Dust v2

Polymorphic Protocol Engine for Circumventing Filters

Introduction

Dust v2 is a tool for encoding Internet traffic so that it can circumvent filtering hardware. It uses statistical models to generate a family of encodings which conform to the constraints necessary to be passed through specific filters rather than blocked. It builds on the work of Dust v1, which focused on encoding traffic so that all characteristics followed a uniform random distribution. Dust v2 improves on this by shaping the characteristics to fit an arbitrary target distribution. This allows for protocol mimicry in addition to protocol obfuscation and is a more practical means of circumventing real world filters.

Use Case

The use case for Dust assumes that there is a network connected to the Internet through a filtering devices which selectively blocks some traffic while allowing other traffic through. It is also assumed that there is an unfiltered out-of-band channel of communication available to users of this network. This is an assumption which is shared by circumvention tools in general as in order to use software tools to circumvent filtering the users must first obtain these tools via some out-of-band channel. Dust uses this channel not just to distribute the software, but also to bootstrap a secure obfuscated connection.

Packet Filtering Techniques

The packet filtering techniques currently in use attempt to keep as little state as possible in order to be scalable. Filtering can happen either at the individual packet level (drop banned packets) or at the stream level (block or throttle streams containing banned packets). For stream level filtering it is common to sample only the initial packets of the stream, or to do random statistical sampling of packets. Filters do not keep persistent state about streams other than whether they have been marked as banned. Technique for defeating filtering can therefore concentrate on not sending packets which will be marked as banned.

There are two general classificiations of techniques for determining of a whether a packet is banned. Shallow packet inspection (SPI) uses just the headers of the packet. This is less expensive because the headers need to be examined anyway in order to route the packet. Deep packet inspection (DPI) simply refers to examining the packet contents as well as the headers. DPI used to be considered too expensive to be practical, but is now in widespread use by some filters. SPI techniques in active use for marking packets are as follows: source IP and port, destination IP and port, and packet length. DPI techniques in active use for marking packets are as follows: examing packet for connection headers and handshakes of known protocols, examining packet contents for banned static strings, and examining packet contents for banned string patterns.

How Dust Circumvents Filters

Dust is an engine for generating Internet protocols used to send packets which defeat the various filtering techniques currently in use. There is a client and a server. In order for them to communicate they must both be using the same encoding. Encodings can be devised which make traffic look random or which mimic existing protocols.

IP and port blacklists - Like most circumvention tools in use today, a proxy server is necessary which has an IP address has not been added to the filters blacklist. Additionally, a port must be chosen for communication which has not been blacklisted by the filter.

Packet length - Dust packets have randomized lengths, shaped to a target distribution. Different encodings can make the packet lengths look random or like an existing protocol.

Connection headers and handshakes - Unlike SSH and SSL (both now blocked but some filters), Dust contains no plaintext handshake. All Dust traffic is encrypted, starting with the very first byte.

Static strings and string patterns - Dust packets are encrypted, therefore the contents are randomized and static strings and string patterns are removed. However, some filters actually require certain strings to be present in order to circumvent filtering. Therefore, in the shaping stage Dust can actually insert strings.

Statistical properties of content - Even encrypted content can filtered, either by looking for high entropy content, or requiring that the content conform to the statistical properties of a certain protocol. After encryption, Dust content is shaped to have a statistical distribution based on a target distribution. Different encodings can make the content look random or like an existing protocol.

High-Level Overview

Introduction

Dust is a polymorphic protocol engine capable of encoding and shaping network traffic to conform to specified properties. It is composed of several layers which first erase all identifying information from the original traffic and then shape the results to conform to the specified protocol model. This breaks the information flow between the hidden and observable traffic, allowing for control of how the traffic is classified by the filter.

Application Layer

The first step in using Dust is to define an application-specific format for communication. The further layers of the protocol are agnostic as to the contents of each message and treat them as opaque byte strings. The application layers therefore have complete freedom in choosing a format. Some useful functions are provided in the Dust library for handling tasks such as cryptographic signing of messages. Any security features, such as secrecy or authentication, must be added to this layer as the other layers are intended to only provide obfuscation.

