ANDRES ANDREU

Cracking LDAP Salted SHA Hashes

Difficulty

▮▮▯

In the realm of Web applications, user data is traditionally stored in an accessible manner due to the fact that it is needed for all future use by any authorized user(s). User data contains login credentials where the password (and potentially usernames and other attributes) must be stored for future reference.

To protect against a myriad of attacks, including malicious injection attacks and the exposure of archived data, user data, particularly passwords are stored in a non-reversible, non clear-text form. Interestingly enough, this same thought process and storage technique has carried over to the desktop login space with desktop OS logins now tied into Active Directory and other LDAP based back-ends.

Storing user data such as passwords in plain text represents a potential security risk. In the event of a breach, crackers gaining data access via software flaws (such as improper input validation) could gain unauthorized access to a multitude of systems. These days the risk is exponentially higher than in the past due to developments in Internet/Web based single-password and single-sign-on (SSO) technologies. This access could lead to malicious activity of any arbitrary real user, with the permissions of that user. The extent of these actions are limited only by your imagination and what access the target application has been allowed. To mitigate this security risk the industry generally has relied upon password data being stored as the output of a one-way hashing algorithm. Although, given the elevated sophistication of modern-day attack techniques coupled with the way one-way hash algorithms natively work, vanilla flavoured one-way hashing algorithms have really outlived their effectiveness. The need for randomness, which has come from the age old techniques of the Unix world, became critical to the industry. The specifics of this have come in the form of salting the one-way hashes to add randomness to the stored output. This randomness increases the level of work required for a successful crack, in the event of a breach.

A one-way hash is a binary computed value of fixed length that is normally represented in either Base64 or Hexadecimal encoded notation. The idea behind using non-reversible hashes is that they should unequivocally identify a set of clear text data as being valid (through some form of comparison). Some experts consider this a *digital fingerprint* of clear text.

A salt is a randomly computed set of data to alter the output of any one-way hashing algorithm (in the context of this article). These sets of data traditionally come in 4 or 8 byte blocks. With regard to the randomness aspect of the salt, true randomness in computing environments has been argued time and time again and is beyond the scope of this article. Suffice it to say that most sets of web based code that generate random salts do so utilizing pseudo-random functions. This article does not attack the mathematical foundation of randomness as used in today's web based computing environments, the randomness of the salt value is actually of no relevance for the techniques discussed here.

The reason for using a salt in conjunction with one-way hashed data should be an obvious one by now. Advanced technologies such as PKI, client-side X509 certificates, and biometric solutions have been around for some time now, but the reality of the *Information Technology* (IT) industry is that

WHAT YOU WILL LEARN...

How LDAP Salted SHA (SSHA) Hashes are structured,

How to employ modern day tools to crack LDAP SSHA hashes,

Why LDAP SSHA hashes should be treated as if they are clear-text data.

WHAT YOU SHOULD KNOW...

Basic knowledge of compiling C source code in Linux (x-86 based),

Basic scripting in standard languages (some code and/or snippets given in Python, Ruby & PHP),

Basic knowledge of encodings of binary data,

Concepts of storage techniques for user password data.

simple username/password combinations are still the most prevalent authentication model, especially in the space of web applications and related technologies such as SSL based VPNs. A major player in the modern-day space of user data storage and authentication/authorization services is the *Lightweight Directory Access Protocol* (LDAP).

LDAP is a protocol that can be back-ended by numerous different technologies, such as XML or traditional relational databases. Within the possible schemas used by LDAP, implementations of certain objectClass structures are accepted as industry standards, and they come ready to store sensitive user data such as passwords. The obvious one is inetOrgPerson described in RFC-2798 that has an attribute named *userPassword*. Any standards based implementation of the LDAP protocol will support this objectclass. Software engineers can write code that interacts with these attributes and have confidence that their code will function in a product agnostic fashion based on the LDAP servers adherence to standards. The userPassword attribute traditionally supports password data in one of the following forms:

- CLEAR – literally clear text data,
- BASE64 – Base64 encoded representation of clear text data,
- MD5 – using the Message Digest 5 one-way hash algorithm,
- SHA – using the one-way Secure Hashing Algorithm.

