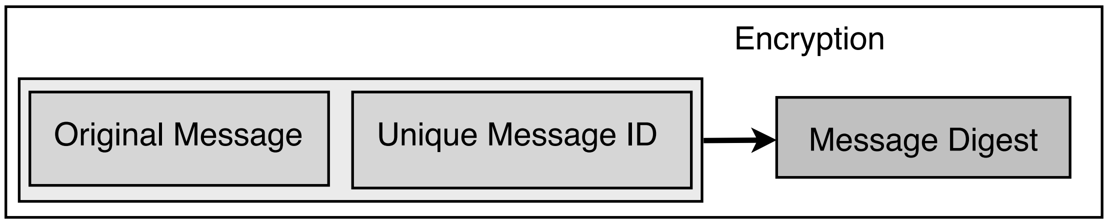
StealthNet - An Implementation

Implementation Summary



Authentication

Authentication within our *StealthNet* implementation uses key exchange to strengthen the system against a passive attacker. If we were to exchange keys or decide on a key in the clear it would be very easy for an attacker to simply listen to each message then decrypt them with the information they have obtained. Instead we are using the *Diffie-Hellman* key agreement protocol to agree on a secret key without ever revealing this key in the clear.

Diffie-Hellman uses a private key on each side along with shared public keys to arrive at the same shared secret key. While this is not a perfect key exchange/decision system it has some valuable security. It is also a simple system that does not use very much communication yet still reaches a secure point.

There are many other alternative methods, some are beyond the scope of this assignment, others are not. One example of another alternative method we could have implemented is *Merkle's Puzzles*. We chose *Diffie-Hellman* over *Merkle's Puzzles* as it is simpler to implement and just as secure.

Confidentiality

Once we have agreed on a shared secret key we can use this in a keyed cypher. There are many popular cypher methods and each of these is implemented by a number of different cyphers. We decided on using the *JCE Sun* implementation of *Blowfish* a fast, keyed, symmetric block cypher.

Our *Blowfish* cypher is initialised with IV and key based on the secret key we have agreed upon during the *Diffie-Hellman* steps of Authentication. This allows each side to use the same key (and cypher) for encryption and decryption. This simplifies a lot of operations.

Blowfish itself as an algorithm is fast and easy to use. As we are using the *JCE* we are completely abstracted from the implementation details of these algorithms. Because of this we chose the strongest available algorithm we could. Originally we had looked at using either *DES* or our own stream cipher. We finally settled on *Blowfish* after reading about how strong it still was.

Integrity

We provide Integrity in our implementation via the use of a *Hash Based Message Authentication Code*. The basic premise is that we hash the original message with a *Hashing Algorithm*, in our case *MD5*, take the hash and append it to our message. We then let this combined message go through encryption. At the other end we can pull it

by Benjamin Taylor - 307204189 - btay9478@uni.sydney.edu.au
and Joshua Akehurst - 306123304 - jake0672@uni.sydney.edu.au

apart and piece it back together, we'll have the original message and we'll be able to tell if it has been modified as well.

There are many algorithms that can be used in *HMAC* integrity checking. We decided to use the *md5* algorithm, mostly because of its popularity. While it is not the best possible Hashing algorithm it is a very good one that does not have any very large flaws. It was also chosen as it came with the existing *JCE* framework and allowed us to (as before) stay away from implementation details.

Preventing Replay

Preventing replay is an interesting problem as all previous methods ensure security, there is no way the current system (explained so far) can protect against duplicate valid messages.

The obvious method to solve this problem is to timestamp each message. The timestamp would stay inside the *MAC* and encryption and be quite secure, making each message unique. However we were not happy enough with this system as it seems to be more *security-by-obscurity* than real security.

An alternative method would be for each side to give each message a unique identifier. If these identifiers were synchronised then we would have a message order and would know when a message had been seen before. This is close to a great solution.

Our final solution was to use the secret key decided on earlier as a seed to a random number generator. To simplify synchronisation issues we combined the secret key with the username to create two different random number generators. This way we had one for ourselves and one for the other party. We then stored a value which was the expected identification code for the next message. Upon recieving a message we would check if its code was the same as the expected one, if not it is an invalid message, if so then progress the random number generator and store the next value. For sending we simply progress our own random number generator and send the output value along with the message.

Possible Attacks

Chosen Plaintext Attack

Using a *Chosen Plaintext Attack* it would be possible to expose the previous protocol being used to send this data. There is no real encryption, everything is sent in the clear, however it looks encrypted due to the hex form string. If one were to send a single message with Chosen Plaintext it would be simple to see that there is a direct correlation between the message and its encoding. This would allow complete reverse engineering of the protocol and would help further cracking attempts. This is what is often considered security by obscurity.

Man In The Middle Attack

As with most security protocols we are most vulnerable to a *Man In The Middle Attack*. If one were to sit between each transmission, sending your own public keys and decomposing, storing, then recomposing each message using the same protocol we use it would be very simple to be completely invisible. In fact it would be possible to bypass the integrity of the *HMAC* scheme completely and send your own falsified data without anyone knowing.