Encryption Layer

The encryption layers takes as input a plaintext message generated by the application layer and generates an encrypted message. Unlike other encrypted protocols such as TLS, there is no unencrypted header. The output of this layer is uniformly random. Only two classes of content are included in the output: encrypted data and single use random bytes (nonces). The output of the encryption layer is a uniformly random stream which internally consists of a header followed by one or more encrypted messages. This layer in agnostic as to the contents of the plaintext message, so the application layer is free to choose any format for messages.

The purpose of the encryption layer is to ensure a uniformly random distribution of bytes as input to the shaping layer. This layer disconnects information flow from the hidden traffic to the observed traffic. It ensures that the output of the shaping layer is dependent only on the statistical model used and not on the original message. The encryption is only for providing better obfuscation and is not intended to provide long-term secrecy. The encryption only needs to remain secure until the message has passed through the filter and has been received at the other side. Care has nonetheless been taken to provide a secure encryption layer.

The Dust protocol engine can generate encodings which communicate over TCP or UDP. The message to be delivered is first encrypted. Then a header is added specifying the necessary information to read the message such as the public key of the sender and an initialization vector, both of which are random. Together the ciphertext and random header form a random bytestring. This is shaped using reverse Huffman encoding to the target distribution for content. It is then divided into packets using the target distribution for packet lengths.

The Key Exchange

Protocols such as SSL and SSH initiate a public key exchange using a plaintext handshake. They are therefore susceptible to protocol fingerprinting and filtering. Dust also requires a public key exchange, but does not utilize a plaintext handshake. The Dust handshake uses the Seattle Protocol, developed at the Obfuscation Lab workshop in Seattle in 2013. The Seattle Protocol is a specific implementation of the ntor cryptographic key exchange protocol. The original ntor paper did not specify implementation details such as ciphers to be used or the wire protocol for exchanging messages. These are specified in the Seattle Protocol specification.

In order to accept a connection from an unknown host, a Dust server must first complete a key exchange with the client. The Dust server first creates a permanent public key. The server then creates an invite, which contains the server's IP, port, public key, and the parameters for the encoding to be used. The server operator then distributes the invite to the client out-of-band.

The client generates an ephemeral public key for use only on the current connection and then uses the IP and port information from the invite to connect to the server. The client and server then exchange ephemeral public keys and the client provides proof that it knows the servers permanent public key. This provides perfect forward secrecy as well as defends against active attacks which probe random IP addresses looking for servers. The encryption key is derived from the ephemeral public keys. This completes the second layer of the public key exchange.

At this point, the client generates a random initialization vector and sends it to the server. This IV along with the encryption key allows for a conversation consisting of one or more encrypted messages to begin.

Sending Messages

Once the conversation has begun, messages can be sent. To send a message, the encryption key and the initialization vector are used to encrypt first the message length and then the message. This creates an entirely encrypted, random-looking sequence of bytes. This is then encoded according to the current protocol parameters to create an encoded message that fits the given parameters. For instance, it may be encoded to look like an HTTP message, mimic any other protocol, or be shaped to look unlike any existing protocol, all dependent on the parameters. This shaped message is then sent to the server.

When the server receives an encrypted message, it first decodes it according to the given protocol parameters to recover the encrypted message. It then decrypts the message length and uses this to extract a message of the correct length. As bytes may be added to the end of the message to adjust its length to the target protocol parameters, the length is necessary to know which data to keep and which to discard. Once the message is decrypted, it is delivered to whatever backend service the server is providing.