Clear text data is obviously insecure and Base64 encoded data is really no better. Both of these methods do nothing to protect your user data if an attacker manages to penetrate your target LDAP data source. One-way hashing algorithms have come to the rescue to reference stored data because they are not reversible. Unfortunately they are consistent in the way they operate so an attacker could easily figure out that some users in your LDAP data source all have the same passwords based on identical hashes. Listing 1 shows you Python code that will generate a SHA1 hash of some clear text data. You can run this Python script numerous times against the same clear text data and see that the output does not change. This is a concern because many entities out there use a consistent default password for all new and/or temporary users within their infrastructure. In operations that utilize SHA1 hashes you will find code that performs this type of action and outputs to an attribute (typically *userPassword.*)

To make matters tougher for the IT security staff, there are online services that attempt to identify a collision for hash data (numerous MD5 instances are available online), brute-force crackers for specific hash forms, rainbow crack programs and huge rainbow tables that can be pulled down with torrent technologies. This is pretty disturbing because it becomes really difficult to protect sensitive user data nowadays.

So the suggestion of security experts is to use a salt along with strong one-way hashing algorithms. MD5 and SHA1 have both seen successful collisions in security research at this point (see hyperlinks provided at the end of this article). This does not mean that anyone can cause such a situation as it requires expert level knowledge and decent computing power (even though the BOINC based collision projects minimise the need for real knowledge). The strong hashing algorithms commonly used today are MD5, SHA and the SHA2 family of algorithms along with random salts, so in the more sophisticated LDAP implementations you will now run across the following identifiers in the data stored in userPassword attributes:

- SMD5 – salted MD5,
- SSHA – salted SHA1,
- SSHA256 – salted SHA256,
- SSHA384 – salted SHA384,
- SSHA512 – salted SHA512.

This usage of a salt means that for every user object, a unique hash is used to store password data. If two users have the same password and a salt is used then normal analysis or the human eye could never identify this fact, as the stored hashes will not visually match. It is important that salt is generated with the highest possible level of entropy in order to optimise the use of this technology. Listing 2 provides you with a Python2.5 script that generates salted SHA hashes, this particular script encompasses the currently common family of SHA2 hash algorithms. If you run this script numerous times with the same clear text string you will always get unique outputs in the resulting hashes. This output emulates what a more sophisticated environment would do if they were in an LDAP realm.

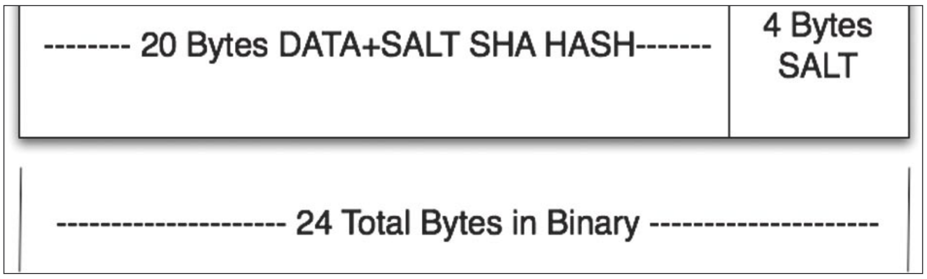
-------- 20 Bytes DATA+SALT SHA HASH-------
4 Bytes SALT
-------------------- 24 Total Bytes in Binary --------------------

Figure 1. *SSHA Hash structure – A visual depiction of the hash data structure (SHA1 – 4 byte salt) detailed in the output from the code in Listing 3*

Listing 1. *Simple code to generate unsalted SHA1 hashes of clear-text data*

```
import sys, sha, base64

ctx = sha.new( sys.argv[1] )
hash = "\n{SHA}" + base64.b64encode( ctx.digest() )
print hash

Assuming the code above is saved in a file named genSHA.py then a sample run (using the
                       clear-text string "test") would look something like this:
$ python genSHA.py test

{SHA}qUqP5cyxm6YcTAhz05Hph5gvu9M=
```

The Salt Is Always Available

There is no black magic involved with these salted hashing techniques. The salt part of a salted hash must be made available to the code/application(s) that interacts with it for proper functionality to exist. Otherwise it would be impossible for an application to verify data that gets submitted to it. When interacting with salted hashes in an authentication scenario, for instance, an application will generally follow these steps (these are specific to interaction with open standards based LDAP servers but the concept applies to other scenarios as well):

- `getSaltedHash` (from internal storage – LDAP, database, etc)
- `detectHashingAlgorithm` (analysing the stored salted hash makes it possible to determine the one-way algorithm used – in LDAP there is usually an identifying prefix like `{SSHA}`; there is also the fact that one way hashes output statically sized data)
- `extractSalt` (by knowing the algorithm, the code can then perform this action from the stored hash it just acquired)
- `getClearTextData` (this would be data submitted by a user or other

