Link Server Protocol

DynamicSystems™

Contents

1 Connections **2**

2 Protocol details **3**

2.1 Messages . . . 3

2.2 Command numbers . . . 4

2.2.1 From the server . . . 4

Greeting - `0x01` - not secure . . . 4

Protocol Error - `0x02` - not secure . . . 4

Generic Response - `0x03` - not secure . . . 4

Secure Response - `0x04` - secure . . . 4

Timeout - `0x05` - not secure . . . 4

2.2.2 From the client . . . 4

Get Time - `0x81` - not secure . . . 4

Start Security - `0x82` - not secure . . . 4

Ping - `0x83` - secure . . . 4

Request Flag - `0x84` - secure . . . 4

2.3 Security . . . 5

2.3.1 Introduction . . . 5

2.3.2 Operation . . . 5

Encryption and decryption . . . 5

2.3.3 Working with data . . . 6

2.3.4 Checksum . . . 6

Chapter 1

Connections

Connections are made over TCP, using port **24992**. The production DynamicSystems™ link server is available at the host **problems.hackdalton.com** using this port.

Please note that there is a timeout of **10 seconds**. If you do not send any data for longer than the timeout, you will automatically be disconnected.

The purpose of the DynamicSystems™ link server is to provide our employees secure access to confidential information. We refer to this information as a *flag*. The flag is for DynamicSystems™ employees only.

The rest of this document contains technical details. As a DynamicSystems™ employee, you should use your link client and may ignore the rest of this document. Please remember that this document and your link client, like all DynamicSystems™ information, is confidential and proprietary.

Chapter 2

Protocol details

2.1 Messages

Once you open a connection, you then exchange *messages* with the server. Messages have the following structure. All sizes are measured in bytes.

Name	Position	Size	Description
Header	0	4	This should always be set to the bytes 0x44, 0x4E, 0x53, 0x4D in that order.
Command number	4	1	The command number describes the type of message. See section 2.2 for more information.
Sequence number	5	1	The sequence number starts at 0 and counts up for every message. Once it reaches 255 (the maximum number for a byte), it rolls over to 0. The server and the client have separate sequence numbers.
Data length	6	1	The amount of data attached to the message.
Security flag	7	1	If this byte is 0, the message is not secure. If it's 1, the message is secure and you must follow the steps in section 2.3. Other values are invalid.
Security message key	8	4	Used as part of the security system. Stored in *big endian*, also called *network byte order*. See section 2.3.
Security checksum	12	4	Used as part of the security system. Stored in *big endian*, also called *network byte order*. See section 2.3.
Data	16	?	Additional data for the message. This depends on the command you're using. This will have the length set in the data length field. The current link server has a maximum of 80 bytes of data in one message.

2.2 Command numbers

The *command number* describes the type of message. Certain command numbers are marked as secure, and so must be protected with the security mechanism in section 2.3. If a number is not marked as secure, you can still send it securely, but it's just not a requirement.

2.2.1 From the server

Greeting - `0x01` - not secure

This message is automatically sent when you first connect. The data will be a friendly greeting that should be displayed to the user. This may also be broadcast to all users if a company-wide announcement is made.

Protocol Error - `0x02` - not secure

This message is sent to indicate that there was an protocol error, such as sending an invalid command number. The data will be a human-readable string that describes the error.
This will **not** be sent for other types of errors. For example, if you send a Request Flag (`0x84`) message without the appropriate authorization, you will get a Secure Response (`0x04`) with an error message.

Generic Response - `0x03` - not secure

This is used as a response to the Get Time (`0x81`) message. The data will be a human-readable string that responds to whatever command was issued. You should display this string to the user.

Secure Response - `0x04` - secure

This is equivalent to the Generic Response (`0x03`) message, but sent using the security mechanism described in Section 2.3. It's used as a response to the Ping (`0x83`) and Request Flag (`0x84`) messages.

Timeout - `0x05` - not secure

This message will be sent if you are automatically disconnected due to inactivity. The data will be a string that you can display to the user.

2.2.2 From the client

Get Time - `0x81` - not secure

This message will ask the DynamicSystems™ server for the current time. You can use it to make sure the connection is working before beginning the security mechanism, since all other messages require a secure connection. The time is returned in the UTC timezone.

Start Security - `0x82` - not secure

This message is sent by the client to begin the security mechanism. See section 2.3.

Ping - `0x83` - secure

A ping message, used to test the security system. You may provide data along with your message, and it will be echoed back to you by the server as a Secure Response (`0x04`). Make sure that your data is as long as the data length you specify in the message! Also, make sure to abide by the maximum data length, as specified in section 2.1.

Request Flag - `0x84` - secure

This is the heart of the DynamicSystems™ link server. It may only be called by authorized users. If you are authorized, this message will result in a Secure Response (`0x04`) containing the flag.

2.3 Security

2.3.1 Introduction

The DynamicSystems™ link server sometimes deals with highly sensitive data. In order to protect DynamicSystems™ and its valuable intellectual property, certain messages must be sent in a secure manner. When a message is sent securely, only its data is encrypted. The other parts of the message (as described in section 2.1) remain unencrypted.

