Salted hashes demystified – A Primer

This primer will provide a basic level explanation of how seeded (or salted) hashes of clear text data are structured / created. The original formalization of this concept comes from RFC-3112[1]. This document is written so that an understanding of this type of functionality becomes possible to anyone with a good computer science foundation. For the purposes of this exploratory journey we will use the Secure Hash Algorithm (SHA-1) hashing algorithm (NIST FIPS 180-2[2], RFC-3174[3]). The salting concepts apply to any one-way hashing algorithm including the MD-5 algorithm (RFC-1321[4]).

Setting the data

The process starts with 2 elements of data:

1. a clear text string (this could represent a password for instance).

2. the salt, a random seed of data. This is the value used to augment a hash in order to ensure that 2 hashes of identical data yield different output.

For the purposes of this document we will analyze the steps within code that perform the necessary actions to achieve the end resulted hash. Cryptographers call this hash a digest. We will not however go into an explanation of the SHA one-way encryption scheme. Readers of this document are encouraged to get information on that subject by following links in the footnote section of this page.

Theoretically, an implementation of SHA-1 as an algorithm takes input, and provides output, that are both in binary form; realistically though digests are typically encoded and stored in a database or in a flat text or XML file. Take slappasswd[5] for instance, it performs the exact functionality described above. We will use it as a black box compiled piece of code for our analysis.

We will also use some custom written code in Java (Appendix A). Our code is merely for testing purposes and will take a clear text string (representing a password in this document) and hash that string into a salted hash. For the purposes of this analysis our code will also accept a user created salt value when creating this hash. Typically this salt value is a randomly generated element created with the highest possible entropy. Then we will base64 encode the final

---

[1] http://www.faqs.org/rfcs/rfc3112.html

[2] http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[3] http://www.faqs.org/rfcs/rfc3174.html

[4] http://www.faqs.org/rfcs/rfc1321.html

[5] http://www.openldap.org

hash output. There is also some very useful Perl code in Appendix B to help those that are more comfortable in that environment.

In pseudocode we generate a salted hash as follows:

```
Get the source string and salt as separate binary objects
Concatenate the 2 binary values
SHA hash the concatenation into SaltedPasswordHash
Base64Encode(concat(SaltedPasswordHash, Salt))
```

In Java this method would look like this:

```
public String createDigest(byte[] salt, String entity) {
        String label = "{SSHA}";

        // Update digest object with byte array of clear text string and salt
        sha.reset();
        sha.update(entity.getBytes());
        sha.update(salt);

        // Complete hash computation, this results in binary data
        byte[] pwhash = sha.digest();

        return label + new String(Base64.encode(concatenate(pwhash, salt)));
}
```

We take a clear text string and hash this into a binary object representing the hashed value of the clear text string plus the random salt. In our Java based example it will be held in a byte array. Then we have the Salt value, which are typically 4 bytes of purely random binary data represented as hexadecimal notation (Base16 as 8 bytes).

SaltedPasswordHash is of length 20 (bytes) in raw binary form (40 bytes if we look at it in hex). Salt is then 4 bytes in raw binary form. The SHA-1 algorithm generates a 160 bit hash string. Consider that 8 bits = 1 byte. So 160 bits = 20 bytes, which is exactly what the algorithm gives us.

The Base64 encoded final string representation of the binary result looks like:

```
{SSHA}B0O0XSYdsk7g9K229ZEr73Lid7HBD9DX
```

Take note here that the final output is a 32-byte string of data. The Base64 encoding process uses bit shifting, masking, and padding as per RFC-3548[6].

A couple of examples of salted hashes using on the same exact clear-text string:

```
[me@a2 ~]# slappasswd -s testing123
{SSHA}72uhy5xc1AWOLwmNcXALHBSzp8xt4giL
```

[6] http://www.faqs.org/rfcs/rfc3548.html

```
[me@a2 ~]# slappasswd -s testing123
{SSHA}zmIAVaKMmTngrUi4UlS0dzYwVAbfBTl7

[me@a2 ~]# slappasswd -s testing123
{SSHA}Be3F12VVvBf9Sy6MSqpOgAdEj6JCZ+0f

[me@a2 ~]# slappasswd -s testing123
{SSHA}ncHs4XYmQKJqL+VuyNQzQjwRXfvu6noa
```

4 runs of slappasswd against the same clear text string each yielded unique end-result hashes. The random salt is generated silently and never made visible.