Listing 2. *Simple Python2.5 code to generate salted SHA1 and SHA2 hashes of clear-text data*

```
import hashlib, binascii, sys
from base64 import b64encode
from random import randrange
str = sys.argv[1]
saltsize = int(sys.argv[2])
if saltsize <> 4 and saltsize <> 8:
    print "Lets stick to what is out there, 4 or 8 byte salt
                     sizes ...\n\n"
    sys.exit(0)
print "generating simple random salt of %d bytes...\n" %
                     saltsize
salt = ''
for n in range(saltsize/2):
  salt += chr(randrange(256))
salt = binascii.hexlify(salt)
print "SHA1"
m = hashlib.sha1()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA}" + w
print "Base64 encoded:\n%s" % w
print "%s" % wo
print
print "SHA256"
m = hashlib.sha256()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA256}" + w
print "Base64 encoded:\n%s" % w
print "%s" % wo
print
print "SHA384"
m = hashlib.sha384()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA384}" + w
print "Base64 encoded:\n%s" % w
print "%s" % wo
print
print "SHA512"
m = hashlib.sha512()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA512}" + w
print "Base64 encoded:\n%s" % w
print "%s" % wo
print
Assuming the code above is saved in a file named genSSHA.py
                      then a sample run (using the clear-text
                      string "test") would look something like
                      this:
$ python genSSHA.py test 4
generating simple random salt of 4 bytes...
SHA1
In Hex:
98f161a269d8d3b5567749420f8024a27a9844c0
Base64 encoded:
mPFhomnY07VWd0lCD4AkonqYRMBjNTg1
{SSHA}mPFhomnY07VWd0lCD4AkonqYRMBjNTg1
SHA256
In Hex:
a1efb0cc52ed95dca4536d6d21d68044ca742fec269326992f5d9279e7cc
                      cf48
Base64 encoded:
oe+wzFLtldykU21tIdaARMp0L+wmkyaZL12SeefMz0hjNTg1
{SSHA256}oe+wzFLtldykU21tIdaARMp0L+wmkyaZL12SeefMz0hjNTg1
SHA384
In Hex:
4034f8cedd3b59e44810c113b88c7b04475193aeab6629034994b1c71e8213
                      392bd5f07d25e1b2d42547150b7679618c
Base64 encoded:
QDT4zt07WeRIEMETuIx7BEdRk66rZikDSZSxxx6CEzkr1fB9JeGy1CVHFQt2eW
                      GMYzU4NQ==
{SSHA384}QDT4zt07WeRIEMETuIx7BEdRk66rZikDSZSxxx6CEzkr1fB9JeGy1
                      CVHFQt2eWGMYzU4NQ==
SHA512
In Hex:
ff24c1b3cf119bf449478c1931a645d240b3454213531ee3fd1ebe2d24a15
                      017c7aacdebdae4d181b6d62696dcb1fb200466
                      84096bf2ae71bf1fd20409ca3dfb
Base64 encoded:
/yTBs88Rm/RJR4wZMaZF0kCzRUITUx7j/R6+LSShUBfHqs3r2uTRgbbWJpbcsf
                      sgBGaECWvyrnG/H9IECco9+2M1ODU=
{SSHA512}/yTBs88Rm/RJR4wZMaZF0kCzRUITUx7j/R6+LSShUBfHqs3r2uTRgb
                      bWJpbcsfsgBGaECWvyrnG/H9IECco9+2M1ODU=
Clearly there is a difference in the byte size of the hashes
                      generated and the larger ones represent
                      a greater work factor for a successful
                      crack. Although they require more
                      effort, they are nevertheless crackable
                      via collisions as long as the cracker/
                      attacker knows where the salt is and how
                      to extract it.
```

ChicagoCon combines a professional security conference, certification training and a hacker con into a single, unique event.