Randomness

A key consideration in the Dust protocol is that the only information in a Dust packet should always be effectively random to an observer. There are only two types of information in a Dust packet: encrypted (should appear random) and single-use random bytes (actually is random). The apparent randomness is essential to the protocol's ability to avoid detection. Therefore, care should be taken to use a good random number generator and to never reuse random bytes. For instance, the client's ephemeral public key is intended for a single use and should never be reused. The server's permanent public key is intended for multiple uses and so should never be sent over the wire. Similarly, the initialization vector used in data packets should always be randomly generated and never reused between data packets.

Wire Protocol

The cryptographic protocol and wire protocol used to establish secure communication using the uniformly random byte stream is fully documented in the Seattle protocol specification. For the purposes of this document, all that matters is that the application layer data goes into the encryption layer and results in a uniformly random encrypted byte stream for use in the shaping later.

Shaping Layer

The shaping layer takes the uniformly random output from the encryption layer and shapes it to match the target statistical model. Each statistical property of the traffic (for example, the distribution of characters in the contents or the packet length) is represented in the statistical model as a probability distribution. The encrypted byte stream is used as a psuedo-random number generator (PRNG) to sample from the probability distribution. Some properties (for example, packet timing) are not transferred losslessly and so are not used to encode information. They are sampled using a separate secure PRNG rather than the encrypted byte stream. These sampled values are then used to form the generated traffic. For example, a model that includes the properties of the packet contents, length, and timing will generate a packet with a random contents, length, and timing drawn from those probability distributions. Since the sampled values are drawn from the probability distributions representing the significant properties, they conform statistically to those distributions. From the filter’s point of view, the observed traffic will fit the profile of the target protocol and will therefore be classified as the target protocol. Since the target protocol is chosen to be one that is allowed by the filter, the encoded traffic will pass through the filter. When the traffic is received after it has passed through the filter, it is decoded to recover the original hidden traffic. Properties that were sampled using an independent PRNG (such as packet timing) are ignored. Properties that were sampled using the encrypted byte stream as a PRNG are processed by a decoder function that is the inverse of the encoder function. This reverse transformation recovers the encrypted byte stream. The encrypted byte stream is then decrypted to reveal the original plaintext message.

The shaping later determines what traffic is actually sent over the network. It takes as input the encrypted stream from the encryption layer. However, traffic sent over the network is determined independently of whether or not there is actual data to send. This is achieved by decoupling in the API. There is one API call to register new data to be sent, and another API call to obtain the actual bytes that will be sent over the network. The latter always succeeds, regardless of whether the former has been called. Therefore, when there are bytes to send they will be used and when there are no bytes to send random padding will be used.

Shaping Features

The first step in sending traffic is to obtain a connection duration. Once the connection duration has been determined the client opens a connection to the server. The duration is measured in milliseconds and when the duration has elapsed the connection to the server is closed. The connection is closed regardless of whether there is still data to send. If there is additional data to send, a new connection must be opened.

While the connection is open, the client and the server both send packets to each other. In order to determine how many packets to send, a number of patches to send is obtained from the Dust engine. Unlike other protocol obfuscation tools, Dust does not model packet inter-arrival times, but instead models the flow rate in terms of the number of packets per second. Each time the Dust engine is queried for the number of packets to send, the number of milliseconds which is elapsed since the last set of packets were sent must be supplied.

Once a number of packets to send has been determined, the length of each packet must also be determined. The Dust engine is queried for a length for each packet. For each packet to be sent, a number of bytes equal to the packet's length is obtained from the engine. These bytes may or may not contain encrypted data. If they do not contain encrypted data that they will contain randomly generated padding. While these bytes are randomly generated, they will not be uniformly random. They will follow the target distribution for content.

To decode a received message, packet timing and length is ignored as these characteristics may be altered during transmission. The bytes of the packet content are decoded and decrypted. Bytes which contain padding are discarded.

An important thing to note about the client/server communication process with Dust is that the traffic sent over the network will always follow the specified target distribution for each property regardless of any other concerns. Servers will continue to communicate with clients even if decoding and decryption of the received messages fails. The reason behind this is that the traffic sent over the network should never reveal any information about the underlying messages being sent.