Extracting the data

One of the keys to note is that the salt is dealt with twice in the process. It is used once for the actual application of randomness to the given clear text string, and then it is stored within the final output as purely Base64 encoded data. In order to perform an authentication query for instance, we must break apart the concatenation that was created for storage of the data. We accomplish this by splitting up the binary data we get after Base64 decoding the stored hash.

In pseudocode we would perform the extraction and verification operations as such:

```
Strip the hash identifier from the Digest
Base64Decode(Digest, 20)
Split Digest into 2 byte arrays, one for bytes 0 – 20(pwhash), one for bytes 21 – 32 (salt)
Get the target string and salt as separate binary object
Concatenate the 2 binary values
SHA hash the concatenation into targetPasswordHash
Compare targetPasswordHash with pwhash
Return corresponding Boolean value
```

In Java, a method to extract the salt from the original hash, salt and hash the new clear text object, and then compare the 2 looks like this:

```
public boolean checkDigest(String digest, String entity) {
        boolean valid = true;

        digest = digest.substring(6); // ignore the {SSHA} hash ID

        // extract the hashed data into hs[0], salt into hs[1]
        byte[][] hs = split(Base64.decode(digest), 20);
        byte[] hash = hs[0];
        byte[] salt = hs[1];

        // Update digest object with byte array of clear text string and salt
        sha.reset();
        sha.update(entity.getBytes());
        sha.update(salt);
```

```
                // Complete hash computation, this is now binary data
                byte[] pwhash = sha.digest();

                if (!MessageDigest.isEqual(hash, pwhash)) {
                        valid = false;
                        System.out.println("Hashes DON'T match: " + entity);
                }

                if (MessageDigest.isEqual(hash, pwhash)) {
                        valid = true;
                        System.out.println("Hashes match: " + entity);
                }

                return valid;
        }
```

Our job is to split the original digest up into 2 distinct byte arrays, one of the left 20 (0 - 20 including the null terminator) bytes and the other for the rest of the data. The left 0 – 20 bytes will represent the salted binary value we will us for a byte-by-byte data match against the new clear text presented for verification. The string presented for verification will have to be salted as well. The rest of the bytes (21 – 32) represent the random salt which when decoded will show the exact hex characters that make up the once randomly generated seed.

We are now ready to verify some data. Lets start with the 4 hashes presented earlier. We will run them through our code to extract the random salt and then using that verify the clear text string hashed by slappasswd. First, lets do a verification test with an erroneous password; this should fail the matching test:

```
[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -v -c
{SSHA}72uhy5xc1AWOLwmNcXALHBSzp8xt4giL Test123
Hash extracted (in hex): ef6ba1cb9c5cd4058e2f098d71700b1c14b3a7cc
Salt extracted (in hex): 6de2088b
Hash length is: 20 Salt length is: 4
Hash presented in hex: 256bc48def0ce04b0af90dfd2808c42588bf9542
Hashes DON'T match: Test123
```

The match failure test was successful as expected. Now let’s use known valid data through the same exact code:

```
[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -v -c
{SSHA}72uhy5xc1AWOLwmNcXALHBSzp8xt4giL testing123
Hash extracted (in hex): ef6ba1cb9c5cd4058e2f098d71700b1c14b3a7cc
Salt extracted (in hex): 6de2088b
Hash length is: 20 Salt length is: 4
Hash presented in hex: ef6ba1cb9c5cd4058e2f098d71700b1c14b3a7cc
Hashes match: testing123

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -v -c
{SSHA}zmIAVaKMmTngrUi4UlS0dzYwVAbfBTl7 testing123
Hash extracted (in hex): ce620055a28c9939e0ad48b85254b47736305406
Salt extracted (in hex): df05397b
```

```
Hash length is: 20 Salt length is: 4
Hash presented in hex: ce620055a28c9939e0ad48b85254b47736305406
Hashes match: testing123

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -v -c
{SSHA}Be3F12VVvBf9Sy6MSqpOgAdEj6JCZ+0f testing123
Hash extracted (in hex): 05edc5d76555bc17fd4b2e8c4aaa4e8007448fa2
Salt extracted (in hex): 4267ed1f
Hash length is: 20 Salt length is: 4
Hash presented in hex: 05edc5d76555bc17fd4b2e8c4aaa4e8007448fa2
Hashes match: testing123

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -v -c
{SSHA}ncHs4XYmQKJqL+VuyNQzQjwRXfvu6noa testing123
Hash extracted (in hex): 9dc1ece1762640a26a2fe56ec8d433423c115dfb
Salt extracted (in hex): eeea7a1a
Hash length is: 20 Salt length is: 4
Hash presented in hex: 9dc1ece1762640a26a2fe56ec8d433423c115dfb
Hashes match: testing123
```