The core of the security system is an advanced mathematical operation referred to as "exclusive OR", or *XOR* for short. Cryptographers have proven XOR to be completely impossible to break under any circumstances. This operation is the fundamental building block used by the DynamicSystems™ Security System, and it's why we're so confident in calling it an *unhackable* system.

Description of the XOR operation is outside of the scope of this document; however, you may find many online resources available, such as `https://en.wikipedia.org/wiki/Exclusive_or`.

This document uses the $\oplus$ symbol to denote the XOR operation. For example, $A \oplus B$ is the notation for A XOR B.

2.3.2 Operation

To begin a secure connection, send a Start Security (`0x82`) command. The server will respond with two Generic Response (`0x03`) messages. The first will contain a useful security tip, which you should display for the user. The second will contain the *connection key*, described below. Once you've done that, you can start sending *secure messages*.

A secure message is the same as a regular message, but the security flag field should be set to 1. You'll also need to set a *security message key*. This can be anything you want, and should be different between messages. This key gets combined with the connection key to generate a *final key*. Specifically, the mathematical formula used is as follows:

$$\text{connection key} + \text{security message key} = \text{final key}$$

You should treat all of the keys as arrays of individual bytes. This means that values might overflow. An example of how this might look is as follows:

$$\left[\texttt{0x2E}, \texttt{0x8A}, \texttt{0xDA}, \texttt{0xFF}\right] + \left[\texttt{0x35}, \texttt{0xC1}, \texttt{0x47}, \texttt{0x73}\right] = \left[\texttt{0x63}, \texttt{0x4B}, \texttt{0x21}, \texttt{0x72}\right]$$

Encryption and decryption

The XOR operation is easy to reverse. If you've done $A \oplus B = C$, then $C \oplus B = A$. By using this principle, encryption and decryption of data is actually the same operation!

$$\text{output} = \text{input} \oplus \text{final key}$$

For encryption, the input would be the *plaintext* (the data you wish to encrypt) and the output would be the *ciphertext* (the encrypted data). For decryption, you would swap them, so the input is the ciphertext and the output is the plaintext.

For secure messages, you should send the ciphertext as the data for the message. As long as you have the correct settings in the message details (as described in section 2.1), the server will automatically decrypt your data.

The following sections (2.3.3 and 2.3.4) describe additional details about the correct message settings. You should read them carefully before attempting to implement this encryption.

2.3.3 Working with data

In the previous section, you might have noticed that not everything in the XOR operation has the same length. The final key is 4 bytes long, but your data might be longer than 4 bytes.

To work around this, you should repeat the key over and over. This is best explained with an example. Let's say you're encrypting some data, and that you've already determined the final key.

$$plaintext = \left[\texttt{0x53}, \texttt{0x65}, \texttt{0x63}, \texttt{0x72}, \texttt{0x65}, \texttt{0x74}, \texttt{0x20}, \texttt{0x79}, \texttt{0x61}, \texttt{0x79}\right] \text{ and } key = \left[\texttt{0x12}, \texttt{0x34}, \texttt{0x56}, \texttt{0x78}\right]$$

From before, we know that we must perform a XOR operation between the two. At a byte level, that would look like:

$$\begin{aligned}
plaintext[0] \oplus key[0] &= ciphertext[0] \\
plaintext[1] \oplus key[1] &= ciphertext[1] \\
plaintext[2] \oplus key[2] &= ciphertext[2] \\
plaintext[3] \oplus key[3] &= ciphertext[3] \\
plaintext[4] \oplus key[0] &= ciphertext[4] \\
plaintext[5] \oplus key[1] &= ciphertext[5] \\
plaintext[6] \oplus key[2] &= ciphertext[6] \\
plaintext[7] \oplus key[3] &= ciphertext[7] \\
plaintext[8] \oplus key[0] &= ciphertext[8] \\
plaintext[9] \oplus key[1] &= ciphertext[9]
\end{aligned}$$

Filling in the actual values, you get:

$$\begin{aligned}
\texttt{0x53} \oplus \texttt{0x12} &= \texttt{0x41} \\
\texttt{0x65} \oplus \texttt{0x34} &= \texttt{0x51} \\
\texttt{0x63} \oplus \texttt{0x56} &= \texttt{0x35} \\
\texttt{0x72} \oplus \texttt{0x78} &= \texttt{0x0a} \\
\texttt{0x65} \oplus \texttt{0x12} &= \texttt{0x77} \\
\texttt{0x74} \oplus \texttt{0x34} &= \texttt{0x40} \\
\texttt{0x20} \oplus \texttt{0x56} &= \texttt{0x76} \\
\texttt{0x79} \oplus \texttt{0x78} &= \texttt{0x01} \\
\texttt{0x61} \oplus \texttt{0x12} &= \texttt{0x73} \\
\texttt{0x79} \oplus \texttt{0x34} &= \texttt{0x4d}
\end{aligned}$$

Notice how the key repeats itself.

2.3.4 Checksum

So that the server can verify that it decrypted a message correctly, you must calculate a *checksum* of your plaintext. We use the *CRC-32* algorithm, which is commonly available as a built-in function of many programming languages. You'll want to use the *IEEE polynomial*, which is most likely the default. When encrypting a message, you should calculate the CRC-32 of its plaintext, and then set the security checksum field, as described in section 2.1.