Protocol Specifications

Application Layer Protocol Specification: Ensemble Protocol

Overview

The ensemble protocol allows for connection oriented byte streams to be transmitted over an ensemble of available connections. It is suitable for use over multiple TCP connections. By allowing for transport over multiple connections, data transmissions or interactive conversations can be communicated which are longer than would otherwise be available. The use case this protocol is primarily designed for is one in which obfuscating transports are used to transport data across filtered networks. The limitations of obfuscating transports are such that multiple discrete TCP connections may be necessary to transport all of the data received from a single source TCP connection. Other potential use cases which the protocol may be used ones in which TCP connections are unreliable and may drop or stall before all the data has been transmitted.

Protocol

The client begins by either initiating a new session or continuing an existing session. When creating a new session, the server returns the session ID for the new session. A number of data messages are then sent which include the session ID and the sequence number starting at 0. At any time, either side can send a close message to end the session. For particularly long-lived sessions, the sequence numbers may be insufficient for the number of messages to be sent. This case, an extend message can be sent to reset the sequence number to zero and get a new session ID which extends the current session.

Starting a new session

- Client request
    - 1 byte - command code, 0x00 New Session Request
- Server response
    - 1 byte - command code, 0x01 New Session Okay
    - 32 bytes - new session ID

Continuing an existing session

- Client request
    - 1 byte - command code, 0x02 Continue Session Request
    - 32 bytes - session ID to continue
- Server response
    - 1 byte - command code, 0x03 Continue Session Okay
    - 32 bytes - session ID to continue

Extending a session

- Client request

    - 1 byte - command code, 0x04 Extend Session Request
    - 32 bytes - session ID to extend
- Server response
    - 1 byte - command code, 0x05 Extend Session Okay
    - 32 bytes - new session ID

Closing a session

- Request (can be client or server)
    - 1 byte - command code, 0x06 Close Session Request
    - 32 bytes - session ID to close
- Response (other side, server or client)
    - 1 byte - command code, 0x07 Close Session Okay
    - 32 bytes - session ID to close

Sending data

- 1 byte - command code, 0x0F Data
- 32 bytes - session ID
- 2 bytes - sequence number
- 2 bytes - length
- Variable - data

Errors

- 1 byte - error code, 0xF0 Too Many Sessions
- 1 byte - error code, 0xF1 Unknown Session ID
- 1 byte - error code, 0xF2 Connection Is Closed
- 1 byte - error code, 0xF3 Bad Sequence Number

Encryption Layer Protocol: Seattle Protocol

Introduction

This document outlines the Seattle Protocol, which consists of a secure key exchange protocol with server identity verification and a wire protocol for transmitting the key exchange and messages encrypted with the exchanged key.

Handshake Protocol

The ntor Protocol, Step by Step

When $\overline{B}$ is initialized as a server:

- 1. Set b $\overset{\$}{\longleftarrow}$ {1, . . . , q − 1} and set B ← g^b.
- 2. Set (b, B) as $\overline{B}$’s static key pair.
- 3. Set $cert_B$ = ($\overline{B}$, B) as $\overline{B}$’s certificate.

When initializing the server, no messages are output.

The server generates a key pair [1,2] and the public key becomes the server's certificate [3].

When $\overline{A}$ is initialized as a client:

- 4. Obtain an authentic copy of $\overline{B}$'s certificate.

When initializing the client, the server's certificate is delivered out of band [4], along with the other information necessary for contacting the server such as the server's IP address and port.

When $\overline{A}$ receives the message (params, pid) = ((“new session”, ntor), $\overline{B}$):

- 5. Verify that $\overline{A}$ holds an authenticated certificate $cert_B = (\overline{B}$,B).
- 6. Obtain an unused ephemeral key pair (x, X $\leftarrow g^x$); set session id $\Psi_a = H_{sid}(X)$.
- 7. Set $M_{state}^{\overline{A}}[\Psi_a] \leftarrow (ntor, \overline{B}, x, X)$.
- 8. Return session identifier Ψ_a and outgoing message $msg' \leftarrow (ntor, \overline{B}, X)$.