When good data is passed in we see that the data matches are good. Now let's re-create the same exact hashes that slappasswd created since we know how to extract the salt values.

```
[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -s 6de2088b testing123
{SSHA}72uhy5xc1AWOLwmNcXALHBSzp8xt4giL

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -s df05397b testing123
{SSHA}zmIAVaKMmTngrUi4UlS0dzYwVAbfBTl7

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -s 4267ed1f testing123
{SSHA}Be3F12VVvBf9Sy6MSqpOgAdEj6JCZ+0f

[me@a2 ~]# java -classpath .:./xercesImpl.jar TestSSHA -s eeea7a1a testing123
{SSHA}ncHs4XYmQKJqL+VuyNQzQjwRXfvu6noa
```

These hashes are exact matches to the ones generated by slappasswd.

The demystification of this functionality is evident. We see that salting hashed data does indeed add another layer of security to the clear text one-way hashing process. But we also see that salted hashes should also be protected just as if the data was in clear text form. Now that we have seen salted hashes actually work you should also realize that in code it is possible to extract salt values and use them for various purposes. Obviously the usage can be on either side of the colored hat line, but the data is there.

About the authors:

Author: Andres Andreu, CISSP-ISSAP <andres [at] neurofuzz dot com>

Perl code (Appendix B) provided by David Shu <ydpanda [at] gmail dot com>

Appendix A

```
/**
 * This class provides a test framework for an understanding of SSHA digests.
 *
 * @author Andres [at] andreugroup dot com (Andres Andreu)
 */

import org.apache.xerces.impl.dv.util.Base64;
import java.security.MessageDigest;

public class TestSSHA {

	private boolean verbose = false;

	private MessageDigest sha = null;

	/**
	 * public constructor
	 */
	public TestSSHA() {
		verbose = false;

		try {
			sha = MessageDigest.getInstance("SHA-1");
		} catch (java.security.NoSuchAlgorithmException e) {
			System.out.println("Construction failed: " + e);
		}
	}

	/**
	 * Create Digest for each entity values passed in
	 *
	 * @param salt
	 *		byte array to set the base for the encryption
	 * @param entity
	 *		string to be encrypted
	 * @return string representing the salted hash output of the encryption
	 *		operation
	 */
	public String createDigest(byte[] salt, String entity) {
		String label = "{SSHA}";

		// Update digest object with byte array of the source clear text
		// string and the salt
		sha.reset();
		sha.update(entity.getBytes());
		sha.update(salt);

		// Complete hash computation, this results in binary data
		byte[] pwhash = sha.digest();

		if (verbose) {
			System.out.println("pwhash, binary represented as hex: "
					+ toHex(pwhash) + " \n");
			System.out.println("Putting it all together: ");
			System.out.println("binary digest of password plus binary salt: "
					+ pwhash + salt);
			System.out.println("Now we base64 encode what is respresented above this line ...");
		}

		return label + new String(Base64.encode(concatenate(pwhash, salt)));
	}

	/**
	 * Check Digest against entity
	 *
	 * @param digest
	 *		is digest to be checked against
```

```
 * @param entity
 *          entity (string) to be checked
 * @return TRUE if there is a match, FALSE otherwise
 */
public boolean checkDigest(String digest, String entity) {
        boolean valid = true;

        // ignore the {SSHA} hash ID
        digest = digest.substring(6);

        // extract the SHA hashed data into hs[0]
        // extract salt into hs[1]
        byte[][] hs = split(Base64.decode(digest), 20);
        byte[] hash = hs[0];
        byte[] salt = hs[1];

        // Update digest object with byte array of clear text string and salt
        sha.reset();
        sha.update(entity.getBytes());
        sha.update(salt);

        // Complete hash computation, this is now binary data
        byte[] pwhash = sha.digest();

        if (verbose) {
                System.out.println("Salted Hash extracted (in hex): " + toHex(hash)
                                + " " + "\nSalt extracted (in hex): " + toHex(salt));
                System.out.println("Hash length is: " + hash.length
                                + " Salt length is: " + salt.length);
                System.out.println("Salted Hash presented in hex: " + toHex(pwhash));
        }

        if (!MessageDigest.isEqual(hash, pwhash)) {
                valid = false;
                System.out.println("Hashes DON'T match: " + entity);
        }

        if (MessageDigest.isEqual(hash, pwhash)) {
                valid = true;