Training

Management: CISSP - Security Management (7-Day Course), PMP/CAPM - Project Management, SOX Compliance
Beginner: Ethical Hacking Fundamentals (Network+/Security+), CCNA (Covers New Material), MS ISA Server 2006 (70-351)
Intermediate: CEH - Certified Ethical Hacker, CHFI - Certified Forensics Investigator, ECSA - Certified Security Analyst
Advanced: CEPT - Certified Expert Penetration Tester, Web Application Hacking, ***First Time EVER BackTrack to the Max***

Special 2-Day Workshop provided by the SANS Institute
Cutting-Edge Hacking Techniques - Hands On
Course taught by keynoter, Matthew Carpenter & written by Ed Skoudis

Keynotes

SA Patrick M. Geahan - FBI Cyber Crimes
Ralph Echemendia - Hacking Instructor
Luke McOmie - TruTV's Tiger Team
Mike Murray - Director Neohapsis Labs
Matthew Carpenter - SANS, Intelguardians

"Con" Activities

- 2 Days of hour-long presentations
- Breakout Sessions on Professional Pen Testing, Microsoft Technologies, SCADA Security and Open Source Hacking Tools
- Hacking contests, book giveaways and more
- All for only $100 (Free for Training Students)

White Hats Come Together In Defense of the Digital Frontier

ChicagoCon 2008s

May 12 - 18

Presented by:
The Ethical Hacker Network

ChicagoCon is a Service Mark of THE DIGITAL CONSTRUCTION COMPANY

WWW.CHICAGOCON.COM

application, in an authentication request this is the password)

- `combineClearTextDataAndSalt` (combine the submitted data with the recently extracted salt)
- `applyAlgorithm` (apply the detected hashing algorithm to the result of the previous step)
- `compareValues` (compare the original stored hash from the internal data store and the now salted and hashed data submitted to your code)

A snippet of PHP code performing some of these actions on salted SHA1 data could look something like this:

```
//strip out {SSHA}
$encrypted = substr($encrypted, 6);
// $hash now has binary data
$hash = base64_decode($encrypted);
// extract salt from binary data
$salt = substr($hash, 20);
if ($hash == mHash(MHASH_SHA1,
                        $cleartext .
                        $salt)) {
   return true;
}
```

Listing 3. *A small ruby script illustrating the process of data being put through a one-way salted hashing algorithm*

```
#!/usr/bin/env ruby
# For illustrative purposes a static clear text string and salt have been used
require 'sha1'
require 'base64'

salt = 'SALT'
pass = 'testing'
conc = pass+salt

sha = Digest::SHA1.digest(conc)
puts "SHA1 Digest"
puts "In Binary: #{sha}"
puts "Length of Binary: #{sha.length}"
puts "\nIn Hex: #{sha.unpack('H*').to_s}"
puts "Length of Hex: #{sha.unpack('H*').to_s.length}"

puts "\nSalt\nIn ASCII: #{salt}"
puts "In Hex: #{salt.unpack('H*')}"

concsalt = sha+salt
puts "\nSHA1 Hash plus salt (RAW): #{concsalt}"
puts "SHA1 Hash plus salt (RAW - Length): #{concsalt.length}"
puts "SHA1 Hash plus salt (Hex): #{concsalt.unpack('H*')}"
puts "SHA1 Hash plus salt (Hex - Length): #{concsalt.unpack('H*').to_s.length}"

hash = "{SSHA}"+Base64.encode64(concsalt).chomp!
puts "\nSalted SHA1 Hash(Base64 Encoded): #{hash}"

A run of this script generates the following output:
$ ruby genSSHA.rb
SHA1 Digest
In Binary: yP?`x&%u??V?Cf9M
Length of Binary: 20

In Hex: 790250aa1e6078262575a2c6991856ec4366394d
Length of Hex: 40

Salt
In ASCII: SALT
In Hex: 53414c54

SHA1 Hash plus salt (RAW): yP?`x&%u??V?Cf9MSALT
SHA1 Hash plus salt (RAW - Length): 24
SHA1 Hash plus salt (Hex): 790250aa1e6078262575a2c6991856ec4366394d53414c54
SHA1 Hash plus salt (Hex - Length): 48


Salted SHA1 Hash(Base64 Encoded): {SSHA}eQJQqh5geCYldaLGmRhW7ENmOU1TQUxU
```

Even though this article is focused on LDAP salted hashes and how they are commonly used in the industry, the concepts described apply to any similar technique for storage of this type of data. Some solutions store all of the relevant data in a database table where one field stores the salted hash and another field stores the related salt value. The point that needs to be understood is that the salt is somewhere and an attacker will try to get at it. If the salt is compromised then brute-force and dictionary attacks become possible as you will shortly see.