The “new session” message is conceptual and not actually output. It represents the event in which the client seeks to initiate a new session with the server.

In order for the client to connect to the server, the client and the server both must already be initialized [1,2,3] and the server's certificate must have been transmitted out of band to the client [4,5].

The client generates an ephemeral key pair which is only used for this session [6]. A hash of the public key of the client's ephemeral keypair is used as the session identifier [6]. The client stores session information in memory consisting of the ephemeral keypair and the server's public key, using the session identifier as the key for retrieving the session information [7].

The output is a message generated by concatenating the static string “ntor”, the server's public key, and the client's ephemeral public key [8]. This message requests from the server that a new session be established.

When $\overline{B}$ receives the message msg = (ntor, $\overline{B}$, X):

- 9. Verify X $\in G^*$.
- 10. Obtain an unused ephemeral key pair (y, Y $\leftarrow g^y$); set session id $\Psi_b \leftarrow H_{sid}(X)$.
- 11. Compute $(sk', sk) = H(X^y, X^b, \overline{B}, X, Y, ntor)$.
- 12. Compute $t_B = H_{mac}(sk', \overline{B}, Y, X, ntor, "server")$.
- 13. Return session identifier Ψ_b and outgoing message $msg' \leftarrow (ntor, Y, t_B)$.
- 14. Complete Ψ_b by deleting y and outputting $(sk, *, (\overline{v}_o, \overline{v}_1))$, where $\overline{v}_o = (X)$ and $\overline{v}_1 = (Y, B)$.

When the server receives a message requesting a new session be established, it first checks that the client's ephemeral public key is a valid public key [9].

The server then generates an ephemeral key pair which is only used for this session [10]. A hash of the public key of the server's ephemeral keypair is used as the session identifier [10].

The server then generates a shared key and a confirmation code. The shared key is calculated by taking a hash of the following values: the key generated by an ECDH between the client's ephemeral public key and the server's ephemeral private key, the key generated by an ECDH between the client's ephemeral public key and the server's static private key, the client's ephemeral public key, the server's ephemeral public key, and the string "ntor" [11]. The shared key is then used to calculate the confirmation code. The confirmation code is calculated by taking an HMAC of the following values: the shared key, the server's identifier (for instance, the server's IP address), the server's public key, the client's public key, the string "ntor", and the string "server" [12].

The server responds to the client's request for a new session with a message consisting of the concatenation of the following values: the string "ntor", the server's ephemeral public key, and the confirmation code [13].

Once the confirmation code has been sent, the server forgets it's ephemeral private key (this is what makes it ephemeral) and records the session information in association with the session identifier. The session information consists of the following values: the shared key, the client's ephemeral public key, the server's ephemeral public key, and the server's static public key.

When $\overline{A}$ receives the message $msg' \leftarrow (ntor, Y, t_B)$ for session identifier Ψ_a:

- 15. Verify session state $M^{\overline{A}}_{state}[\Psi a]$ exists.
- 16. Retrieve $\overline{B}$, x, and X from $M^{\overline{A}}_{state}[\Psi a]$.
- 17. Verify Y $\in$ G^*.
- 18. Compute $(sk', sk) = H(Y^x, B^x, \overline{B}, X, Y, ntor)$.
- 19. Verify $t_B = H_{mac}(sk', \overline{B}, Y, X, ntor, "server")$.
- 20. Complete Ψ_a by deleting $M^{\overline{A}}_{state}[\Psi a]$ and outputting $(sk, \overline{B}, (\overline{v}_o, \overline{v}_1))$, where $\overline{v}_o = (X)$ and $\overline{v}_1 = (Y, B)$.

When the client receives a response from the server, it firsts checks to see if it did in fact request a session with that server [15]. If so, it retrieves the session information, which contains the server's identifier (for instance, the server's IP address), the client's ephemeral private key, and the client's ephemeral public key [16].