                System.out.println("Hashes match: " + entity);
        }

        return valid;
}

/**
 * set the verbose flag
 */
public void setVerbose(boolean verbose) {
        this.verbose = verbose;
}

/**
 * Combine two byte arrays
 *
 * @param l
 *          first byte array
 * @param r
 *          second byte array
 * @return byte[] combined byte array
 */
private static byte[] concatenate(byte[] l, byte[] r) {
        byte[] b = new byte[l.length + r.length];
        System.arraycopy(l, 0, b, 0, l.length);
        System.arraycopy(r, 0, b, l.length, r.length);
        return b;
}

/**
 * split a byte array in two
```

```
 *
 * @param src
 *          byte array to be split
 * @param n
 *          element at which to split the byte array
 * @return byte[][] two byte arrays that have been split
 */
private static byte[][] split(byte[] src, int n) {
        byte[] l, r;
        if (src == null || src.length <= n) {
                l = src;
                r = new byte[0];
        } else {
                l = new byte[n];
                r = new byte[src.length - n];
                System.arraycopy(src, 0, l, 0, n);
                System.arraycopy(src, n, r, 0, r.length);
        }
        byte[][] lr = { l, r };
        return lr;
}

private static String hexits = "0123456789abcdef";

/**
 * Convert a byte array to a hex encoded string
 *
 * @param block
 *          byte array to convert to hexString
 * @return String representation of byte array
 */
private static String toHex(byte[] block) {
        StringBuffer buf = new StringBuffer();

        for (int i = 0; i < block.length; ++i) {
                buf.append(hexits.charAt((block[i] >>> 4) & 0xf));
                buf.append(hexits.charAt(block[i] & 0xf));
        }

        return buf + "";
}

/**
 * Convert a String hex notation to a byte array
 *
 * @param s
 *          string to convert
 * @return byte array
 */
private static byte[] fromHex(String s) {
        s = s.toLowerCase();
        byte[] b = new byte[(s.length() + 1) / 2];
        int j = 0;
        int h;
        int nibble = -1;

        for (int i = 0; i < s.length(); ++i) {
                h = hexits.indexOf(s.charAt(i));
                if (h >= 0) {
                        if (nibble < 0) {
                                nibble = h;
                        } else {
                                b[j++] = (byte) ((nibble << 4) + h);
                                nibble = -1;
                        }
                }
        }

        if (nibble >= 0) {
                b[j++] = (byte) (nibble << 4);
```

```
                }

                if (j < b.length) {
                        byte[] b2 = new byte[j];
                        System.arraycopy(b, 0, b2, 0, j);
                        b = b2;
                }

                return b;
        }

        private static void usage(String className) {
                System.out.print("usage: "
                        + className
                        + " [-v] -s salt SourceString ...\n"
                        + "   or: "
                        + className
                        + " [-v] -c EncodedDigest SourceString ...\n"
                        + "       salt must be in hex.\n"
                        + "       digest contains SHA-1 hash or salted hash, base64 encoded.\n");
        }

        /**
         * Main program for command line use and testing
         */
        public static void main(String[] args) {
                TestSSHA sh = new TestSSHA();
                String className = "TestSSHA";

                if (args.length <= 1) {
                        usage(className);
                        return;
                }

                int i = 0;

                if (args[i].equals("-v")) {
                        ++i;
                        sh.setVerbose(true);
                }

                // -c validate data against digest
                // -s using a seed for randomness
                if (args[i].equals("-c")) {
                        ++i;
                        String digest = args[i++];
                        sh.checkDigest(digest, args[i]);
                } else if (args[i].equals("-s")) {
                        // generate digest from data passed in
                        byte[] salt = {};
                        ++i;
                        salt = fromHex(args[i++]);
                        System.out.println(sh.createDigest(salt, args[i]));
                } else {
                        usage(className);
                }
        }
}
```

