When a client wants to talk to IP 1.2.3.4, he looks in his ARP table and sees that the IP is at MAC ABC and sends off the packet to ABC. ARP poisoning involves an attacker sending unsolicited ARP responses which tell the victim 1.2.3.4 is actually at MAC DEF, so all traffic destined to ABC actually goes to DEF first. Of course DEF would be the attacker's machine set up to forward the traffic to the real ABC yet now the attacker has full view of any traffic going between the two machines. This of course would be a futile attack if a tunnel were configured between the victim's IP and the 1.2.3.4/ABC machine since all traffic would be unreadable by the attacker.

- Simple IP Address Spoofing
    - Sometimes a machine will plug into a network and simply spoof the IP address of another host in the hopes that they will see the traffic destined for that host. If a tunnel is configured for that IP address, then spoofing will do you no good since your applications wont have a clue what to do with wrapped encrypted packets.

Disabling non-encrypted traffic is of course an option in the GateKeeper system; however this is not practical for most environments since people need to send email outside of the company and surf the web.

In some situations like in hospitals and military, perhaps even corporate research facilities, the need for security may be great enough that the GateKeeper would drop all non-encrypted traffic.

Conclusion

The distributed key architecture put forward in this project, combined with any system requiring point to point secure communications like the use of secure internet tunnels is definitely a valuable asset to the information technology community.

As the project currently stands, and as outlined in the future enhancements section, it is not commercially ready, though the project does fulfill its duty as a proof of concept of the designs developed.

The Whitenoise cryptographic design is very flexible, and is perfect for this type of system.

Future Whitenoise Laboratories projects, as well as future BCIT practicum projects will undoubtedly continue on from where this project has left off.

Appendix

Source Code

See included source code:

GateKeeper code: Source Code\GK
KeyVault code: Source Code\KV

UVIC Performance Analysis

See
http://www.isot.ece.uvic.ca/projects/whitenoise/report.pdf

Whitenoise Stream Cipher Reference

See included
Whitenoise Technical Reference.ppt

Alton's Layer 2 Attacks – West Coast Security Forum

See included
AllanAlton – Layer2Security.pdf

This PDF contains a numeration of many different layer 2 local area network attacks that would be ineffective against capturing information that is utilizing this project.