The Structure of the Hashed Data

In the case of LDAP salted hashes the structure of the final hashed data looks something like this (again, this is specific to a salted SHA1 hash with a 4 byte salt but think about it all in a wider scope):

There is a salt value, it is in binary form. This salt consists of 4 bytes of purely random binary data represented as hexadecimal notation (Base16 as 8 bytes). The final salted hash is of length 20 bytes in raw binary form (40 bytes if you look at it in hex). The SHA1 algorithm ultimately generates a 160 bit hash string. At 8 bits per byte that equates to 20 bytes. Figure 1 should give you a simple and clear visual depiction of this. When dealing with data that has already been hashed you must obviously understand the structure well. The goal is to deconstruct this stored data in order to get to the salt and some stored data that can be used for hash comparison, thus the stored hash must be split apart. In the case of SHA1 the goal is to split up the original hash into 2 distinct byte arrays, one for the left 20 bytes (0 – 20 including the null terminator) and one for the rest of the data. The left 0 – 20 bytes will represent the salted binary value that we will use for a byte-by-byte data match against the new clear text presented for verification. The inbound clear text 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.

Take a look at the Ruby script in Listing 3, it outputs interesting details along the way of the salted SHA1 creation process. It gives you a good understanding of what normally takes place under the hood in code that generates these types of hashes. In a real world scenario the resulting hash seen in the very last line of the output is what would be stored in the LDAP attribute (in Listing 3 it would be: `{SSHA}eQJQqh5geCYldaLGmR hW7ENmOU1TQUxU`). Figure 1 should visually reinforce the final salted hash binary data structure at hand.

Cracking the Hash

Based on what you have learnt, you will see that everything necessary to crack a salted SHA hash from LDAP is readily available. There have been tools written to accomplish this. John the Ripper, or John as it is commonly referred to, has a patch available that gives it the ability to crack salted SHA1 hashes. John is a multi-purpose cracking utility and it is very powerful, but it is somewhat limited due to the lack of support for the SHA2 family of algorithms. Our focus for this article is a very specific type of hash based on the entire SHA family (except for SHA224 since it does not seem widely used in the industry) topping off at SHA512. SSHA Attack is a tool written for this purpose exactly, as it supports attacks on salted SHA1, SHA256, SHA384 & SHA512 hashes as commonly used in data stores accessed via LDAP.

SSHA Attack is written in C to maximize its performance. It uses the authentication concept explained earlier for the crack attack on a given hash. If you think about it simplistically under the hood a crack attack of this sort is nothing more than performing the same exact action as an authentication query when salted hashes are in place. This technique does not attack the hashing algorithm at all, it merely uses it for the purpose of hash comparison, the output of these algorithms is what we are attacking.

Technique aside, it is critical to understand the structure of the data that was explained earlier. Extracting the salt from the salted hashes is at the heart of the attack process and has a direct impact on the success of a hash crack effort. Analyse the snippet of code in Listing 4, it shows you where SSHA Attack extracts the salt from a salted SHA hash based on the hash type.

With the salt in hand, SSHA Attack applies it to the clear text data. In the scenario of an attack with SSHA Attack the clear text data would either come from the brute-force process or a dictionary file specified at run time. These steps are seen in the source code as such:

Table 1. *For size 1*

a	b	c	d

```
...
//copy requestPW to unsigned array
strcpy(finalRequestPW, requestPW);
//cat the binary salt to binary array
strcat(finalRequestPW, tempSalt);
```

Table 2. *For size 2*

aa	ba	ca	da
ab	bb	cb	db
ac	bc	cc	dc
ad	bd	cd	dd

Listing 4. *Snippet from SSHA Attack outlining the salt extraction process from a salted hash that has been acquired from an LDAP implementation*

```
// grab salt from temp & cpy to tempSalt
if (strcmp(hashtype, "SHA1") == 0) {
   strcpy(tempSalt, temp + 20);
} else if (strcmp(hashtype, "SHA224") == 0) {
   strcpy(tempSalt, temp + 28);
} else if (strcmp(hashtype, "SHA256") == 0) {
   strcpy(tempSalt, temp + 32);
} else if (strcmp(hashtype, "SHA384") == 0) {
   strcpy(tempSalt, temp + 48);
} else if (strcmp(hashtype, "SHA512") == 0) {
   strcpy(tempSalt, temp + 64);
}
At the end of this code snippet the array tempSalt will hold the value for the salt
                        from the hash. Notice how the intimate knowledge of the hash
                        sizes are used to calculate where the salt extraction starts.
                        With this element of data, the crack attacks can commence. It
                        should be obvious by now that this salt will be used to generate
                        hashes of clear text data based on the cracking methodology you
                        chose to use.
```