Appendix B - Perl

```
#
# Author :          David Shu <ydpanda [at] gmail dot com>
# Created :         5/26/2005
# Description:      This is a collection of functions that will assist in
#                   working with salted SHA (SSHA) passwords.
#
use Digest::SHA1;
use MIME::Base64;

#
# Description : Extracts the prefix portion of the hashed password
# Parameters :      hashed password => (required; the hashed password must contain
#                   the appropriate prefix)
# Return : scheme (string)
#
sub getPassScheme
{
        my $hashed_pass = shift;

        # extract prefix from hash
        $hashed_pass=~m/\{([^\}]*)/;
        return $1;
}


#
# Description : Extracts the hash portion of the hashed password
# Parameters :      hashed password => (required; the hashed password must contain
#                   the appropriate prefix)
# Return : hash (string)
#
sub getPassHash
{
        my $hashed_pass = shift;

        # extract hash from passwordhash
        $hashed_pass=~m/}([^\s]*)/;
        return $1;
}

#
# Description :     Generate a SHA or SSHA hash
# Parameters :      password => clear text (required)
#                   salted => boolean (optional; default = FALSE)
#                   salt => hexString (optional; default = ""; a random salt will be
#                           generated if none is provided
# Return :          Hash (string)
#
sub generateSHA
{
        my $password = shift;
        my $salted = shift;
        my $salt = shift;

        if($salted && $salt eq ""){
                $salt = generateHexSalt();
        }

        my $hashed_pass = "";
        my $ctx = Digest::SHA1->new;
        $ctx->add($password);
        print $password;
        if($salted){
                print $salt;
                $salt = pack("H*", $salt);
                $ctx->add($salt);
                $hashed_pass = encode_base64($ctx->digest . $salt ,'');
```

```
		}
		else{
			$hashed_pass = encode_base64($ctx->digest,'');
		}

		return $hashed_pass;

}

#
# Description : Generate a SHA or SSHA hashed password; same as generateSHA
#			but adds the appropriate prefix
# Parameters :	password => clear text (required)
#			salted => boolean (optional; default = FALSE)
#			salt => hexString (optional; default = ""; a random salt will be
#				generated if none is provided
# Return :		Hashed Password (string)
#
sub generateSHAWithPrefix
{
		my $password = shift;
		my $salted = shift;
		my $salt = shift;
		my $hashed_pass = "";

		if(!$salted){
			$hashed_pass = "{SHA}" . generateSHA($password,$salted,$salt);
		}else{
			$hashed_pass = "{SSHA}" . generateSHA($password,$salted,$salt);
		}

		return $hashed_pass;
}

#
# Description : Randomly generate a 4 byte hex-based string
# Parameters :	N/a
# Return : Hex based salt (string)
#
sub generateHexSalt
{
		# RANDOM KEY PARAMETERS
		my @keychars = ("0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f");
		my @keychars_initial = ("1","2","3","4","5","6","7","8","9","a","b","c","d","e","f");
		my $length = 8;

		# RANDOM KEY GENERATOR
		my $randkey = "";
		for ($i=0;$i<$length;$i++) {
			if($i==0){
				$randkey .= $keychars_initial[int(rand(15))];
			}
			else{
				$randkey .= $keychars[int(rand(16))];
			}
		}

		return $randkey;

}

#
# Description : Extracts the hex based salt that was used in the hashed password
# Parameters :	hashed password => (required; the hashed password must contain
#			the appropriate prefix)
# Return : Hex based salt (string)
#
sub extractSalt
{
		my $hashed_pass=shift;
```

```
	my $hash = getPassHash($hashed_pass);
	my $ohash = decode_base64($hash);
	my $osalt = substr($ohash, 20);
	return join("",unpack("H*",$osalt));
}

#
# Description : Compare the hashed password with the clear text password;
#			Currently this only supports 3 password schemes (all are
#			base64 encoded):
#				1) SSHA (sha1 algorithm)
#				2) SHA (sha1 algorithm)
#				3) MD5
# Parameters :	hashed password => (required; the hashed password must contain
#			the appropriate prefix)
#			cleartext password => (required)
# Return :		1/0
#
sub validatePassword
{
	$hashed_pass = shift;
	$clear_pass = shift;
	$scheme = lc(getPassScheme($hashed_pass));
	$hash = getPassHash($hashed_pass);
	$clear_pass=~s/^\s+//g;
	$clear_pass=~s/\s+$//g;
	$retval = 0;
	if($scheme eq "ssha"){
		$salt = extractSalt($hashed_pass);
		$hpass = generateSHA($clear_pass,1,$salt);

		if($hash eq $hpass){
			$retval = 1;
		}
	}
	elsif($scheme eq "sha"){
		$hpass = generateSHA($clear_pass,0,"");
		if($hash eq $hpass){
			$retval = 1;
		}
	}
	else{
		$hpass = encode_base64(pack("H*",md5($clear_pass)));
		if($hash eq $hpass){
			$retval = 1;
		}
	}

	return $retval;
}

1;
```