The client next verifies that the server's ephemeral public key is a valid public key [17].

The client then generates a shared key and a confirmation code. The shared key is calculated by taking a hash of the following values: the key generated by an ECDH between the server's ephemeral public key and the client's ephemeral private key, the key generated by an ECDH between the client's ephemeral private key and the server's static public key, the client's ephemeral public key, the server's ephemeral public key, and the string "ntor" [18]. The shared key should be identical to the one calculated on the server. The shared key is then used to calculate the confirmation code. The confirmation code is calculated by taking an HMAC of the following values: the shared key, the server's identifier (for instance, the server's IP address), the server's public key, the client's public key, the string "ntor", and the string "server" [19]. The confirmation code calculation is identical to the one used on the server and should have an identical result if the shared key is, at it should be, also identical between the client and server.

The client then deletes the session state, including it's ephemeral private key, and stores a new session state consisting of the shared key, the server identifier, the client's ephemeral public key, the server's ephemeral public key, and the server's static public key.

Once this last step has been completed, the client and server both have the same shared key and they have verified by virtue of the confirmation code that the other side also has the correct shared key. At this point, encrypted communication can occur using the shared key as the encryption key.

Problems with ntor

There are two messages which are exchanged between the client and server in the ntor protocol. The out of band exchange of the server's static public key is not counted here. The first message is sent from the client to the server requesting a new session and the second message is sent from the server to the client in response. The client's message contains the following: the string "ntor", the server's identifier, and the client's ephemeral public key. The server's message contains the following: the string "ntor, the server's ephemeral public key, and a confirmation code.

In order for a handshake to work with obfuscating protocols, all output messages must be free of information which can identify it as belonging to an obfuscating protocol. As ntor is intended for use in obfuscating protocols, identifying characteristics such as transmitting the string "ntor" at the beginning of the handshake are undesirable.

Specifically, the requirements set forth by the Dust project for wire protocols, including handshake protocols, are that all bytes contained in output messages must be either encrypted or random single-use values (nonces). Given this criteria, we can look at the ntor messages and determine what changes are necessary to the protocol in order for it to be acceptable.

The client's message contains the following: the string "ntor", the server's identifier, and the client's ephemeral public key. The string "ntor" is clearly not encrypted or random and must be removed from the protocol. The server's identifier could be something non-random like an IP, or it could be a randomly generated string. However, since the same server identifier is used for each new session initiated with the same server, it is not single-use. Therefore, it does not meet the requirements and must be removed. The client's ephemeral public key is random because it is an Elligator public key and they are guaranteed to be uniformly random. Also, because it is ephemeral it is single use. Therefore, it meets the requirements and can remain.

The server's message contains the following: the string "ntor, the server's ephemeral public key, and a confirmation code. The string "ntor" is clearly not encrypted or random and must be removed from the protocol. The client's ephemeral public key is random because it is an Elligator public key and they are guaranteed to be uniformly random. Also, because it is ephemeral it is single use. Therefore, it meets the requirements and can remain. The confirmation code is a special case. It is neither encrypted nor random. However, it is the result of a hash function. If this hash cannot be calculated be an observer and is single use, then it can be allowed. Examining how the hash function is calculated, two of the inputs are the results of ECDH calculations. These calculations require private keys which are never transmitted. Ephemeral keys are also used, which are single use. Therefore, the confirmation code is a single use value since the ephemeral keys used to generate it will not be used again. Since it cannot be calculated by an observer it can be considered a random single-use value and is acceptable.

The modified version of the ntor protocol which fits the requirements therefore consists of a client message containing just the client's ephemeral public key, followed by a server response containing the server's ephemeral public key and the confirmation code.

One can assume that the parts removed from the ntor protocol were added for a reason and that removing them could have some impact on the security of the modified ntor protocol. While the security arguments made in the ntor paper do not reference these parts of the protocol, the author's may have had some other attacks in mind which were not documented in the paper. A security audit of the modified ntor protocol is therefore recommended.