Listing 5. *C Snippet from SSHA Attack's GenerateHash function*

```
...
EVP_MD_CTX_init(&mdctx);
// Initialize the digest
EVP_DigestInit_ex(&mdctx, md, NULL);
// Add the clear text password to the digest
EVP_DigestUpdate(&mdctx,
                  value,
                  (unsigned int) strlen(value));

// If we have a salt, add that to the digest as well
if(salt) {
   EVP_DigestUpdate(&mdctx,
                     salt,
                     (unsigned int) strlen(value));
}

// Create the hash
EVP_DigestFinal_ex(&mdctx,
                     md_value,
                     &md_len);

EVP_MD_CTX_cleanup(&mdctx);

for(i = 0; i < md_len; i++) {
    // copy the hex values into the buffer
    sprintf(&buffer[i*2], „%02x", md_value[i]);
}
...
```

```
// generate a salted SHA hash
GenerateHash(hashtype, finalRequestPW,     ...
```

```
NULL, buffer);
```

Table 3. *For size 3*

aaa	baa	caa	daa
aba	bba	cba	dba
aca	bca	cca	dca
ada	bda	cda	dda
aab	bab	cab	dab
...	...	...	...
adc	bdc	cdc	ddc
aad	bad	cad	dad
abd	bbd	cbd	dbd
acd	bcd	ccd	dcd
add	bdd	cdd	ddd

Listing 6. *SSHA Attack's usage statement*

```
Usage: ./ssha_attack -m mode [-d attack_dictionary_file | [-n min] -u max -a alphabet |
                      -a 20 -c custom_alphabet] -s SSHA_hash_string

-m  This is the mode for the prog to operate under.  The currently supported modes are
                      "dictionary" and "brute-force".  This switch is required.

-d  This option is to be used to engage "dictionary" mode. The dictionary is a regular
                      text file containing one entry per line. The data from this
                      file is what will be used as the clear text data to which the
                      discovered salt will get applied.

-l  The minimum amount of attack characters to begin with.

-u  The maximum amount of attack characters to use. If -l is not used processing will
                      start with size 1

-a  The numerical index of the attack alphabet to use:
        1. Numbers only
        2. lowercase hex
        3. UPPERCASE HEX
        4. lowercase alpha characters
        5. UPPERCASE ALPHA characters
        6. lowercase alphanumeric characters
        7. UPPERCASE ALPHANUMERIC characters
        8. lowercase & UPPERCASE ALPHA characters
        9. lowercase & UPPERCASE ALPHAnumeric characters
        10. All printable ASCII characters
        11. lowercase & UPPERCASE ALPHAnumeric characters, as well as:
            !"?$%^&*()_+-=[]{}'#@~,.<>?/|
        20. Custom alphabet - must be used with -c switch

-c  The custom attack alphabet to use, for example abcABC123!
Take note that this forces a permutation based process so the larger the alphabet the
                      longer the process will take. Also, when used with the -a 20
                      switch, but not the -u switch, the permutations are all based
                      on the size of the alphabet you submit. Using the example from
                      above all permutations would be 10 characters in length. This
                      can also force an incremental attack when coupled with the -n
                      switch

-s  The SSHA hash string that will be attacked.  This must be a Base64 encoded string.
                      This switch is required.
```

The GenerateHash function utilizes the OpenSSL libraries on a Linux system to generate the appropriate hash. The hashtype has already been dynamically established and it gets passed in as the first parameter to GenerateHash. In the GenerateHash function you will find code as seen in Listing 5.

As you can see (Listing 5) the last parameter passed in to GenerateHash (called buffer) will end up with the salted hash binary data after the algorithm has performed its one-way magic. This operation takes place for each clear text string from either your dictionary or the brute-force process. Then the final check that queries the 2 elements of data that will either establish whether a crack is successful or not looks like this:

```
...
// perform the actual comparison of
// formattedPW and buffer
if(strcmp(formattedPW, buffer) == 0) {
    // passwords matched
    return 1;
}
...
```

Using SSHA Attack

The link for SSHA Attack can be found in the *On the 'Net* section. Once the tarball has been downloaded, untar it in the standard fashion. There is a Makefile there for your convenience that basically abstracts the compilation and linking statements for you. The real statement to link the runnable program together is as follows: `gcc -O3` fucntions.o `ssha _ attack.o -lssl -o ssha _ attack`. This requires that you have already compiled the 2 files named functions.c and `ssha _ attack.c` into object files. The compilation statement looks like: `gcc -O3 -c -o` functions.o functions.c. But you can just use make on a Linux distro. Once you run the make utility with the included Makefile you should have an executable program in the same directory where you extracted the source files. This means that you should be able to invoke SSHA Attack with standard dot slash notation, ie. `./ssha _ attack` from the same directory where you ran the make utility.

