SRP_Hash

Generates a cryptographic hash for a given string.

Syntax

```
Hash = SRP_Hash(String, Algorithm, Encode)
```

Returns

The hash.

Parameters

Parameter	Description
String	The data to hash. **Required.**
Algorithm	The hashing algorithm to be used. **Optional.**
Encode	The maximum tracking size of the window. **Optional.**

Remarks

The SRP_Hash function creates a hash based on given input. You supply the input via the String parameter. The data could be as simple as a small string or as complex as data loaded from a large file. You are given several options on the complexity and security of the hash, as well as the output format. First, let's explain cryptographic hashing in greater detail.

What is Hashing?

You may have heard the term hash before. In general, hashing refers to the technique in which some algorithm is applied to some input to produce a unique result. Hashing can be done with very simple and fast algorithms or very complex and slow algorithms. Those familiar with Java or C++ may be familiar with hash tables (sometimes called maps or dictionaries). These data structures use hashing to locate an item based on a key. OpenInsight programmers know the term Linear Hash, referring to the fact that records are located in a table using hashing.

The SRP_Hash focuses on cryptographic hashing. Cryptographic hashing uses much more complicated algorithms to meet security needs. Specifically, this kind of hashing is useful for verification purposes. For instance, when you create an RDK and zip it up, you can use the zip file to create a hash. The recipient of your zip file can use the same hashing algorithm on his/her end to verify that the zip file is not corrupt or compromised. The level of confidence in the hash depends on the algorithm used and its collision resistance.

Collision

All hash algorithms have a limited number of results whereas input can be infinite. After all, we're talking about strings of any size with any number of characters. Therefore, it is always possible that multiple inputs will equate to the same output, i.e., they will collide. The more collision resistant an algorithm, the stronger and more secure it is. An algorithm's strength is usually determined by it's output size. Some algorithms produce 32-bit hashes while others produce 512-bit hashes. More bits means more possible outputs--reducing the likelihood of collision.

Collision is important to security because there needs to be sufficient confidence that the file or data has not been tampered with. So, if an attacker can alter your zip file while ensuring it hashes to the same output value, then your RDK has been compromised. An attacker will only be able to do this through brute force, that is, by using computers to rebuild a zip file over and over until it happens to hash to the same value. This won't take long to do for a 32-bit hash, but no computers today could accomplish this for a 256-bit or greater hash. However, if the hash can be broken using less than brute force, then it is considered weak and insecure.

Weak Hashes

SRP_Hash provides many algorithms, but some of them are now considered too weak for security purposes. In other words, attackers can tamper with files and reconstruct them with the same hash result without using brute force. The weak algorithms are included for backward compatibility, but it is recommended that you use the stronger algorithms if possible.

Hashing vs. Encryption

It's important to point out that hashing is not a type of encryption. Encryption implies that the input can be encrypted and decrypted. Hashes cannot be decrypted. This makes sense when you consider that a hash always has the same size output. If you hashed a 1GB file using an algorithm that generates a 512 byte output, how would you get back to the 4MB? Even the best compression methods available can't do that.

Algorithm

You can use one of the following algorithms by setting the Algorithm parameter to the given algorithm name:

Algorithm	Strength
ADLER32	32-bit
CRC32	32-bit
MD2	128-bit
MD4	128-bit
MD5	128-bit
RIPEMD	(same as RIPEMD-128)
RIPEMD-128	128-bit
RIPEMD-160	160-bit
RIPEMD-256	256-bit
RIPEMD-360	360-bit
SHA	(same as SHA-1)
SHA-1	**160-bit**
SHA-2	(same as SHA-256)
SHA-224	224-bit
SHA-256	256-bit
SHA-384	384-bit
SHA-512	512-bit
TIGER	192-bit
WHIRLPOOL	512-bit

The green algorithms are useful for quick checksums, such as quickly verifying that a file hasn't been altered. They are not at all considered useful for security situations.

The red algorithms indicate weak algorithms. While they are stronger than the green algorithms, they are no longer considered as secure as they used to be due to available cracks. They can still be useful is some situations, but avoid them if security is your top priority.

The bold algorithm is the default algorithm. If you supply a blank or unrecognized algorithm, the bold one is used.

Encode

The Encode parameter allows you to specify how the output is to be encoded. By default, the output is returned as binary data. This will appear as strange characters when viewed from the debugger. If you need to the output to be readable or acceptable for certain transmission protocols, set the Encode property to one of the following encodings:

Encode	Description
HEX	The output contains hexadecimal pairs for each byte
BASE32	The output is encoded in BASE32 for safe transmission
BASE64	The output is encoded in BASE64 for safe transmission

Examples

```
* Hash using defaults
Hash = SRP_Hash("My Test String")

* Hash variable using SHA-1
Hash = SRP_Hash(HashString, "SHA-1")

* Hash variable using RIPEMD-320 and HEX encoding
Hash = SRP_Hash(HashString, "RIPEMD-320", "HEX")

* Hash variable using SHA-512 and BASE32 encoding
Hash = SRP_Hash(HashString, "SHA-512", "BASE32")

* Hash variable using TIGER and BASE64 encoding
Hash = SRP_Hash(HashString, "TIGER", "BASE64")

* Hash file using WHIRLPOOL
OSRead Data from "C:\MyFile.zip" then
 Hash = SRP_Hash(Data, "WHIRLPOOL")
end
```