Dust/Seattle Handshake

Initializing a server

- The server generates a static ECDH keypair using Curve25519 and becomes the server's certificate.

Initializing a client

- The client obtains a copy of the server's certificate out of band, along with the server's IP and port

Creating a new session

- Verify that the client has the certificate for the server
- Create an ephemeral Curve25519 keypair, associating it with the given server
- Encrypt the ephemeral public key with Elligator
- Send the encrypted ephemeral public key to the server

When the server receives a client request for a new session

- Obtain the client's encrypted ephemeral public key from the message.
- Decrypt the client's encrypted ephemeral public key with Elligator to get the client's Curve25519 ephemeral public key
- Create an ephemeral Curve25519 keypair
- Create a shared key (see below)
- Create a server confirmation code using the shared key (see below)
- Encrypt the ephemeral public key with Elligator
- Send the encrypted ephemeral public key and server confirmation code to the client
- Delete the ephemeral private key
- Store the shared key, associated with the client

When the client receives a server response for a new session

- Obtain the server's encrypted ephemeral public key and the server confirmation code from the message.
- Decrypt the server's encrypted ephemeral public key with Elligator to get the server's ephemeral Curve25519 public key
- Create a shared key (see below)
- Create a client confirmation code using the shared key (see below)
- Verify that the client confirmation code and server confirmation code are identical
- Delete the ephemeral private key
- Store the shared key, associated with the server

Creating a shared key

- On the server, hash with Skein-256-256 the following:
    - 32 bytes - ECDH(server's ephemeral private key, client's ephemeral public key)
    - 32 bytes - ECDH(server's static private key, client's ephemeral public key)
    - 7 or 19 bytes - server identifier (see below)
    - 32 bytes - client's ephemeral public key
    - 32 bytes - server's ephemeral public key
    - 4 bytes - "ntor"
- On the client, hash with Skein-256-256 the following:
    - 32 bytes - ECDH(client's ephemeral private key, server's ephemeral public key)
    - 32 bytes - ECDH(client's ephemeral private key, server's static public key)
    - 7 or 10 bytes - server identifier (see below)
    - 32 bytes - client's ephemeral public key
    - 32 bytes - server's ephemeral public key
    - 4 bytes - "ntor"

Creating a confirmation code

- Identical on both client and server, HMAC with Skein-256-256 and the shared key the following:
    - 7 or 19 bytes - server identifier (see below)
    - 32 bytes - server's ephemeral public key

    - 32 bytes - client’s ephemeral public key
    - 4 bytes - “ntor”
    - 6 bytes - “server”

Creating a server identifier

- For an IPv4 address:
    - 1 byte - flags (see below)
    - 4 bytes - IP Address
    - 2 bytes - port
- For an IPv6 address:
    - 1 byte - flags (see below)
    - 16 bytes - IP Address
    - 2 bytes - port

Creating server identifier flags

- 0 - 0 for IPv4, 1 for IPv6
- 1 - 0 for TCP, 1 for UDP
- 2-7 - Reserved, set to 0

Wire Protocol

The following reduces the complexity of the handshake protocol description into just what is actually output:

- Client Request to Server
    - 32 bytes - client’s ephemeral public key
- Server Response to Client
    - 32 bytes - server’s ephemeral public key
    - 32 bytes - confirmation code

Message Protocol

The final result of the handshake protocol is that the client and server both have an identical shared key which can be used for encrypting messages. Once the handshake is complete, message transmission can commence. For this phase of communication, the Seattle protocol provides a message transmission protocol.

The message transmission protocol begins with a header which consists of a random initialization vector (IV). After the header, any number of records can be sent until such point as the session is ended. The record format consists of an encrypted length followed by an encrypted payload of the specified length. The payload contains a header, data, and a verification code. As the client and server use the same shared key, the message protocol is identical for both client and server.