Running SSHA Attack with the `-help` switch gives you further information on the usage. Listing 6 shows the output of such an action. Decide on your attack methodology, you currently have 2 choices of either dictionary or brute-force.

The dictionary attack is self-explanatory, you have to also provide the dictionary file with the `-d` switch. The dictionary is basically a list of strings (one per line) that will each get the salt applied to them and then get hashed with the appropriate algorithm.

The brute-force mode is a little more complicated as you must tell the program what alphabet you want to use. You can choose from a pre-constructed set using the `-a` switch or roll your own with the `-c` switch. Rolling your own has proved interesting for some Tiger Teams that have some inkling of possible characters used (based on shoulder surfing or an understanding of personal habits/history) but know an entire password.

Using a pre-constructed alphabet will kick-off the generation of combinations (as in the *Cartesian Product Algorithm*) of the data to be used. For instance, using an alphabet of abcd and a min – max combination of 1- 4 will yield the following as the clear text data set to use with the already extracted salt: Tabele 1, Tabele 2, Tabele 3, Tabele 4.

Using a custom alphabet forces the generation of all permutations of the data set at hand. For instance using an alphabet of *abcd* the generated clear text data set would be as presented in Table 5:

To give an example of what a real world run would be like let's generate some hashes first. For the sake of this example here is an output of the Python script that generates multiple SHA family hashes. To keep things simple I have used a small (4 characters long) clear text string of `T35t`.

Once these hashes are generated we will use SSHA Attack against them. In the real word we would obviously not know the clear text value but this is just an example for educational purposes. When analysing the work factor for these simple collisions, understand that these examples were run on a dual-processor (Pentium(R) D 2.8 GHz) Linux based VMWare image with 768 MB RAM. During run time an instance

Table 4. *For size 4*

aaaa	baaa	caaa	daaa
abaa	bbaa	cbaa	dbaa
acaa	bcaa	ccaa	dcaa
adaa	bdaa	cdaa	ddaa
aaba	baba	caba	daba
...	...	...	...
adcd	bdcd	cdcd	ddcd
aadd	badd	cadd	dadd
abdd	bbdd	cbdd	dbdd
acdd	bcdd	ccdd	dcdd
addd	bddd	cddd	dddd

Table 5. *Using the Custom Alphabet feature with an alphabet of „abcd"*

aaaa	bbbb	cccc	dddd
abcd	abdc	acbd	acdb
adcb	adbc	bacd	badc
bcad	bcda	bdca	bdac
cbad	cbda	cabd	cadb
cdab	cdba	dbca	dbac
dcba	dcab	dacb	dabc

Table 6. *Run-time summary for Listing 7*

SHA Algorithm	Time (in seconds) for collision at 4 bytes
SHA1	22
SHA256	29
SHA385	33
SHA512	35

On the 'Net

- http://cryptography.hyperlink.cz/MD5_collisions.html,
- http://www.mscs.dal.ca/~selinger/md5collision/,
- http://www.stachliu.com.nyud.net:8090/collisions.html,
- http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html,
- http://www.rsa.com/rsalabs/node.asp?id=2927,
- https://www.iaik.at/research/krypto/collision/SHA1Collision_Description.php,
- http://sourceforge.net/projects/ssha-attack.

RUNNING SHORT ON SNORT®?

Are your sensors sucking wind?

Speed up your IDS deployments on multi-gigabit Ethernet segments by up to 16X, with hardware solutions from Endace.

Standard source code. Full preprocessing. Your complete ruleset. Faster Snort without the run around.

Ensure your biggest vulnerability is not your server.

Accelerate Snort with NinjaBox-Z.

www.endace.com/accelerate

endace accelerated

SNORT® is a registered trademark of Sourcefire, Inc of the Linux utility *top* showed that the memory usage of SSHA Attack program never surpassed 0.1% while the CPU usage was well over 90%. The following table summarizes the run time correlated with the hash type/size (Table 6).

