DR Asset Encryption Specification

This specification applies to the encryption format used in the DR PSDB Related APIs, and can be used for any other similar scenarios.

# Background

Currently, all url references to video assets served to clients through the MU-Online API are delivered in clear text, in an API that has no authorization mechanism. This is deemed a practical and cost effective design from a technical standpoint.

This approach, while serving it’s purpose well, could be in violation of DRs responsibility to protect the rights granted to it by the various rights holders.

In order to address this, a solution that appropriately encrypts these urls with a shared key encryption is to be implemented. This restricts the readability of these urls to only code which implement the appropriate decryption key. This would be code developed by DR, or it’s partners. Third parties would only gain access to this secret key by deliberately reverse engineering DRs code, which would be a violation of DRs intellectual property rights to these implementations themselves.

# The Encrypted String Format

The string consists of one continual lowercase hexadecimal string, divided into several sections. For clarity, it’s a string representation of a continuous byte array, where a pair of hexadecimals is one byte (for those who did not know this already).

At the beginning of the string is a 1 byte header used to express the version of the encryption used. This is to ensure the ability to change this in the future, and allow compatibility with several versions. A 1 byte header makes room for 256 versions which should be more than sufficient.

All remaining bytes are considered a part of the version of the algorithm used, and can contain further headers specific to it. Refer to the individual algorithm versions for details.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Header | Data | | | |
| 01 | 01 e3 ce 32 c4 be be ce ce 3e a2 34 be 33 be fe de ad be ef 01 11 12 22 32 32 ... e3 | | | |

# Algorithm Version 1 (code name: “Duck and Cover”).

This algorithm AES Encrypts the data in question, and bundles it with the initial vector used. The cipher is derived based on a shared secret (not disclosed in this document), and the initial vector.

The first 4 bytes in this algorithm is a header that contains the length of the AES encrypted data (as number of hexadecimals). The remaining hexadecimals in the string will be the initial vector. It’s the assumption that the length of the data will never exceed that of a 32 bit signed integer. *We use a signed integer to make integration with languages like JavaScript simpler.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Header | Alg. Header | Data | | | |
| 01 | 00 00 fe 12 | 01 e3 ce 32 c4 be 34 be ce be fe de ad be ef 01 11 12 22 32 32 ... e3 | | | |

## Cipher Determination Function

Most encryption libraries work with byte arrays, which is why we use a hex representation. All hexadecimal strings in the data part of the payload is straight serialization from a byte array to a string of hexadecimals. *Whenever strings are converted to byte arrays we always assume UTF-8.*

|  |
| --- |
| cipher = SHA256(iv + “:” + secret) |

Where **iv** = *lowercase hex representation of the initial vector* (can be copied directly from the tail of the data), and **secret** is a string not disclosed in this document.

### Example

Assume the following byte sequence of hexdecimals:

|  |
| --- |
| 010000004001e3ce32c4be34becebefedeadbeef01122ad201e3ce32c4be34becadebadefadedeadbeef01111222 |

The first byte is the header. This shows 01 = (decimal 1), meaning it’s algorithm version one.

|  |
| --- |
| **01**0000004001e3ce32c4be34becebefedeadbeef01122ad201e3ce32c4be34becadebadefadedeadbeef01111222 |

According to the algorithm specification, the following 4 bytes (8 hex signs), will then be a header defining the payload length.

|  |
| --- |
| 01**00000040**01e3ce32c4be34becebefedeadbeef01122ad201e3ce32c4be34becadebadefadedeadbeef01111222 |

This value is 00000040 = (decimal 64), meaning that the following 64 characters are the encrypted payload, and the remainder is the initial vector.

|  |
| --- |
| 0100000040**01e3ce32c4be34becebefedeadbeef01122ad201e3ce32c4be34becadebadefade***deadbeef01111222* |

From this we can determine that:

|  |  |
| --- | --- |
| Encrypted Payload | 01e3ce32c4be34becebefedeadbeef01122ad201e3ce32c4be34becadebadefade |
| Initial Vector | deadbeef01111222 |

For this example, we’re going to pretend that the shared secret is “roflmao”. To determine the cipher:

|  |
| --- |
| cipher = SHA256(“deadbeef01111222:roflmao”) |

In C# this would look like this:

|  |
| --- |
| var data = “deadbeef01111222:roflmao”;  var dataBytes = Encoding.UTF8.GetBytes(data);  using (var algorithm = new SHA256Managed())  {  var cipher = algorithm.ComputeHash(dataBytes);  return cipher;  } |

Or in JavaScript, using the forge library:

|  |
| --- |
| var md = forge.md.sha256.create();  md.update(“deadbeef01111222:roflmao”);  var cipher = md.digest();  return cipher; |