Message Protocol Step-by-step

Sending a message with data

- If this is the first message, send the header (see below)
- Generate flags (see below)
- Generate a verification code for the data (see below)
- Create a payload by concatenating the following:
  - 1 byte - flags
  - Variable - data
  - 32 bytes - verification code
- Encrypt the payload with the shared key to get the encrypted payload
- Obtain the length of the payload and encode it as a 2-byte value in network byte order
- Encrypt the length with the shared key to get the encrypted length
- Create a record by concatenating the following:
  - 2 bytes - encrypted length
  - Variable - encrypted payload
- Deliver the message

Sending a message without data

- If this is the first message, send the header (see below)
- Generate flags (see below)
- Generate randomly sized blank data
  - A random number of 0-valued bytes
- Generate a blank verification code for the data
  - 32 0-valued bytes
- Create a payload by concatenating the following:
  - 1 byte - flags
  - Variable - blank data
  - 32 bytes - blank verification code
- Encrypt the payload with the shared key to get the encrypted payload
- Obtain the length of the payload and encode it as a 2-byte value in network byte order
- Encrypt the length with the shared key to get the encrypted length
- Create a record by concatenating the following:
  - 2 bytes - encrypted length
  - Variable - encrypted payload
- Deliver the message

Sending the header

- Generate a random 32-byte initialization vector (IV)
- Deliver the IV

Generate flags

- 0 - 0 for no data, 1 for data included
- 1-7 - Reserved, set to 0

Generate a verification code for the data

- HMAC with the shared key the following:
    - 2 bytes - unencrypted length
    - 1 byte - flags
    - Variable - data

Wire Protocol

The following reduces the complexity of the message protocol description into just what messages are actually output. The client and server side use identical wire protocols.

- 32 bytes - IV
- Repeating
    - 2 bytes - encrypted length
    - Variable - encrypted payload
        - If data
            - 1 byte - flags
            - Variable - data
            - 32 bytes - verification code
        - otherwise
            - 1 byte - flags
            - Variable - blank data
            - 32 bytes - blank verification code

Dust API

Encoding

unsigned short int packet_count(unsigned short int milliseconds)

Takes the number of milliseconds elapsed since the last call and returns the number of packets to send

- Command byte: 0x00
- Request args: 2 bytes
- Response: 2 bytes

unsigned short int encoded_ready()

Returns the size of the next packet to send. This value is determined randomly, regardless of how much actual data has been queued up.

- Command byte: 0x01
- Request args: none
- Response: 2 bytes

void put_decoded(unsigned short int len, byte[] bytes)

Takes number of bytes and the actual bytes and adds them to the encoding queue.

- Command byte: 0x02
- Request args: 2 bytes, variable
- Response: none

byte[] get_encoded(unsigned short int len)

Takes number of bytes and returns that number of bytes, which may contain either encoded data or random padding. This call will always succeed and return the specified number of bytes, regardless of how much data has been queued.

- Command byte: 0x03
- Request args: 2 bytes
- Response: variable

Decoding

unsigned short int decoded_ready()

Returns the number of bytes available in the decoded queue. Since some data which is received maybe just random padding and contain no actual data, this function may return a lower value than expected relation to how much data has been queued for decoding.

- Command byte: 0x11
- Request args: none
- Response: 2 bytes

void put_encoded(unsigned short int len, byte[] bytes)

Takes number of bytes and the actual bytes. If the bytes contain encoded bytes, they will be decoded and added to the decoded queue. If the bytes contain random padding, they will be discarded. A given sequence of bytes may contain both actual data and random padding.

- Command byte: 0x12
- Request args: 2 bytes, variable
- Response: none

byte[] get_decoded(unsigned short int len)

Takes number of bytes and returns that number of bytes from the front of the decoded queue, removing the bytes from the queue. The number of bytes specified here should be whatever was returned from calling decoded_ready(). The smaller value can be used, but is less efficient. It is an error to use a larger value.

- Command byte: 0x13
- Request args: 2 bytes
- Response: variable