Listing 7. *Generating salted hashes, LDAP style, and then cracking them*

```
python genSSHA_py25.py T35t 4
SHA1
Base64 encoded: XjW3J0gbK+nkHDwCdLsksYxx/50wYmJm
SHA256
Base64 encoded: DP8Qwmb5LP1Br1H3EoJ/F7MXJwY9IPt8w3MiDm9r72QwYmJm
SHA384
Base64 encoded:
Yn19q3hVFGN8xUkfvfbCfZg7cZ6d3wqN2vl99Ezuxjd9M0N4y8s6LN+ihIAxWV2tMGJiZg==

SHA512
Base64 encoded:
G1kSnef8EObDZdmlSHhO911J8TWP5eL0jGCtHbG83NNhpWtV34fv8wuF3gOP/N37+RM0dbr8TP28ZQlkxKr0r
                      DBiYmY=
We will use these hashes here as an example of using SSHA Attack to try to discover
                      collisions, in essence cracking the clear text component
                      represented by a salted hash.
./ssha_attack -m brute-force -u 8 -a 9 -s XjW3J0gbK+nkHDwCdLsksYxx/50wYmJm
Hash Algorithm Detected: SHA1

Trying Word Length: 1
No hits for Word Length: 1
...
Trying Word Length: 4

There is a match on value "T35t"
Elapsed time in seconds for successful attack: 22

./ssha_attack -m brute-force -u 8 -a 9 -s DP8Qwmb5LP1Br1H3EoJ/F7MXJwY9IPt8w3MiDm9r72Q
                      wYmJm
Hash Algorithm Detected: SHA256

Trying Word Length: 1
No hits for Word Length: 1
...
Trying Word Length: 4

There is a match on value "T35t"
Elapsed time in seconds for successful attack: 29

./ssha_attack -m brute-force -u 8 -a 9 -s Yn19q3hVFGN8xUkfvfbCfZg7cZ6d3wqN2vl99Ezuxjd9
                      M0N4y8s6LN+ihIAxWV2tMGJiZg==
Hash Algorithm Detected: SHA384

Trying Word Length: 1
No hits for Word Length: 1
...
Trying Word Length: 4

There is a match on value "T35t"
Elapsed time in seconds for successful attack: 33

./ssha_attack -m brute-force -u 8 -a 9 -s G1kSnef8EObDZdmlSHhO911J8TWP5eL0jGCtHbG83NNh
                      pWtV34fv8wuF3gOP/N37+RM0dbr8TP28ZQlkxKr0rDBiYmY=
Hash Algorithm Detected: SHA512

Trying Word Length: 1
No hits for Word Length: 1
...
Trying Word Length: 4

There is a match on value "T35t"
Elapsed time in seconds for successful attack: 35
```

Conclusion

Using random salt elements when storing sensitive data is a solid practice and it is a wise part of a layered defence architecture but it is not a panacea. An attacker can be crafty in terms of extracting salt values from either embedded methods such as the typical LDAP model analysed in this article or other storage techniques. The salt needs to be available for legitimate use within an application and by the same token it is available to an attacker, therefore salted hashes are susceptible to cracking attacks as shown in this article. As tools get more and more sophisticated, password and clear-text data protection will become more and more challenging. There are tools out there to easily and quickly crack unsalted one-way hashes. Now a new generation of cracking tools are appearing and these target the more difficult areas of sensitive data. Do not be surprised if these tools also start to utilize sophisticated programming techniques based on distributed computing so as to increase their efficiency exponentially. A perfect example of this would be the BOINC based project to research collisions with unsalted SHA-1 hashes.

The BOINC Project brings about the power of distributed computing to the world. This is done in an open source fashion through volunteers donating computing power for the solving of computationally intense and complex problems. You can get further details on this project at: *http://boinc.berkeley.edu/*. The communities that were intended to utilise such work were originally scientific ones, but the computer science community has realized the benefits of tapping into a grid based computing platform for computationally intensive areas such as cracking encryption schemes. Somewhat relevant to this article is the SHA-1 Collision Search project, details can found at: *http://boinc.iaik.tugraz.at/sha1_coll_search/*. This is only one example and many more interesting efforts can be seen at: *http://distributedcomputing.info/ap-crypto.html*

About the Author
Andres Andreu has been working in the software engineering/architecture arena for many years now building global web based solutions for U.S. Government entities and corporations alike. He is also heavily involved in the Web applications security and pen testing space and is the author of the OWASP WSFuzzer and SSHA Attack programs as well as the book entitled Professional Pen Testing for Web Applications (ISBN-13: 978-0471789666).